

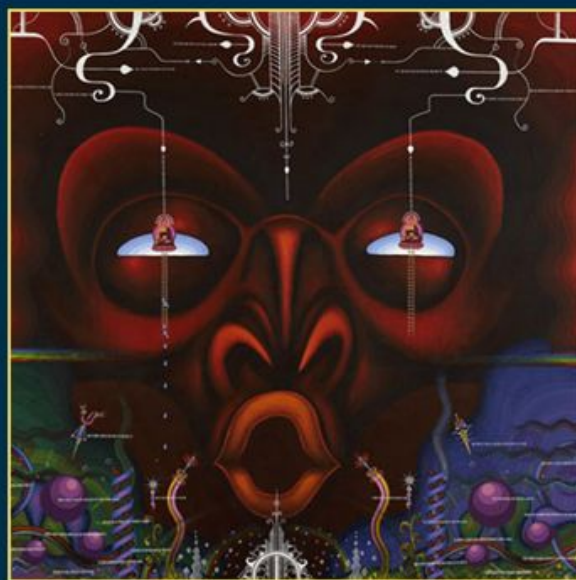
Festschrift

LNCS 7000

Gul Agha
Olivier Danvy
José Meseguer (Eds.)

Formal Modeling: Actors, Open Systems, Biological Systems

Essays Dedicated to Carolyn Talcott
on the Occasion of Her 70th Birthday



 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Gul Agha Olivier Danvy
José Meseguer (Eds.)

Formal Modeling: Actors, Open Systems, Biological Systems

Essays Dedicated to Carolyn Talcott
on the Occasion of Her 70th Birthday

Volume Editors

Gul Agha

José Meseguer

University of Illinois

Thomas M. Siebel Center for Computer Science

201 N. Goodwin Avenue, MC 258, Urbana, IL 61801, USA

E-mail: {agha,meseguer}@illinois.edu

Olivier Danvy

Aarhus University

Department of Computer Science

Åbogade 34, 8200 Aarhus N, Denmark

E-mail: danvy@cs.au.dk

The illustration appearing on the cover of this book is the work of Daniel Rozenberg (DADARA).

ISSN 0302-9743

e-ISSN 1611-3349

ISBN 978-3-642-24932-7

e-ISBN 978-3-642-24933-4

DOI 10.1007/978-3-642-24933-4

Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011938996

CR Subject Classification (1998): D.2, F.3, D.3, C.2, D.2.4, C.2.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)



Dr. Carolyn Talcott

Preface

This volume contains the papers presented at a symposium in honor of Carolyn Talcott held during November 3–4, 2011 in Menlo Park, California.

Carolyn Talcott, who celebrated her 70th birthday in 2011, is a leading researcher and mentor of international renown among computer scientists. Dr. Talcott has made key contributions to a number of areas of computer science including:

1. Semantics and verification of programming languages
2. Foundations of Actor-based systems
3. Middleware and meta-architectures
4. Maude and rewriting logic
5. Computational biology

Dr. Talcott’s earliest contributions to the semantics and verification of programming languages started with her PhD thesis and continued with her work on the Actor model. Her thesis addressed the challenging problem of formalizing reasoning about state change in high-level languages like LISP. The proof methods she developed for reasoning about state change are widely cited; she was recognized as a leading figure in the field, serving in key positions such as Co-Editor-in-Chief of Springer’s journal *LISP and Symbolic Computation* and then *Higher-Order and Symbolic Computation* (HOSC), and in roles such as chair or co-organizer of many scientific meetings in the field.

Dr. Talcott made substantial contributions to advancing the formal development of the Actor model. Actors are a foundational model of concurrency; they capture the asynchronous nature of parallel and distributed systems, and provide the flexibility needed to build open, extensible concurrent systems. In recent years, the Actor model has acquired increasing importance and use, providing a basis for a number of programming languages and frameworks. The growth of the model is due to the fact that Actors go a long way toward addressing the challenges of programmability in systems such as Web services, cloud computing and scalable multicore processor architectures. Her seminal contributions to the foundations and formal reasoning techniques for Actors not only defines the state of the art today but provides the foundation for future developments in this field.

Computer systems and applications are not only increasingly distributed, they also need to deal with changing physical constraints such as energy and real-time requirements, sensing and actuation control loops, as well as security, reliability, etc. Designing such systems so that they can flexibly adapt to changing conditions and remain resilient and safe is an enormous challenge. Dr. Talcott’s contributions address this challenge by developing methods for reasoning about novel distributed object reflection techniques whereby “meta-objects” can monitor and control the runtime state of other objects (which could in turn

themselves be meta-objects controlling lower-level meta-objects). The techniques she developed are not only mathematically well-founded, they provide practical methods for building adaptive middleware. In particular, her work on the Two-Level Actor Machine (TLAM) model, and on the “Russian Dolls” model of distributed reflection is well known. These methods are having, and will continue to have, a significant impact in the emerging area of cyber-physical systems.

During her tenure as a senior scientist at Stanford University, Dr. Talcott began research which has led to a series of key conceptual contributions to rewriting logic and Maude—arguably the most advanced executable formal specification language currently available. These contributions included definition of the semantics of Actors and Actor languages in rewriting logic, and the development of formal reasoning systems. After her retirement from Stanford, Dr. Talcott moved to SRI in 2001 to head the Maude team and her contributions have been even more significant. Thanks in no small part to these contributions, Maude has gained a scientific network consisting of several universities across Europe, as well as institutions in the USA. A Springer LNCS Tutorial volume on Maude was published in 2007, with significant contributions to this volume made by Dr. Talcott. She has also been an important contributor to many of the new releases of the Maude software; such releases are regularly made as new features are incorporated. The field of rewriting logic is now firmly established with regular scientific conferences as well as hundreds of peer-reviewed publications in the area.

Dr. Talcott’s move to SRI has been fruitful for the area of computational biology: at SRI, she has led a remarkably productive collaboration between molecular biologists and computer scientists. Specifically, Dr. Talcott has played a key leadership role in advancing this entire field by the application of formal methods to systems biology. She has initiated the Pathway Logic Project which has made many contributions, not only conceptual ones, but also by developing practical tools that biologists find easy to understand and use; these tools enable visualization and efficient formal analysis of biological systems.

Over the years, Dr. Talcott has collaborated with a large number of researchers across the globe, among them the editors of this volume. Not surprisingly, some of the papers we were able to include in this volume have their genesis in such collaborations. Her impact, beyond her technical contributions, includes the scores of researchers in the computer science community whom she has inspired over the years.

It is our good fortune to be able to organize this Festschrift in honor of Dr. Carolyn Talcott and we look forward to many more years of her leadership as an innovative researcher, valued colleague and inspiring mentor.

August 2011

Gul Agha
Olivier Danvy
José Meseguer

Table of Contents

Essays on Carolyn Talcott

Two PhD Students for the Price of One	1
<i>Solomon Feferman</i>	
Honoring Carolyn Talcott’s Contributions to Science	4
<i>Sylvan Pinsky</i>	

Actors and Programming Languages

Ten Years of Analyzing Actors: Rebeca Experience	20
<i>Marjan Sirjani and Mohammad Mahdi Jaghoori</i>	
Mathematical Models of Object-Based Distributed Systems	57
<i>Carlos Henrique C. Duarte</i>	
From Explicit to Symbolic Types for Communication Protocols in CCS	74
<i>Hanne Riis Nielson, Flemming Nielson, Jörg Kreiker, and Henrik Pilegaard</i>	
Abstract LR-Parsing	90
<i>Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt</i>	

Cyberphysical Systems

Fractionated Software for Networked Cyber-Physical Systems: Research Directions and Long-Term Vision	110
<i>Mark-Oliver Stehr, Carolyn Talcott, John Rushby, Pat Lincoln, Minyoung Kim, Steven Cheung, and Andy Poggio</i>	
Model Feasible Interactions in Distributed Real-Time Systems	144
<i>Shangping Ren, Yue Yu, and Miao Song</i>	

Middleware and Meta-architectures

Puff, The Magic Protocol	169
<i>Farhad Arbab</i>	
A Formal Methodology for Compositional Cross-Layer Optimization	207
<i>Minyoung Kim, Mark-Oliver Stehr, Carolyn Talcott, Nikil Dutt, and Nalini Venkatasubramanian</i>	

From Service Identification to Service Selection: An Interleaved Perspective	223
<i>Devis Bianchini, Francesco Pagliarecci, and Luca Spalazzi</i>	
Towards a System Model for Ensembles	241
<i>Matthias Hözl and Martin Wirsing</i>	
Algorithmic Aspects of Risk Management	262
<i>Ashish Gehani, Lee Zaniewski, and K. Subramani</i>	
Formal Methods and Reasoning Tools	
Parameterized Metareasoning in Membership Equational Logic	277
<i>Manuel Clavel, Narciso Martí-Oliet, and Miguel Palomino</i>	
Fast Sort Computations for Order-Sorted Matching and Unification	299
<i>Steven Eker</i>	
Solving the First Verified Software Competition Problems Using PVS	315
<i>Sam Owre and Natarajan Shankar</i>	
Towards a Maude Formal Environment	329
<i>Francisco Durán, Camilo Rocha, and José María Álvarez</i>	
Multisimulations: Towards Next Generation Integrated Simulation Environments	352
<i>Leila Jalali, Sharad Mehrotra, and Nalini Venkatasubramanian</i>	
Semantics, Simulation, and Formal Analysis of Modeling Languages for Embedded Systems in Real-Time Maude	368
<i>Peter Csaba Ölveczky</i>	
Computational Biology	
Computational Biology: A Programming Perspective	403
<i>Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen, and Søren Bjerregaard Vrist</i>	
Applications of Pathway Logic Modeling to Target Identification	434
<i>Anupama Panikkar, Merrill Knapp, Huaiyu Mi, Dave Anderson, Krishna Kodukula, Amit K. Galande, and Carolyn Talcott</i>	
Author Index	447

Publications of Dr. Carolyn Talcott

(Note - some publications as C.T. Williamson)

- [1] Abate, A., Bai, Y., Sznajder, N., Talcott, C., Tiwari, A.: Quantitative and probabilistic modeling in Pathway Logic. In: IEEE 7th International Symposium on Bioinformatics and Bioengineering, pp. 922–929. IEEE (2007)
- [2] Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: Towards a theory of actor computation. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 565–579. Springer, Heidelberg (1992)
- [3] Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. *Journal of Functional Programming* 7, 1–72 (1997)
- [4] Amaral, A.M.S.C., Linnett, J.W., Williamson, C.T.: The double bond in ethylene. *Theoretical Chimica Acta* 16, 249–262 (1970)
- [5] Arbab, F., Talcott, C. (eds.): COORDINATION 2002. LNCS, vol. 2315. Springer, Heidelberg (2002)
- [6] Bronstein, A., Talcott, C.L.: Formal verification of pipelines based on string-functional semantics. In: IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium (1989)
- [7] Bronstein, A., Talcott, C.L.: Formal verification of synchronous circuits based on string-functional semantics: The 7 paillet circuits in boyer-moore. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 317–333. Springer, Heidelberg (1990)
- [8] Burgoyne, N., Williamson, C.: Some computations involving simple lie algebras. In: Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, pp. 162–171 (1971)
- [9] Burgoyne, N., Williamson, C.: Semi-simple classes in chevalley type groups. *Pacific Journal of Mathematics* 70, 83–100 (1977)
- [10] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.0 Manual (2003), <http://maude.cs.uiuc.edu>
- [11] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: The Maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
- [12] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Unification and narrowing in maude 2.4. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 380–390. Springer, Heidelberg (2009)
- [13] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
- [14] Coglio, A., Giunchiglia, F., Meseguer, J., Talcott, C.L.: Composing and Controlling Deduction in Reasoning Theories Using Mappings. In: Kirchner, H. (ed.) FroCos 2000. LNCS, vol. 1794, pp. 200–216. Springer, Heidelberg (2000)
- [15] Coglio, A., Giunchiglia, F., Pecchiari, P., Talcott, C.L.: A logic level specification of the NQTHM simplification process. Technical report, IRST, University of Genova, Stanford University (1997)
- [16] Denker, G., García-Luna-Aceves, J.J., Meseguer, J., Ölveczky, P.C., Raju, J., Smith, B., Talcott, C.L.: Specifications and analysis of a reliable broadcasting protocol in Maude. In: Hajek, B., Sreenivas, R.S. (eds.) 37th Allerton Conference on Communication, Control, and Computing, pp. 738–747 (1999) Case study details, <http://maude.cs1.sri.com/casestudies/rbp/>

- [17] Denker, G., Meseguer, J., Talcott, C.L.: Protocol specification and analysis in Maude. In: Workshop on Formal Methods and Security Protocols (June 1998), <http://www.cs.bell-labs.com/who/nch/fmsp/index.html>
- [18] Denker, G., Meseguer, J., Talcott, C.L.: Rewriting Semantics of Distributed Meta Objects and Composable Communication Services (1999) (submitted)
- [19] Denker, G., Meseguer, J., Talcott, C.L.: Formal specification and analysis of active networks and communication protocols: The Maude experience. In: DARPA Information Survivability Conference and Exposition (DISCEX 2000), vol. 1, pp. 251–265. IEEE (2000)
- [20] Denker, G., Meseguer, J., Talcott, C.L.: Rewriting semantics of distributed meta objects and composable communication services. In: Third International Workshop on Rewriting Logic and Its Applications (WRLA 2000), Kanazawa, Japan, September 18-20. Electronic Notes in Theoretical Computer Science, vol. 36. Elsevier (2000), <http://www.elsevier.nl/locate/entcs/volume36.html>
- [21] Denker, G., Talcott, C.L.: Formal checklists for remote agent dependability. In: Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004). Electronic Notes in Theoretical Computer Science. Elsevier (2004)
- [22] Denker, G., Talcott, C.L.: A formal framework for goal net analysis. In: Workshop on Verification and Validation of Planning Systems. AAAI (2005)
- [23] Denker, G., Talcott, C., Ghanadan, R., Kumar, S.: An architecture for policy-based cognitive tactical networking. In: Military Communications Conference, MILCOM (2006)
- [24] di Blasio, P., Fisher, K., Talcott, C.: A control-flow analysis for a calculus of concurrent objects. Transactions in Software Engineering, TSE (2000)
- [25] di Blasio, P., Fisher, K., Talcott, C.L.: A control-flow analysis for a calculus of concurrent objects. In: Bowman, H., Derrick, J. (eds.) Formal Methods for Open Object-based Distributed Systems, vol. 2, pp. 73–88. Chapman & Hall (1997)
- [26] Dill, D.L., Knapp, M.A., Gage, P., Talcott, C., Lincoln, P., Laderoute, K.: The pathalyzer: A tool for analysis of signal transduction pathways. In: Eskin, E., Ideker, T., Raphael, B., Workman, C. (eds.) RECOMB 2005. LNCS (LNBI), vol. 4023, pp. 11–22. Springer, Heidelberg (2007)
- [27] Donaldson, R., Talcott, C., Knapp, M., Calder, M.: Understanding signalling networks as collections of signal transduction pathways. In: Computational Methods in Systems Biology (2010)
- [28] Duarte, C.H.C., Talcott, C.L.: Clara: An actor language for high performance distributed computing. In: Proc. Brazilian Symposium on Computer Architecture – High Performance Computing (SBAC-PAD 2000), Sao Pedro, SP, Brazil (2000)
- [29] Duran, F., Eker, S., Escobar, S., Meseguer, J., Talcott, C.: Variants, unification, narrowing, and symbolic reachability in maude 2.6. In: Rewriting Techniques and Applications (2011)
- [30] Eker, S., Laderoute, K., Lincoln, P., Sriram, M.G., Talcott, C.: Representing and simulating protein functional domains in signal transduction using MAUDE. In: Priami, C. (ed.) CMSB 2003. LNCS, vol. 2602, pp. 164–165. Springer, Heidelberg (2003)
- [31] Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Talcott, C.: Pathway Logic: Executable models of biological networks. In: Fourth International Workshop on Rewriting Logic and Its Applications (WRLA 2002), Pisa, Italy, September 19-21. Electronic Notes in Theoretical Computer Science, vol. 71, Elsevier (2002), <http://www.elsevier.nl/locate/entcs/volume71.html>

- [32] Ekins, S., Freundlich, J.S., Choi, I., Sarker, M., Talcott, C.: Computational databases, pathway and cheminformatics tools for tuberculosis drug discovery. *Trends in Microbiology* 19(2) (February 2011)
- [33] Galbiati, L., Talcott, C.L.: A Simplifier for Untyped Lambda Expressions. In: Okada, M., Kaplan, S. (eds.) *CTRS 1990. LNCS*, vol. 516, pp. 342–353. Springer, Heidelberg (1991)
- [34] Galbiati, L., Talcott, C.L.: A simplifier for untyped lambda expressions. Technical Report STAN-CS-90-1337, Computer Science Department, Stanford University (1990)
- [35] Giunchiglia, F., Pecchiari, P., Talcott, C.L.: Reasoning theories: Towards an architecture for open mechanized reasoning systems. Technical Report 9409-15, IRST, Also appears as Stanford University Computer Science Department Technical Note STAN-CS-94-TN-15 (November 1994)
- [36] Giunchiglia, F., Pecchiari, P., Talcott, C.L.: Reasoning theories: Towards an architecture for open mechanized reasoning systems. In: *Workshop on Frontiers of Combining Systems FRODOS 1996* (1996)
- [37] Giunchiglia, F., Pecchiari, P., Talcott, C.L.: Reasoning theories: Towards an architecture for open mechanized reasoning systems (1996) (submitted for publication)
- [38] Higher Order Operational Techniques in Semantics II, *Electronic Notes in Theoretical Computer Science*. Elsevier (1998), <http://www.elsevier.nl/locate/entcs/volume10.html>
- [39] Greco, M.A., Murray, J., Talcott, C.: Modeling sleep-related activities from experimental observations - initial computational frameworks for understanding sleep function(s). In: *AHFE*. Taylor and Francis, LLC (2010)
- [40] Gutierrez-Nolasco, S., Venkatasubramanian, N., Stehr, M.-O., Talcott, C.L.: Towards adaptive secure group communication: Bridging the gap between formal specification and network simulation. In: *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*, pp. 113–120. IEEE Computer Society (2006)
- [41] Gutierrez-Nolasco, S., Venkatasubramanian, N., Talcott, C., Stehr, M.-O.: Tailoring group membership consistency for mobile networks. In: *CTS* (2011)
- [42] Heiser, L.M., Wang, N.J., Talcott, C.L., Laderoute, K.R., Knapp, M., Guan, Y., Hu, Z., Ziyad, S., Weber, B.L., Laquerre, S., Jackson, J.R., Wooster, R.F., Kuo, W.-L., Gray, J.W., Spellman, P.T.: Integrated analysis of breast cancer cell lines reveals unique signaling pathways. *Genome Biology* 10, R31 (2009)
- [43] Honsell, F., Mason, I.A., Smith, S.F., Talcott, C.L.: A theory of classes for a functional language with effects. In: Martini, S., Börger, E., Kleine Büning, H., Jäger, G., Richter, M.M. (eds.) *CSL 1992. LNCS*, vol. 702, pp. 309–326. Springer, Heidelberg (1993)
- [44] Honsell, F., Mason, I.A., Smith, S.F., Talcott, C.L.: A variable typed logic of effects. *Information and Computation* 119(1), 55–90 (1995)
- [45] Iida, S., Denker, G., Talcott, C.: Document logic: Risk analysis of business processes through document authenticity. In: *DDBP. IEEE Digital Library* (2009)
- [46] Iida, S., Denker, G., Talcott, C.: Document logic: Risk analysis of business processes through document authenticity. *Journal of Research and Practice in Information Technology* (2011)
- [47] Iyengar, S.M., Talcott, C., Mozzachiodi, R., Cataldo, E., Baxter, D.A.: Executable symbolic models of neural processes. In: *Network Tools and Applications in Biology, NETTAB 2007* (2007)

- [48] Jones, N., Talcott, C. (eds.): Proceedings of The Atlantique Workshop on Semantics Based Program Manipulation, University of Copenhagen DIKU Technical Report 94/12 (1994)
- [49] Katz, T.J., Talcott, C.L.: The cyclononatetraene anion radical. *Journal of the American Chemical Society* 88, 4732 (1966)
- [50] Khakpour, N., Jalili, S., Talcott, C., Sirjani, M., Mousavi, M.R.: Pobsam: Policy-based managing of actors in self-adaptive systems. In: *Formal Aspects of Component Software (FACS)*. *Electronic Notes in Theoretical Computer Science* (2009)
- [51] Kim, M., Stehr, M.-O., Talcott, C.: A distributed logic for networked cyber-physical systems. In: *Foundations of Software Engineering*. LNCS. Springer, Heidelberg (2011)
- [52] Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: Combining formal verification with observed system execution behavior to tune system parameters. In: *Formal Methods for Open Object-based Distributed Systems*. Springer, Heidelberg (2007)
- [53] Kim, M.-Y., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: A probabilistic formal analysis approach to cross layer optimization in distributed embedded systems. In: *Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007*. LNCS, vol. 4468, pp. 285–300. Springer, Heidelberg (2007)
- [54] Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: Constraint refinement for online verifiable cross-layer system adaptation. In: *IEEE/ACM Design Automation and Test in Europe (DATE 2008)*. IEEE/ACM (2008)
- [55] Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: xtune: A formal methodology for cross-layer tuning of mobile embedded systems. *Transactions on Embedded Computing Systems* (2011)
- [56] Lincoln, P.D., Talcott, C.: Symbolic systems biology and pathway logic. In: *Iyengar, S. (ed.) Symbolic Systems Biology*. Jones and Bartlett (2010)
- [57] Mason, I.A., Pehoushek, J.D., Talcott, C.L., Weening, J.S.: *A Qlisp Primer*. Technical Report STAN-CS-90-1340, Department of Computer Science, Stanford University (1990)
- [58] Mason, I.A., Smith, S.F., Talcott, C.L.: From Operational Semantics to Domain Theory. *Information and Computation* 128(1), 26–47 (1996)
- [59] Mason, I.A., Talcott, C.L.: Memories of S-expressions: Proving properties of Lisp-like programs that destructively alter memory. Technical Report STAN-CS-85-1057, Department of Computer Science, Stanford University (1985)
- [60] Mason, I.A., Talcott, C.L.: Axiomatizing operational equivalence in the presence of side effects. In: *Fourth Annual Symposium on Logic in Computer Science*. IEEE (1989)
- [61] Mason, I.A., Talcott, C.L.: Programming, transforming, and proving with function abstractions and memories. In: *Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989*. LNCS, vol. 372, pp. 574–588. Springer, Heidelberg (1989)
- [62] Mason, I.A., Talcott, C.L.: A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Department of Computer Science, Stanford University (1989)
- [63] Mason, I.A., Talcott, C.L.: Program transformation for configuring components (1990)
- [64] Mason, I.A., Talcott, C.L.: Reasoning about programs with effects. In: *Deransart, P., Maluszyński, J. (eds.) PLILP 1990*. LNCS, vol. 456, pp. 189–203. Springer, Heidelberg (1990)

- [65] Mason, I.A., Talcott, C.L.: Equivalence in functional languages with effects. *Journal of Functional Programming* 1, 287–327 (1991)
- [66] Mason, I.A., Talcott, C.L.: Program transformation for configuring components. In: *ACM/IFIP Symposium on Partial Evaluation and Semantics-based Program Manipulation* (1991)
- [67] Mason, I.A., Talcott, C.L.: Program transformation via constraint propagation (1991)
- [68] Mason, I.A., Talcott, C.L.: Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science* 105(2), 167–215 (1992)
- [69] Mason, I.A., Talcott, C.L.: References, local variables and operational reasoning. In: *Seventh Annual Symposium on Logic in Computer Science*, pp. 186–197. IEEE (1992)
- [70] Mason, I.A., Talcott, C.L.: Program transformation via contextual assertions. In: Jones, N.D., Hagiya, M., Sato, M. (eds.) *Logic, Language and Computation*. LNCS, vol. 792, pp. 225–254. Springer, Heidelberg (1994)
- [71] Mason, I.A., Talcott, C.L.: Reasoning about object systems in VTLoE. *International Journal of Foundations of Computer Science* 6(3), 265–298 (1995)
- [72] Mason, I.A., Talcott, C.L.: A semantically sound actor translation. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) *ICALP 1997*. LNCS, vol. 1256, pp. 369–378. Springer, Heidelberg (1997)
- [73] Mason, I.A., Talcott, C.L.: Landin-feferman logic. In: *The Fourteenth Workshop on the Mathematical Foundations of Programming Semantics, MFPS 14* (1998)
- [74] Mason, I.A., Talcott, C.L.: Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science* 220, 409–467 (1999)
- [75] Mason, I.A., Talcott, C.L.: Simple network protocol simulation within Maude. In: *Third International Workshop on Rewriting Logic and Its Applications (WRLA 2000)*, Kanazawa, Japan, September 18–20. *Electronic Notes in Theoretical Computer Science*, vol. 36. Elsevier (2000), <http://www.elsevier.nl/locate/entcs/volume36.html>
- [76] Mason, I.A., Talcott, C.L.: Feferman–Landin Logic. In: Sieg, W., Sommer, R., Talcott, C.L. (eds.) *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*. *Lecture Notes in Logic*, pp. 299–344. Association of Symbolic Logic (2002)
- [77] Mason, I.A., Talcott, C.L.: IOP: The InterOperability Platform & IMAude: An interactive extension of Maude. In: *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004)*. *Electronic Notes in Theoretical Computer Science*. Elsevier (2004)
- [78] Mason, I.A., Talcott, C.: Actors and logical analysis of interactive system. In: Viroli, M. (ed.) *Foundations of Interactive Computation (FInCo 2005)*. *Electronic Notes in Theoretical Computer Science*, vol. 141. Elsevier (2005)
- [79] Maurer, W.D., Williamson, C.T.: Algorithm verification applied to the todd-coxeter algorithm. Technical Report ERL-M317, Electronics Research Lab., College of Engineering, U.C. Berkeley (1971)
- [80] Meseguer, J., Olveczky, P.C., Stehr, M.-O., Talcott, C.L.: Maude as a wide-spectrum framework for formal modeling and analysis of active networks. In: *DARPA Active Networks Conference and Exposition (DANCE)*, pp. 494–510. IEEE (May 2002)
- [81] Meseguer, J., Talcott, C.L.: Rewriting logic and secure mobility. In: *NPS Workshop on Active Networks*, Monterey, CA (February 1997)

- [82] Meseguer, J., Talcott, C.L.: Formal foundations for compositional software architectures. Position paper for Workshop on Compositional Software Architectures, Monterey, CA (January 1998)
- [83] Meseguer, J., Talcott, C.L.: Mapping OMRS to Rewriting Logic. In: Kirchner, C., Kirchner, H. (eds.) 2nd International Workshop on Rewriting Logic and Its Applications, WRLA 1998. Electronic Notes in Theoretical Computer Science, vol. 15. Elsevier (1998), <http://www.elsevier.nl/locate/entcs/volume15.html>
- [84] Bevilacqua, V., Talcott, C.: A Partial Order Event Model for Concurrent Objects. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 415–430. Springer, Heidelberg (1999)
- [85] Meseguer, J., Talcott, C.L.: Semantic models for distributed object reflection. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
- [86] Meyers, R.J., Talcott, C.L.: Electron spin resonance of the radical anions of pyridine and related nitrogen heterocyclics. *Molecular Physics* 12, 549–567 (1967)
- [87] Montanari, U., Talcott, C.L.: Can actors and π -agents live together? In: Higher Order Operational Techniques in Semantics II. Electronic Notes in Theoretical Computer Science. Elsevier (1997), <http://www.elsevier.nl/locate/entcs/volume10.html>
- [88] Nagayama, M., Talcott, C.: An nqthm mechanization of “an exercise in the verification of multi-process programs”. Technical Report STAN-CS-91-1370, Computer Science Department, Stanford University (1991)
- [89] Ölveczky, P.C., Keaton, M., Meseguer, J., Talcott, C., Zabele, S.: Specification and analysis of the AER/NCA active network protocol suite in real-time maude. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 333–347. Springer, Heidelberg (2001), <http://maude.csl.sri.com/papers>
- [90] Ölveczky, P.C., Meseguer, J., Talcott, C.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In: Formal Methods in System Design (2006)
- [91] Pagliarecci, F., Spalazzi, L., Stehr, M.-O., Talcott, C.: Formal specification of agent-object oriented programs. In: Symposium on Collaborative Technologies and Systems (2008)
- [92] Santiago, S., Talcott, C., Escobar, S., Meadows, C., Meseguer, J.: A graphical user interface for Maude-NPA. In: Spanish Conference on Programming and Computer Languages (PROLE). ENTCS (2009)
- [93] Sarker, M., Chopra, S., Mortelmans, K., Kodukula, K., Talcott, C., Galande, A.K.: Systems level in silico pathway analysis predicts metabolites that are potential antimicrobial targets. *Journal of Computer Science and Systems Biology* (accepted, April 2011)
- [94] Shmatikov, V., Talcott, C.: Reputation-based trust management. In: 2003 IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS Workshop on Issues in the Theory of Security, WITS 2003 (2003)
- [95] Shmatikov, V., Talcott, C.: Reputation-based trust management. *Journal of Computer Security* (2004)
- [96] Sieg, W., Sommer, R., Talcott, C. (eds.): Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman. *Lecture Notes in Logic*, vol. 15. Association for Symbolic Logic (2002)
- [97] Sieg, W., Sommer, R., Talcott, C. (eds.): Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman. LNL, vol. 15. Association for Symbolic Logic (2002)

- [98] Smith, S.F., Talcott, C.L.: Modular reasoning for actor specification diagrams. In: Ciancariani, P., Fantechi, A., Gorrieri, R. (eds.) *Formal Methods for Open Object-based Distributed Systems*, pp. 313–330. Kluwer (1999)
- [99] Smith, S.F., Talcott, C.L. (eds.): *Formal Methods for Open Object-based Distributed Systems*, vol. 4. Kluwer (2000)
- [100] Smith, S.F., Talcott, C.L.: Specification diagrams for actor systems. *Higer-Order and Symbolic Computation* 15(4), 301–348 (2002)
- [101] Stehr, M.-O., Kim, M., Talcott, C.: Toward distributed declarative control of networked cyber-physical systems. In: Yu, Z., Liscano, R., Chen, G., Zhang, D., Zhou, X. (eds.) *UIC 2010. LNCS*, vol. 6406, pp. 397–413. Springer, Heidelberg (2010)
- [102] Stehr, M.-O., Talcott, C.: PLAN in Maude: Specifying an active network programming language. In: *Fourth International Workshop on Rewriting Logic and Its Applications (WRLA 2002)*, Pisa, Italy, September 19-21. *Electronic Notes in Theoretical Computer Science*, vol. 71, Elsevier (2002), <http://www.elsevier.nl/locate/entcs/volume71.html>
- [103] Stehr, M.-O., Talcott, C.: Practical techniques for language design and prototyping. In: Farwer, B., Moldt, D. (eds.) *Object Petri Nets, Processes, and Object Calculi*, University of Hamburg (2005), Technical Report FBI-HH-B-265/05
- [104] Stehr, M.-O., Talcott, C.: Planning and learning algorithms for routing in disruption-tolerant networks. In: *MILCOM 2008*. IEEE (2008)
- [105] Talcott, C., Eker, S., Knapp, M., Lincoln, P., Laderoute, K.: Pathway logic modeling of protein functional domains in signal transduction. In: *Proceedings of the Pacific Symposium on Biocomputing* (January 2004)
- [106] Talcott, C.L.: The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation. PhD thesis, Stanford University (1985)
- [107] Talcott, C.L.: Rum: An intensional theory of function and control abstractions. In: *Foundations of Logic and Functional Programming. LNCS*, vol. 306, pp. 1–44. Springer, Heidelberg (1986)
- [108] Talcott, C.L.: Algebraic methods in programming language theory. In: *First International Conference on Algebraic Methodology and Software Technology*, Iowa City, Iowa, AMAST 1989 (1989)
- [109] Talcott, C.L.: Programming and proving with function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University Computer Science Department (1989)
- [110] Talcott, C.L.: Binding structures. In: Lifschitz, V. (ed.) *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press (1991)
- [111] Talcott, C.L.: Towards a framework for specifying components of automated reasoning systems: A report on work in progress. In: *TTCP XTP-1 Workshop on Effective Use of Automated Reasoning Technology in System Development*, EUARTSD (1992)
- [112] Talcott, C.L.: Sketch of an architecture for reasoning systems (1993)
- [113] Talcott, C.L.: A theory for program and data specification. *Theoretical Computer Science* 104, 129–159 (1993)
- [114] Talcott, C.L.: A theory of binding structures and its applications to rewriting. *Theoretical Computer Science* 112, 99–143 (1993)
- [115] Talcott, C.L.: Mathematical foundations for survivable systems. In: *Proceedings of IMACS 1994 Workshop on New Mathematics for Computer Science* (1994)
- [116] Talcott, C.L.: Reasoning specialists should be logical services, not black boxes. In: *Proceedings of CADE-12 workshop on Theory Reasoning in Automated Deduction*, pp. 1–6 (1994)

- [117] Talcott, C.L.: Reasoning about programs. Notes from Invited Talk for the Dagstuhl Workshop on New Trends in Integration of Paradigms and Coordination. Dagstuhl, Germany (September 1995)
- [118] Talcott, C.L.: An actor rewriting theory. In: Meseguer, J. (ed.) Proc. 1st Intl. Workshop on Rewriting Logic and Its Applications. Electronic Notes in Theoretical Computer Science, vol. 4, pp. 360–383. Elsevier (1996), <http://www.elsevier.nl/locate/entcs/volume4.html>
- [119] Talcott, C.L.: Interaction semantics for components of distributed systems. In: Najm, E., Stefani, J.-B. (eds.) 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS 1996 (1996); Proceedings published in 1997 by Chapman & Hall
- [120] Talcott, C.L.: Reasoning about functions with effects. In: Higher Order Operational Techniques in Semantics. Cambridge University Press (1996)
- [121] Talcott, C.L.: Reflection in actor systems. Paper presented at the Workshop on New Mathematics for Computer Science – Computational Models and Semantics Session (October 1996)
- [122] Talcott, C.L.: Composable semantic models for actor theories. In: Abadi, M., Ito, T. (eds.) TACS 1997. LNCS, vol. 1281, pp. 321–364. Springer, Heidelberg (1997)
- [123] Talcott, C.L.: Interaction Semantics for Components of Distributed Systems. In: Najm, E., Stefani, J.-B. (eds.) Formal Methods for Open Object-based Distributed Systems, pp. 154–169. Chapman & Hall (1997)
- [124] Talcott, C.L.: Composable semantic models for actor theories. Higher-Order and Symbolic Computation 11(3), 281–343 (1998)
- [125] Talcott, C.L.: Reasoning about programs with effects. In: 2nd NSF-CNPq Workshop on Semantics. Electronic Notes in Theoretical Computer Science. Elsevier (1998)
- [126] Talcott, C.L.: Towards a toolkit for actor system specification. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 391–406. Springer, Heidelberg (2000)
- [127] Talcott, C.L.: Actor theories in rewriting logic. Theoretical Computer Science 285(2) (2002)
- [128] Talcott, C.L., Weyhrauch, R.W.: Partial evaluation, higher-order abstractions, and reflection principles as system building tools. In: Bjorner, D., Erschov, A.P. (eds.) IFIP TC2 Working Conference on Partial and Mixed Computation, Ebberup, Denmark. North-Holland (1987)
- [129] Talcott, C.L., Weyhrauch, R.W.: Towards a theory of mechanized reasoning I: FOL contexts, an extensional view. In: Proc. of the 8th European Conference on Artificial Intelligence (ECAI 1990), pp. 634–639 (1990)
- [130] Talcott, C.: Electron Spin Resonance Studies of Radicals Produced by Electrolysis. PhD thesis, University of California, Berkeley (1967)
- [131] Talcott, C.: Coordination models based on a formal model of distributed object reflection. In: 1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord 2005 (2005)
- [132] Talcott, C.: Formal executable models of cell signaling primitives. In: Margaria, T., Philippou, A., Steffen, B. (eds.) 2nd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation ISOLA 2006, pp. 303–307 (2006)
- [133] Talcott, C.: Policy-based coordination in pagoda: A case study. In: 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2006). ENTCS, vol. 181(7) (2006)

- [134] Talcott, C.: Symbolic modeling of signal transduction in pathway logic. In: Perone, L.F., Wieland, F.P., Liu, J., Lawson, B.G., Nicol, D.M., Fujimoto, R.M. (eds.) 2006 Winter Simulation Conference, pp. 1656–1665 (2006)
- [135] Talcott, C.: A formal framework for interactive agents. In: Arbab, F., Golden, D. (eds.) Foundations of Interactive Computation (FInCo 2007). Electronic Notes in Theoretical Computer Science, vol. 203, pp. 95–106. Elsevier (2007)
- [136] Talcott, C.: Pathway logic. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 21–53. Springer, Heidelberg (2008)
- [137] Talcott, C., Dill, D.L.: The pathway logic assistant. In: Plotkin, G. (ed.) Third International Workshop on Computational Methods in Systems Biology, pp. 228–239 (2005)
- [138] Talcott, C., Dill, D.L.: Multiple representations of biological processes. In: Priami, C., Plotkin, G. (eds.) Transactions on Computational Systems Biology VI. LNCS (LNBI), vol. 4220, pp. 221–245. Springer, Heidelberg (2006)
- [139] Talcott, C., Lincoln, P.: Towards a semantic framework for secure agents: Extended abstract. In: High Confidence Software and Systems, HCSS 2003 (April 2003)
- [140] Talcott, C., Sirjani, M., Ren, S.: Comparing three coordination models: Reo, arc, and rrd. In: Formal Methods for Open Object-based Distributed Systems. Springer, Heidelberg (2007)
- [141] Talcott, C., Sirjani, M., Ren, S.: Comparing three coordination models: Reo, arc, and rrd. Science of Computer Programming (2009)
- [142] Tiwari, A., Talcott, C.: Analyzing a discrete model of alypsia central pattern generator. In: Heiner, M., Uhrmacher, A. (eds.) CMSB 2008. LNCS (LNBI), vol. 5307, pp. 347–366. Springer, Heidelberg (2008)
- [143] Tiwari, A., Talcott, C., Knapp, M., Lincoln, P., Laderoute, K.: Analyzing pathways using SAT-based approaches. In: Anai, H., Horimoto, K., Kutsia, T. (eds.) Ab 2007. LNCS, vol. 4545, pp. 155–169. Springer, Heidelberg (2007)
- [144] Venkatasubramanian, N., Agha, G., Talcott, C.L.: Scalable distributed garbage collection for systems of active objects. In: Bekkers, Y., Cohen, J. (eds.) IWMM-GIAE 1992. LNCS, vol. 637, pp. 134–147. Springer, Heidelberg (1992)
- [145] Venkatasubramanian, N., Agha, G., Talcott, C.L.: Composable QoS-based distributed resource management. Position paper for Workshop on Compositional Software Architectures, Monterey, CA (January 1998)
- [146] Venkatasubramanian, N., Agha, G., Talcott, C.L.: A metaobject framework for qos-based distributed resource management. In: Third International Symposium on Computing in Object-Oriented Parallel Environments, ISCOPE 1999 (1999)
- [147] Venkatasubramanian, N., Agha, G., Talcott, C.: A formal model for reasoning about adaptive QoS-enabled middleware. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 197–221. Springer, Heidelberg (2001)
- [148] Venkatasubramanian, N., Agha, G., Talcott, C.L.: Formal reasoning for QoS-enabled middleware. ACM Transactions on Software Engineering and Methodology (2004) (accepted for publication)
- [149] Venkatasubramanian, N., Talcott, C.L.: A metaarchitecture for distributed resource management. In: Hawaii International Conference on System Sciences, HICSS-26 (January 1993)
- [150] Venkatasubramanian, N., Talcott, C.L.: Reasoning about meta level activities in open distributed systems. In: Principles of Distributed Computation (PODC 1995), pp. 144–153. ACM (1995)

- [151] Venkatasubramanian, N., Talcott, C.L.: A reflective framework for providing safe qos-enabled customizable middleware. In: Workshop on Reflective Middleware, RM 2000 (2000)
- [152] Venkatasubramanian, N., Talcott, C.L.: A semantic framework for modeling and reasoning about reflective middleware (2001)
- [153] Wang, A., Talcott, C., Jia, L., Loo, B.T., Scedrov, A.: Analyzing BGP instances in maude. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 334–348. Springer, Heidelberg (2011)
- [154] Weyhrauch, R.W., Cadoli, M., Talcott, C.L.: Using abstract resources to control reasoning. *Journal of Logic Language and Information* 7, 77–101 (1998)
- [155] Weyhrauch, R.W., Talcott, C.L.: The logic of FOL systems: Formulated in set theory. In: Hagiya, M., Jones, N.D., Sato, M. (eds.) *Logic, Language and Computation*. LNCS, vol. 792, pp. 119–132. Springer, Heidelberg (1994)
- [156] Wilkins, D., Denker, G., Stehr, M.-O., Elenius, D., Senanayake, R., Talcott, C.: Coral - policy language and reasoning techniques for spectrum policies. In: *Policy 2007* (2007)
- [157] Wilkins, D., Denker, G., Stehr, M.-O., Elenius, D., Senanayake, R., Talcott, C.: Policy-based cognitive radios. *IEEE Wireless Communications* (2007); Special Issue on Cognitive Wireless Networks (to appear)
- [158] Wirsing, M., Denker, G., Talcott, C., Poggio, A., Briesemeister, L.: A rewriting logic framework for soft constraints. In: *Sixth International Workshop on Rewriting Logic and Its Applications (WRLA 2006)*. *Electronic Notes in Theoretical Computer Science*. Elsevier (2006)
- [159] Yu, Y., Ren, S., Talcott, C.: Coordinating asynchronous and open distributed systems under semiring-based timing constraints. In: Canal, C., Poizat, P., Sirjani, M. (eds.) *Foundations of Coordination Languages and Software Architectures FOCLASA 2008* (2008)

Two PhD Students for the Price of One

Solomon Feferman

Stanford University, Stanford CA 94305, USA

`feferman@stanford.edu`

Carolyn Talcott received her PhD in Computer Science at Stanford under my nominal direction in 1985; thereby hangs a tale. A year later, largely under her tutelage and again under my nominal direction, Ian Mason received *his* PhD in the Special Program in Logic, Philosophy of Language and the Philosophy of Science at Stanford; thereby hangs a subsidiary tale.

Carolyn first attended my graduate courses at Stanford on mathematical logic and the foundations of mathematics in the spring quarter of 1978. One of those was on set theory; more significantly for our future connections, the other course was on my “Explicit Mathematics” approach to the formalization of the modern development of constructive mathematics due to Errett Bishop. Though she apparently had no substantial background in logic I took special note of her quick absorption of the material presented in those classes. Carolyn continued to follow my graduate courses through the next academic year 1978–79, concluding in the spring of 1979 with one on constructive and effective algebra. Among her contributions to that was an excellent presentation of a paper on Bishop-style constructive algebra, plus a valuable resource bibliography of computational algebra. Even more than before, I was very impressed at that point with her ready understanding and command of research level ideas and results. I was on leave for the academic year 1979–80, so our contact didn’t resume until after my return in the fall of 1980.

As I got to know Carolyn better, I learned that her general background was quite advanced and diverse, that she had already received a PhD in Chemistry at UC Berkeley in the mid 1960s and that she had much preferred the programming aspects of her research to lab work with “smelly” chemicals. During a post-doc year at Cambridge University, she expanded her programming experience with work on mathematical questions on finite groups that could be dealt with computationally. Carolyn told me that for personal reasons she had then spent the years 1969–1975 teaching mathematics as a TA and RA at UC Santa Cruz. Seeing this as a dead-end, she decided to enter graduate work in computer science at Stanford in 1977, during which her main research interests increasingly concentrated on the formal foundations of LISP style programming languages under the direction of John McCarthy. Though I had (and still have) only a superficial knowledge of LISP and knew nothing about its ins and outs in practice, in 1981 Carolyn asked me whether I would be willing to be her PhD supervisor. Of course it would not have been appropriate to have a spouse in that capacity, but (as she tells me) she was also influenced by my way of doing things in the formulation of proof systems for the foundations of constructive mathematics and in particular with the formal handling of its intensional aspects. In

any case, I readily agreed because it was very clear by then that Carolyn was already working at a high level and had definite plans as to the kind of thing she aimed to accomplish; I knew I could rely on her to carry those through in a professional way. Though I saw my role as largely nominal, it was actually not entirely so, since we met with some regularity during the following years in order that I might be apprised of her ongoing research and for me to offer occasional input as to content and exposition. As I had expected, the subject matter of her PhD thesis and its detailed development was entirely her own; it concerned an expansion of the Lisp language and its semantics to a new language baptized Rum (in honor of her favorite ice cream, Rum Raisin), devised to cover functional and control abstractions, program transformations, closures, and continuation structures. Entitled *The Essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation* (Talcott 1985), her thesis was a substantial achievement that marked the beginning of a new career.

In the meantime, I had agreed to supervise the doctoral work of Ian Mason, who had arrived from Australia as a graduate student in the Philosophy Department in 1981. Ian's strong interests in logic were clear from the start and, in his eagerness to move along, he soon took advantage of the existence of the Special Program in Logic within Philosophy to concentrate his studies in that direction. Besides taking my courses, in which he was regularly in the group of top students, I began meeting with him regularly as his advisor by the spring of 1982. In the following year or two I suggested several possible dissertation topics, mostly in the area of abstract model theory. However, none of these really grabbed him and only one thing came out of those explorations, the quite nice paper Mason (1985) on the undecidability of a formalized metatheory of the propositional calculus. But during the same period, Ian was talking more and more to Carolyn about her work, and that did engage him so thoroughly that he decided, with my approval, to change his thesis direction to follow the route prepared by her. That eventuated in his fine dissertation, *The Semantics of Destructive Lisp* (1986), in which, among other things, he accounted at a theoretical level for the operation of Lisp-type programs in the application of which memory is subject to mutation.

Thus it was that I gained two PhD students for the price of one. And, as it turned out, this was the beginning of a joint research program by Mason and Talcott, a very active and productive collaboration that has continued until this day.¹ Of personal interest to me was their paper, *Feferman-Landin Logic* (Mason and Talcott 2002), contributed to the "Feferfest" held in my honor at Stanford in 1998. In that paper, a quite general logic for Landin-type imperative functional programming languages is presented that incorporates features of my theories of explicit mathematics for operations and classes (aka variable types). Though I have not otherwise been able to follow Carolyn's and Ian's subsequent

¹ Following his PhD, Mason returned to Australia where he held academic positions for a number of years. He was able eventually to obtain a research position in Palo Alto and, by means of that and subsequent such positions, he and Talcott have been able to continue their collaboration at close hand.

research, I have maintained warm personal relations with both of them over the years since they completed their respective doctoral theses. In particular, in the years that Carolyn stayed on at Stanford as a Research Associate and then moved over to the Stanford Research Institute (SRI), she was helpful to me in a variety of ways and I always appreciated her unfailing good humor and quiet effectiveness. Among other things, she was one of the organizers—along with two other of my former students, Jon Barwise and Wilfried Sieg, together with my close colleague Richard Sommer—of the 1998 Feferfest. That owed its success in no small measure to her extensive work on that event.

Now, suddenly, here we are at a Festschrift for Carolyn Talcott herself. Cliché or not—how the time does fly! But given her vitality and productivity, I hope she'll see that as just another way-station at which to mark the continued progress of her fine career. And for me it is an occasion to be grateful once more for my rare good fortune in having had her as a student who could stand so readily and so assuredly on her own, and to then having had her significant help in launching Ian Mason's career at just the right time in just the right place.

References

- Feferman, S.: A language and axioms for explicit mathematics. In: Crossley, J.N. (ed.) *Algebra and Logic*. Lecture Notes in Mathematics, vol. 450, pp. 87–139. Springer, Berlin (1975)
- Mason, I.A.: The metatheory of the classical propositional calculus is not axiomatizable. *J. Symbolic Logic* 50, 451–457 (1985)
- Mason, I.A.: *The Semantics of Destructive Lisp*. PhD Dissertation, Stanford University (1986)
- Mason, I.A., Talcott, C.L.: Feferman-Landin logic. In: Sieg, W., Sommer, R., Talcott, C. (eds.) *Reflections on the Foundations of Mathematics*, Association for Symbolic Logic, Urbana, IL. *Lecture Notes in Logic*, vol. 15, pp. 293–328. A K Peters, Ltd., Natick (2002)
- Talcott, C.L.: *The Essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation*. PhD Dissertation, Stanford University (1985)

Honoring Carolyn Talcott's Contributions to Science

Sylvan Pinsky

SRI International, Menlo Park, CA 94025, USA

Abstract. This paper describes both Carolyn Talcott's technical and leadership contributions to formal methods, cryptographic protocol analysis, and systems biology. Carolyn has played a vitally important leadership role in protocol analysis through her significant research and bringing together leading members of the protocol analysis community. Her efforts have resulted in a unified, cohesive, and flexible foundation for the interoperation of maturing tools and techniques for designing and evaluating a wide range of protocols. As the leader of the Symbolic Systems Technology Group at SRI she has been a visionary manager with exceptionally strong technical skills who has guided, advised and mentored numerous scientists in the use of formal methods and other computational tools for modeling or solving diverse biological problems in cancer biology, signal transduction research, neuroscience, and infectious disease research.

1 Introduction

I have extensive experience in formal methods and computer security at the National Security Agency (NSA) starting in 1984. For almost half of that time, I have had the pleasure of being a sponsor and colleague of Carolyn's work in formal methods, rewrite logic, and protocol analysis. This is an area where formal methods are used to identify and fix weaknesses in existing communications protocols and design new protocols that meet security requirements. I hosted quarterly meetings to bring together the leading researchers and developers of protocol analysis tools. Typical representation included the Naval Research Laboratory (NRL), NSA, MITRE, SRI, Kestrel Institute, Stanford University, University of Pennsylvania, University of Illinois at Urbana-Champaign, Naval Postgraduate School, Carnegie Mellon University, and other organizations. Carolyn played a key role in integrating the capabilities of the major tools by developing a common framework and language to describe the inputs, algorithms, and outputs for each tool. This effort resulted in a major advance in the interoperability of protocol analysis tools. It reflects Carolyn's impressive leadership capabilities, scientific knowledge, and ability to creatively solve challenging problems, and communicate approaches to other researchers. Her success was based on the Maude system, and since several participants were unfamiliar with this tool and environment, she gave several presentations on this subject. She educated others, answered difficult and insightful questions with ease, and encouraged active interaction among all participants.

I learned about systems biology from Pat Lincoln, the Director of the Computer Science Laboratory (CSL) at SRI and many conversations with Carolyn regarding her work in pathway logic, an approach to modeling biological systems and processes based on rewriting logic. I retired from NSA and joined Carolyn's symbolic systems technology group at SRI in 2008. She has an excellent ability to explain concepts on many levels of complexity in a very positive, intuitive and nurturing manner. A major reason for coming to SRI was to have the opportunity to work with Carolyn. As a supervisor and mentor, she has demonstrated exceptionally strong technical leadership and the ability to inspire, encourage, and support others. For substantive issues in understanding the complexities of biological systems, she provides clear explanations, appropriate references or recommends individuals to see for more detailed conversations. Although I have been at SRI for a relatively short time, I view Carolyn as having been my mentor in systems biology for several years.

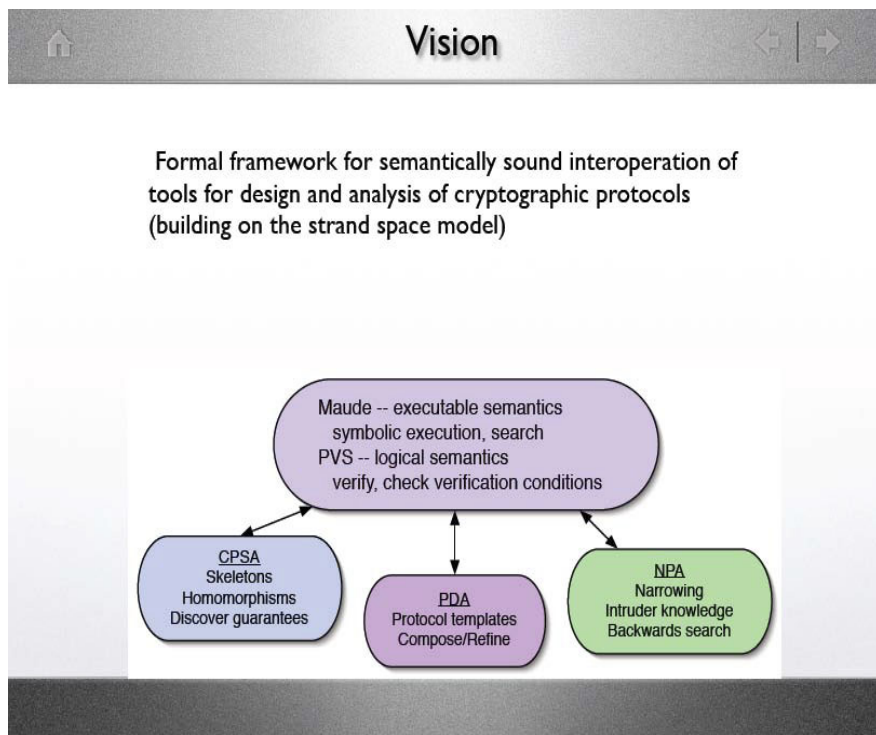
2 Formal Methods and Protocol Analysis

The design and analysis of cryptographic protocols has a long history of activity. Protocols have subtle errors unless special techniques are used for their discovery. The Needham-Schroeder protocol [1] is a three line protocol developed in 1978 that took about seventeen years before a flaw was discovered. An important element in protocol analysis is a model of the intruder. The Dolev-Yao model [2] quickly became the standard for modeling the capabilities of the intruder and played a significant role in the early automated tools for protocol analysis; specifically Jonathan Millen's Interrogator [3] and Cathy Meadows's NRL Protocol Analyzer [4]. These tools and studies by Dick Kemmerer and Cathy Meadows ([5], [6]) and Martín Abadi and Roger Needham [7] generated interest in applying formal methods to protocol analysis. Significant research in applying formal methods to this domain specific area continue to be presented at the Symposium on Security and Privacy and the Computer Security Foundation Workshop. Carolyn Talcott and her colleagues Grit Denker, José Megeguer, Peter Ölvezky, and others ([8], [9], [10], [11]) have contributed to the literature by applying Maude ([12], [13]) to the specification and analysis of active networks.

The National Computer Security Center and the Research and Evaluation Groups at NSA have sponsored research in formal methods and applications of protocol analysis. They have supported the Interrogator model, the strand space theory developed at MITRE by Joshua Guttman, Javier Thayer Fábrega, and Jonathon Herzog ([14], [15], [16]), and the enhancement of protocol analysis tools through PVS and Maude techniques. Meetings with MITRE researchers and Cathy Meadows, Paul Syverson, and Iliano Cervesato of NRL were organized by NSA and later expanded to include Carolyn Talcott, Mark-Oliver Stehr, José Meseguer, Andre Scedrov, John Mitchell, and other researchers active in

protocol analysis. These informal gatherings became regular quarterly meetings with the group expanding to include other active researchers in the field. Iliano Cersesato initiated the first Protocol eXchange website in 2003 which is now being hosted by George Dinolt at the winter meetings held at the Naval Postgraduate School (NPS). The use of model checking ([17], [18]), the maturing of strand space theory and multiset rewriting for protocol analysis developed by Iliano Cersato, Nancy Durgin, Pat Lincoln, John Mitchell, and Andre Scedrov [19] and the introduction of the Protocol Derivation Assistant (PDA) by Dusko Pavlovic [20] prompted a unified approach to tool development and methodology. I asked Carolyn to take the lead in this effort by using the best features of PVS [21] and Maude [13] as an underlying framework for protocol analysis. The idea was to use formal mappings between PVS and Maude in order to provide the best of both worlds for a wide range of formal modeling and analysis problems. She presented an excellent approach to a Maude-PVS tool for strand spaces [22] starting with an informal strand space description and using tools to input the specification and provide visualization of bundles based on strand space structures defined in Maude. Carolyn and Sam Owre [23] subsequently defined strand spaces, penetrators, and proved most of the Needham-Schroeder-Lowe protocol in PVS by taking the MITRE work on skeletons and homomorphisms and proving TCCs and associated lemmas. They also used PVS and Maude to represent the terms, actions, events, and processes and semantics for ground and symbolic execution of Dusko's PDA approach. Carolyn demonstrated impressive leadership qualities in providing the framework for developing cryptographic protocol analysis algorithms. The framework was designed to be applicable to the MITRE Cryptographic Protocol Shape Analyzer (CPSA), the Kestrel Protocol Derivation Assistant, and the NRL Protocol Analyzer written in Maude (NPA-Maude). The initial idea was to specify and prototype in Maude and verify in PVS. Formal representations of strand spaces and strand space protocols would be developed in Maude and PVS with mappings between representations. Carolyn then expanded the vision for the framework to include semantically sound interoperation of tools for the design and analysis of cryptographic protocols. The following schematic from [24] depicts her vision of the role of PVS and Maude for the interoperability and coordination between the tool developers at MITRE, Kestrel, and NRL.

At the Protocol eXchange meeting at NPS in 2007, Sam Owre [25] explained the PVS-Maude connection by describing the PVS protocol analysis specification and showing that it is a theory interpretation of the Maude specification where the axioms are mapped to proof obligations. Providing these proofs guarantee that the interpretation is sound. He also talked about the translation of the strand space specification developed by Carolyn. At this meeting and later ones ([26], [27]), Carolyn described the semantics and algebra for the interoperation of protocol analysis tools and the simulation and analysis of protocol specifications.



Carolyn has played a vitally important leadership role in protocol analysis through her significant research and bringing together leading members of the protocol analysis community. Her efforts have resulted in a unified, cohesive, and flexible foundation for the interoperation of maturing tools and techniques for designing and evaluating a wide range of protocols.

3 Carolyn's Leadership at SRI

Carolyn is the leader of the Symbolic Systems Technology Group of the Computer Science Laboratory at SRI. When I joined her group in 2008, it immediately became clear that the entire team had great respect for her as an outstanding manager, group leader, and mentor. She has continually been an excellent and visionary researcher with exceptionally strong technical skills who is a warm, caring, energetic, positive and supportive person who will always go the extra mile, not only for her group members but colleagues working in her field or interested in learning more about her field. Her devotion to deliver the best work possible has inspired and motivated the group to operate at its maximum effort. Carolyn is a real team player who always contributes to the solution in a highly constructive and strongly encouraging manner. She is a role model for integrity, trust, openness, and respect for others in a professional working environment. Carolyn has been very generous in her professional interactions

with scientific collaborators. She has demonstrated exceptional dedication to helping her staff and colleagues succeed, mentoring younger staff and partnering with more senior staff members. Her upbeat can-do attitude is infectious, and she has led her team to a series of successes in winning business, inventing, achieving new scientific understanding, and developing well-engineered software. She has guided, advised, or mentored numerous scientists in the use of formal methods and other computational tools for modeling or solving diverse biological problems (e.g., in cancer biology signal transduction research, neuroscience, and infectious disease research). We have always been impressed with Carolyn's patience as she works with biologists, particularly those with limited computer skills, enabling them to glimpse some of the power of modern approaches to computational modeling. She was instrumental in hosting weekly meetings with other SRI organizations such as the Artificial Intelligence Center and the Bio-Sciences Division to find areas of interest for mutual collaboration and they have expressed high praise for Carolyn's teaching skills and cheerful willingness to help. She provides opportunities and support to young scientists and shows fairness and equity in resolving issues. As a mentor, she has successfully encouraged professional development for many researchers. The best part about her is her ability to maintain open lines of communication with her staff and colleagues which has enhanced SRI's long-term success

The great thing about Carolyn is that she is always available if you need help or advice (and she always has a solution), but she also enables and encourages members of her group to work entirely independently. It is very clear that Carolyn's deep professional interest is in research, and she understands that while doing an excellent job in managing people and resources. Carolyn is a great role model for a successful researcher and manager. Different from most, she does not get caught in a particular mindset nor does she constrain herself to one narrow research topic, but she is a tremendously open-minded, interdisciplinary person and can easily grasp the essence of new research problems, which makes it simply great fun to work and learn with her. Her devotion to hard and productive work is also a great inspiration and goes beyond almost anybody that we know.

Carolyn has very deep knowledge in a multitude of fields including logic, chemistry, biology, and at the same time is humble to admit where she thinks other people know more, which encourages and hugely impacts team play. If SRI's strength is an interdisciplinary approach then Carolyn is synonymous with SRI. Multiple organizations at SRI have said that research wise, her colleagues are highly inspired by her view of systems biology as an application of logic, and it is not an exaggeration to say that we have learned from her and her systems biology group an entirely new interpretation of computer science.

4 Symbolic Systems Biology

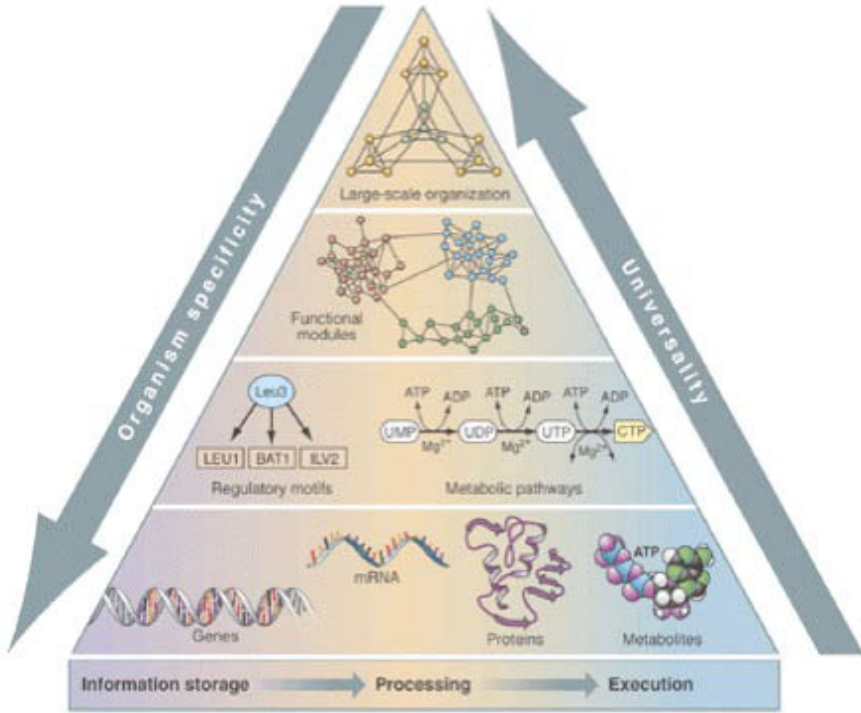
Computational modeling techniques are providing scientists with the tools to address complex biological networks and their interacting sub-networks. The field of systems biology offers capabilities for developing a systems-level knowledge and understanding of the interaction and complexity of gene regulatory

networks and metabolic and signaling pathways that control cellular behaviors and interactions. New approaches in systems biology and computational biology that include reducing complexity by using abstraction and simulation techniques to analyze quotient spaces are the forerunners of an enabling technology that will eventually transform traditional experimental (reductionist) biology into a predictive science. A central element in accelerating progress is the use of an interdisciplinary approach to coupling classical mathematics with formal methods techniques to develop new mathematical concepts that enable modeling and prediction of behavior in large-scale networks that evolve over time, such as those occurring in the biological and communication sciences. Symbolic systems biology, which is the qualitative and quantitative study of biological process as an integrated system, will play a key role in this process. It is intended to model networks of biological processes in a logical framework with the capability to compute and analyze networks. Diverse models from finite automata theory, stochastic models, differential equations, dynamical systems, and control theory can be integrated using symbolic and qualitative reasoning.

Carolyn was the inspiration and driving force behind pathway logic [28], an approach to modeling biological systems and processes based on rewriting logic. Pathway logic uses rewrite theories to formalize the informal models that biologists commonly use to explain biological processes. As a computational science ([29], [30]), it provides researchers with powerful tools to facilitate the understanding of complex biological systems and accelerate the design of experiments to test hypotheses about their functions *in vivo*.

An important goal of systems biology is to build diverse models that are consistent with each other and with the biology, to integrate these models, and to analyze the aggregated information (structures organized at multiple levels of complexity and diversity including molecular, cellular, and organism levels).

The need for combining formal methods techniques with classical mathematics is emphasized when considering differential equations or difference equations which naturally arise in modeling biological systems ([31], [32]). Analytical and numerical studies of such models frequently present interesting and challenging dynamical system questions. Modeling neurons is an excellent example. Neurons are highly specialized cells capable of communicating with each other by means of electrical and chemical signaling. Eugene Izhikevich [33] addressed the computational inefficiency of some of the most useful models of spiking and bursting neurons. The large scale simulation of cortical neural networks quickly becomes computationally infeasible due to the nonlinearity of the neuron equations. Ashish Tiwari and Carolyn Talcott [34] bypassed this roadblock by proving properties of a discrete abstraction of this system using model checking techniques. Many biological systems are described by nonlinear differential equations which require numerical solutions. For example, the spread of infectious diseases is described by the standard Susceptible-Infected-Recovered (SIR) model ([31], [32]) for the three groups of individuals using the notation:



$S(t)$ = number of susceptible individuals in the population at time t
 $I(t)$ = number of infected individuals in the population at time t
 $R(t)$ = number of recovered individuals in the population at time t
 N = population size.

The system of differential equations is given by

$$\begin{aligned}
 \frac{dS}{dt} &= -\beta SI \\
 \frac{dI}{dt} &= \beta SI - \nu I \\
 \frac{dR}{dt} &= \nu I.
 \end{aligned}
 \tag{1}$$

We note that $S(t) + I(t) + R(t)$ is the constant N , the total population; consequently, we need to solve only the first two equations for S and I as a function of time since $R(t) = N - S(t) - I(t)$. We do not have to rely on numerical solutions to obtain the relationship between infected and susceptible individuals. The derivative of S with respect to I is obtained by dividing $\frac{dS}{dt}$ by $\frac{dI}{dt}$ from equation 1. If $I \neq 0$, this reduces to

$$\frac{dS}{dI} = -\frac{\beta S}{\beta S - \nu}$$

and separating variables results in

$$\left(-1 + \frac{\nu}{\beta} \frac{1}{S}\right) dS = dI$$

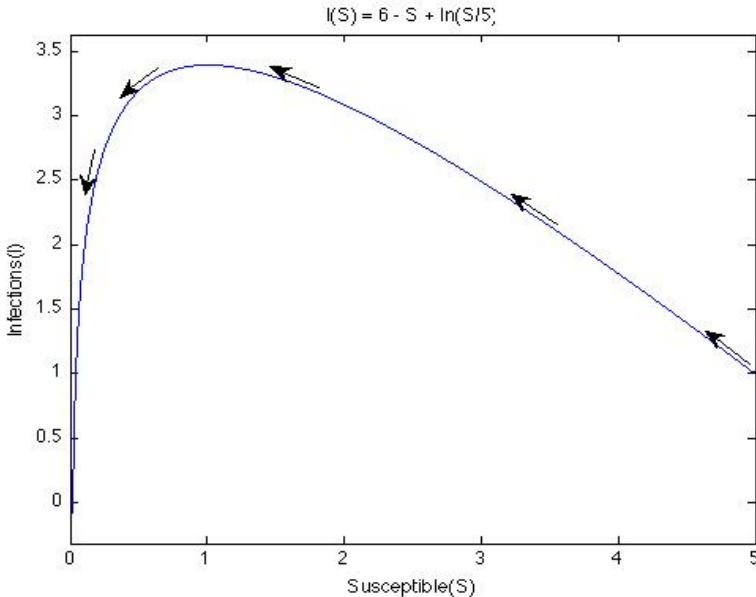
producing the solution $\left(\frac{\nu}{\beta} \ln(S) - S\right) \Big|_0^t = I(u) \Big|_0^t$ that simplifies to

$$I(t) = I_0 + (S_0 - S(t)) + \frac{\nu}{\beta} \ln\left(\frac{S(t)}{S_0}\right).$$

The phase portrait $\{(S(t), I(t)) \mid t \geq 0\}$ is constrained to lie on the curve

$$I(S) = C - S + \frac{\nu}{\beta} \ln(S) \tag{2}$$

where $C = I_0 + S_0 - \frac{\nu}{\beta} \ln(S_0)$. This abstraction removes time and provides an over-approximation for solutions to equation 1. Since $S(t)$ is a decreasing function, the solution starts at S_0 and moves to the left as time increases. The following figure demonstrates this behavior for $S_0 = 5, I_0 = 1, \nu = 1$, and $\beta = 1$.



The difficult task is to assemble a synergistic team of mathematicians, biologists, and computer scientists who will collaborate to formulate mathematical structures to model biologically relevant networks, modules and functional subnets, their interactions and interdependencies. Carolyn is one of those rare individuals who has the ability, insight, energy, and desire to find opportunities to develop interdisciplinary approaches to address key challenges in biology. The

capability to extract and organize crucial information, and discover hidden structures in biological networks will provide insights that could accelerate research in biology with applications to medicine.

5 New Insights for Network Science

In February 2010, Andy Poggio’s idea to experiment with discrete biological systems to gain insights into network science was funded by the Office of Naval Research (ONR) [35]. The research was designed to model various aspects of biology at several levels of organization (genome, cell, organism, and colony) to understand how networks form, are controlled, and adapt to changes, and finally determine the ways in which those insights apply to computer networks. Andy argued that network science examines interconnections and interactions among different networks, and as such, is an interdisciplinary field in search of its foundations as a new discipline that focuses more on integration rather than reduction, on emergent properties rather than continuity of the same. This effort naturally fell under Carolyn’s direction where systems biology and systems chemistry would drive new understanding of computer networks, cyberspace, and cyber-physical space. Fred Vigneault, a microbiologist from the Shenandoah Valley facility of SRI, collected weekly H1N1 virus data reported at the national, regional, and state levels by the CDC. His objective was to analyze the 2009 H1N1 Pandemic in the United States. My participation in the project was to model the overall spread of infectious diseases for a network consisting of states, territories, and regions, and determine how to measure each element of the network and their interaction, aggregation, and influence on the total system behavior.

Carolyn served in her typical role of providing leadership and inspiration when issues emerged that needed resolution. Fred had considerable CDC data that represented the number of infected individuals by state and region. The issue was how to make sense of this “heatmap” data (see the following figure) and determine the influence of airline travel on the spread of disease across the regions. Carolyn, Fred, and I had regular discussions to address various approaches to understanding, modeling, and analyzing the data. Carolyn was tremendously effective in identifying stumbling blocks, carefully listening to our ideas, encouraging alternative approaches, providing insightful comments and suggestions, and keeping us focused.

Legend for spread and severity of influenza infection in the United States:

0 = No Activity (no laboratory-confirmed cases of influenza and no reported increase in the number of cases of ILI),

1 = Sporadic (small numbers of laboratory-confirmed influenza cases or a single laboratory-confirmed influenza outbreak has been reported, but there is no increase in cases of ILI),

2 = Local (outbreaks of influenza or increases in ILI cases and recent laboratory-confirmed influenza in a single region of the state),

3 = Regional (outbreaks of influenza or increases in ILI and recent laboratory

largest airports, which represent 99% of this data, to compare the behavior of three networks:

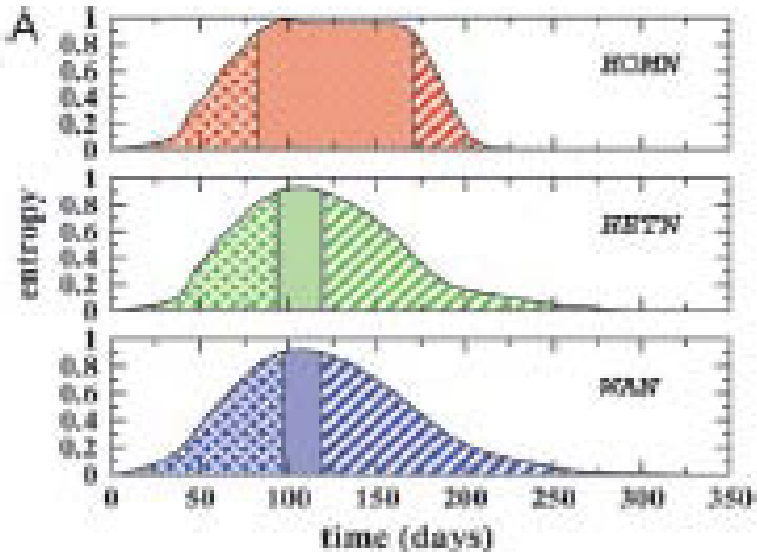
WAN : worldwide airline transportation network

HOMN : homogeneous Erdos-Rényi random graph

HETN : random graph with the same topology as *WAN*

The Erdos-Rényi model has the same number of vertices as the *WAN* with edges (i, j) drawn from a uniform distribution using the average degree from the *WAN*; whereas the *HETN* model uses the actual degree determined from the *WAN* for each edge. Each city has a SIR model and the fraction of infections $i_j(t) = I_j(t)/N_j$ is normalized using $\rho_j = i_j/\sum_k i_k$ to obtain the total entropy-like function:

$$H(t) = -\frac{1}{\ln(N)} \sum_j \rho_j(t) \ln(\rho_j(t)).$$



Although $H(t)$ has a shape similar to the entropy of each node of the *WAN* network, the random graphs have longer time periods where the entropy is near its maximum value. The shaded region in the above figure, reproduced from Colizza [36], corresponds to $H > 0.9$.

Once we developed a mathematical model that reflected the data, Carolyn was especially helpful in identifying implications for communications networks that would be of interest to ONR. The modeling approach can be applied to analyzing and predicting the behavior in large-scale communications networks that evolve over time. We also recommended that open systems should be studied for application to networks. Open systems continually interact with their environment through exchanges of energy, materials, and information to reach a

steady state and can evolve toward states of greater complexity and differentiation. In contrast to closed systems where the laws of thermodynamics constrain entropy to increase, the continual interchange of materials for open systems provides a richer framework for identifying entropy-like functions for designing and analyzing dynamic networks. The next section describes our effort to encourage interdisciplinary research in open systems.

6 Open Systems

Carolyn and I had several interactions with Harvey Rubin at the University of Pennsylvania (Penn) to determine the best forum to bring experts from diverse fields together to define the challenging problems in the biological sciences and identify how systems biology could provide insights into their solutions. Motivated by her work in open distributed systems, she suggested connections to biological systems and recommended that SRI and Penn jointly organize a workshop in open systems. Helen Gill of the National Science Foundation and Brad Martin of NSA sponsored the workshop we organized and hosted at Penn in May 2010.

Open systems maintain themselves in an ongoing build up and break down of components and are capable of self-regulation and adapting to circumstances by changing the structure and process of their internal components. Examples of open systems include both natural and man-made. Biological systems such as immune systems and microbial systems are open, non-equilibrium systems that are embedded in a changing and often hostile environment exchanging energy and molecular species with its surroundings. The web services, social networking, and the Internet itself are examples of man-made open systems. Cyber physical systems that involve close interaction between computational and physical systems include a wide variety of open systems, including: medical devices, robotics swarms, sensor/actor systems, smart buildings, pervasive spaces, energy systems, transportation systems, agriculture and food systems, and supply networks.

The analysis and modeling of non-equilibrium (open) biologic systems that are capable of withstanding stress and of auto-regulating across multiple scales of time and space may be part of a useful strategy in the innovative design of more robust (high confidence) cyber-physical systems. Furthermore, enhanced understanding of the dynamics of pathologic biologic systems may motivate new quantitative measures to forecast the breakdown of man-made systems.

The goals of the workshop were to: assess the current state-of-the art in system modeling and analysis; identify the limitations and critical gaps in existing theories and modeling formalisms; develop benchmark challenge problems whose solutions could have high impact on the design, analysis, and maintainability of critical systems and the understanding of complex natural systems; recommend a research agenda for developing new technologies to enable more rapid advance in areas such as biological science and the science of cyber-physical systems; recommend new directions in education.

Workshop participants included leading researchers in mathematics, biology, formal methods, computer science, chemistry, physics, control theory, and en-

gineering. The workshop agenda included invited presentations on a variety of topics related to open systems, followed by several open discussion sessions probing issues raised and identifying potential grand challenge problems.

Several basic concepts and questions were suggested by the organizers as topics to be addressed at the workshop.

1. How do we decompose entropy into extensive and nonextensive elements and what are their effects on a). open vs. closed systems b). reversible vs. irreversible components/systems?

2. What are the advantages of Tsallis entropy [39] over other forms of entropy such as Boltzmann-Gibbs entropy, especially with respect to our open vs. closed system questions?

3. What are the properties and constraints of the stochastic processes that are fundamental to open systems?

4. What do measures such as entropy, energy, and correlations tell us about a system? How can we use these concepts to design, analyze, and understand complex systems such as biological systems or cyber-physical systems?

5. How can methods for managing complexity, such as abstraction and composability, be applied in the context of open systems?

The group identified challenges in constructing a transformative agenda for open systems. They recommended the development of a combined mathematical and logical framework for modeling the behavior of open dynamic networks that evolve over time. The framework would provide a unified approach to build diverse top down and bottom up models that are consistent with each other and with the system (biological, cyber-physical, or other) being modeled. The identification of a family of properties and functions of open systems that can account for the organization and ability for self-regulation, maintainability, and adaptability is a key to designing and composing systems in principled ways.

Bruce Alberts [40], the past President of the National Academy of Sciences, stated in 1998 that “the education that we are offering today to young biologists in our colleges and universities is seriously in need of a major rethinking ... the result is a major mismatch between what todays students who are interested in biology should be learning and the actual course offerings that are available to them.”

It seems that significant educational progress has not been made since Alberts identified the problem. Effort remains to determine what preparation in physics, chemistry, and mathematics is most appropriate for either the research biologists or the medical doctors who will be working in the coming decades. A similar situation is occurring in the emerging field of cyber-physical systems.

It is important to provide students with cross disciplinary courses and projects that encourage (even require) exploration of new ways of thinking and problem solving. An important example is combining computational thinking/modeling using multiple methods with experimental science topics. This should include principles for designing and building computation models, articulating the questions a model should try to answer, and deciding the appropriate mathematical/logical tools to bring to bear. In addition, learning methods for model

validation is crucial and will lead to important critical thinking skills. Validation should be taken in a broad sense to include questions such as: is the model right; does it capture what was intended, does it exhibit behavior consistent with experimental observations and intuitions; why does it work, i.e. how does it explain the system being modeled?

At the center of the open systems challenge is the realization that modern science has progressed to studying complex processes that dynamically change by continually interacting with their environment. The multi-scale approach developed to address this complexity relies on modeling assumptions, and the appropriateness of a model depends on the appropriateness of these assumptions. Continuous deterministic differential equations arise from discrete (particle-based) probabilistic models, and moving between these two extremes is often subtle and poorly understood; especially the relationship between rare events, determinism, and nonextensive entropy. New mathematical methods will arise by combining diverse techniques using scientific methodologies, computer science, and engineering. New forms of interaction with evolutionary biological networks will be key in the discovery process. Challenge problems such as high confidence supply networks and medical device systems as well as mathematical models of the immune system exhibit these complex behaviors. In particular, the design and evaluation of medical device systems present challenges in systems integration, critical infrastructure, embedded real-time systems design, and their validation and certification. The corresponding research directions to meet these challenges address infrastructure for medical device integration and interoperation, model-based development, component-based design frameworks, patient modeling and simulation, adaptive patient-specific algorithms, and user-centered design. Insights into these challenges and the search for a unifying set of principles for open systems that bring these issues together have potential impact for significantly influencing new directions in research and education.

Acknowledgements. We thank Carolyn's CSL colleagues at SRI for sharing their experiences with her as a leader and mentor. In particular, the section on Carolyn's leadership at SRI was based on input provided by Linda Briesemeister, Grit Denker, Ashish Gehani, Keith Laderoute, Patrick Lincoln, Merrill Knapp, Andy Poggio, Malabika Sarker, Rukman Senanayake, Mark-Olliver Stehr, and Ashish Tiwari. A special thanks goes to Grit Denker for providing input, organizing and coordinating comments from her coworkers, and serving as the chair for local arrangements. We also thank Gul Agha, Olivier Danvy, and José Meseguer for organizing the Festschrift 2011 Symposium, a well deserved tribute to Carolyn.

References

1. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. *Communications of the ACM* (1978)
2. Dolev, D., Yao, A.C.: On the Security of Public Key Protocols, STAN-CS-81-854 (1981)

3. Millen, J.: The Interrogator: A Tool for Cryptographic Protocol Security. In: Proceedings 1884 Symposium on Security and Privacy. IEEE Computer Security Society, Los Alamitos (1984)
4. Meadows, C.: The NRL Protocol Analyzer: An overview. *Journal of Logic Programming* (1996)
5. Kemmerer, R.: Analyzing Encryption Protocols Using Formal Verification Techniques. *IEEE Journal Selected Areas in Communication* 7(4) (1989)
6. Meadows, C.: Applying formal methods to the analysis of a key management protocol. *The Journal of Computer Security* 1(1) (1992)
7. Abadi, M., Needham, R.: Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering* 22(1) (1996)
8. Denker, G., Megeguer, J., Talcott, C.: Protocol Specification and Analysis in Maude. In: Workshop on Formal Methods and Security Protocols (1998)
9. Denker, G., García-Luna-Aceves, J.J., Megeguer, J., Ölvezky, P., Raju, J., Smith, B., Talcott, C.: Specifying a Reliable Broadcasting Protocol in Maude. In: Workshop on Formal Methods and Security Protocols (1998)
10. Denker, G., Megeguer, J., Talcott, C.: Formal Specification and Analysis of Active Networks and Communication Protocols: The Maude Experience. In: DARPA Information Survivability Conference and Exposition (2000)
11. Ölvezky, P., Megeguer, J., Talcott, C.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In: Formal Methods in System Design (2006)
12. Mason, I., Talcott, C.: Simple Network Protocol Simulation within Maude. In: Third International Workshop in Rewriting Logic and Its Applications. Electronic Notes in Theoretical Computer Science (2000)
13. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Megeguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
14. Fábrega, J.T., Herzog, J., Guttman, J.: Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security* 7 (1999)
15. Guttman, J., Fábrega, J.T.: Authentication Tests and the Structure of Bundles. *Theoretical Computer Science* (2001)
16. Guttman, J., Fábrega, J.T.: The sizes of skeletons: security goals are decidable, MITRE Technical Report 05B09 (2005)
17. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055. Springer, Heidelberg (1996)
18. Rushby, J.: The Needham-Schroeder Protocol in SAL, Computer Science Laboratory, SRI International (2005)
19. Cervesato, I., Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: A comparison between Strand Spaces and Multiset Rewriting for Security Protocol Analysis. In: Software Security - Theories and Systems - ISSS (2002)
20. Anlauff, M., Pavlovic, D., Waldinger, R., Westfold, S.: Proving Authentication Properties in the Protocol Derivation Assistant, Kestrel Institute (2006)
21. Owre, S., Shankar, N., Rushby, J.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
22. Talcott, C.: A Maude-PVS tool for Strand Spaces, Protocol eXchange (2004)
23. Talcott, C., Owre, S.: CPSA + Maude + PDA + PVS, Protocol eXchange (2005)

24. Talcott, C.: S-expressions & Maude + PVS, Protocol eXchange (2006)
25. Owre, S.: Maude2PVS, Protocol eXchange (2007)
26. Talcott, C.: TOOLIP Semantics & TOOLIP - Maude NPA, Protocol eXchange (2007)
27. Talcott, C.: TOOLIP Semantics & Interoperation, Protocol eXchange (2008)
28. Talcott, C.: Pathway Logic. In: Bernardo, M., Degano, P., Tennenholtz, M. (eds.) SFM 2008. LNCS, vol. 5016, pp. 21–53. Springer, Heidelberg (2008)
29. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Talcott, C.: Pathway Logic: Executable Models of Biological Networks. In: Fourth International Workshop in Rewriting Logic and Its Applications. Electronic Notes in Theoretical Computer Science (2004)
30. Talcott, C., Eker, S., Knapp, M., Lincoln, P., Laderoute, K.: Pathway Logic Modeling of Protein Functional Domains in Signal Transduction. In: Proceedings of the Pacific Symposium on Biocomputing (2004)
31. Edelman-Keshet, L.: Mathematical Models in Biology. McGraw-Hill, New York (1988)
32. Sontag, E.: Lecture Notes on Mathematical Systems Biology, Rutgers University (2009),
http://www.math.rutgers.edu/~sontag/FTP/_DIR/systems_biology_notes.pdf
33. Izhikevich, E.: Which Model to Use for Cortical Spiking Neurons? IEEE Transactions on Neural Networks 15(5) (2004)
34. Tiwari, A., Talcott, C.: Analyzing a Discrete Model of Aplysia Central Pattern Generator. Computational Methods in Systems Biology (2008)
35. Poggio, A.: New Insights for Network Science: Discrete Mathematical Models of Biological Systems, Computer Science Laboratory, SRI International: ECU 09-416R (2010)
36. Colizza, V., Barrat, A., Barthélemy, M., Vespignani, A.: The role of the airline transportation network in the prediction and predictability of global epidemics. Proceedings of the National Academy of Sciences 103(7) (2006)
37. Grais, R., Ellis, J.: Modeling the spread of annual Influenza epidemics in the U.S.: the potential role of air travel. Health Care Management Science 7 (2004)
38. Barrat, A., Barthélemy, M., Pastor-Satorras, R., Vespignani, A.: The architecture of complex weighted networks. Proceedings of the National Academy of Sciences 101(11) (2006)
39. Tsallis, C.: Is the entropy S_q extensive or nonextensive? Astrophysics Space Sciences 305 (2006)
40. Alberts, B.: The Cell as a Collection of Protein Machines: Preparing the Next Generation of Molecular Biologists. Cell 92 (1998)

Ten Years of Analyzing Actors: Rebeca Experience

Marjan Sirjani^{1,2} and Mohammad Mahdi Jaghoori³

¹ School of Computer Science, Reykjavik University, Reykjavik, Iceland

² University of Tehran, Tehran, Iran

³ CWI, Amsterdam, The Netherlands
marjan@ru.is, jaghoori@cwi.nl

Abstract. In this paper, we provide a survey of the different analysis techniques that are provided for the modeling language Rebeca. Rebeca is designed as an imperative actor-based language with the goal of providing an easy to use language for modeling concurrent and distributed systems, with formal verification support. Throughout the paper the language Rebeca and the supporting model checking tools are explained. Abstraction and compositional verification, as well as state-based reduction techniques including symmetry, partial order reduction, and slicing of Rebeca are discussed. We give an overview of a few extensions of timed actor-based models and formal techniques to check correctness of self-adaptive systems using Rebeca. A summary of design decisions and a brief general comparison of the analysis methods are provided at the end of the paper while specific sections are accompanied with examples and corresponding related work.

Keywords: Actors, Rebeca, Concurrency, Formal Verification, Model Checking, Reduction Techniques, Abstraction.

1 Introduction

As information networks are becoming increasingly important in our society, the number of distributed heterogeneous software systems is rapidly growing. Distributed systems consist of multiple cooperating components where the components are typically encapsulated systems or objects spread over a network, interacting via asynchronous communication. Web-service applications and applications based on wireless network technologies are examples of such distributed and asynchronous applications. Such technologies are now used in a vast variety of applications, such as medical systems, transportation systems, and the significant business of online gaming.

The actor model is among the pioneering ones to address concurrent and distributed applications. The actor language was originally introduced by Hewitt [44] as an agent-based language for programming distributed systems, and was later developed by Agha [3,4,5] into a concurrent object-based model. Valuable work has been done on formalizing the actor model by Talcott et al.

[5,81,113,114]. The actor model has been used both as a framework for theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent and distributed systems.

In the actor model, *actors* are the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message that it receives. Actors have encapsulated states and behavior, and are capable of creating new actors and redirecting communication links through exchange of actor identities.

Different interpretations, dialects and extensions of the actor model are proposed in several domains, for example, designing embedded systems [78] and wireless sensor networks [19]. The actor model is further claimed to be the suitable model of computation for the most dominating application domains of multi-core programming and web services [45,17,16]. Compared to mathematical modeling languages, like process algebras, actors are more natural for designers, software engineers, modelers and programmers. Compared to process-oriented models, like Petri nets, the actor model has the advantages of an object-based language, like encapsulation of data and process, and more decoupled modules. Moreover, the formal semantics of actor-based languages builds a firm foundation for formal analysis and verification.

The actor model of concurrency is getting more and more popular in practice [64,46,47]; as a few examples Erlang [37] and Scala [98] are two programming languages that have applied the actor model of concurrency and are now getting widely used. The Asynchronous Agents Library is an actor-based framework that is added to Microsoft Visual Studio 2010 [85]. Actors have been used in real-world applications like Twitter's message queuing system, and Vendetta's game engine (in 2010) [76].

Applying formal methods in software engineering is an important step towards building more reliable and robust systems. In the more than twenty-five years since its invention, model checking has achieved multiple breakthroughs, bridging the gap between theoretical computer science and practical computer engineering. Today, model checking is extensively used in the hardware industry, and has also become feasible for verification of many types of software [22]. State space explosion is the dominant problem for model checking; investigating new abstraction and reduction techniques is the leading edge in this research area.

Reactive Objects Language, Rebeca [102,106], is an operational interpretation of the actor model with formal semantics and model checking tools. In a Rebeca model, system components are sets of reactive objects, called *rebecs*, which communicate with each other and with their environment, through message passing. Message passing is asynchronous and fair. Messages related to each rebec are stored in the message queue of the rebec. The computation takes place by taking the message at the head of the message queue and executing the corresponding message server [106].

Ten years ago we noticed the urgent need for a formal tool that could be easily used by software engineers. By observing the increasing number of concurrent and distributed systems on one hand, and popularity and efficacy of object-oriented approaches on the other hand, we identified the actor model as the best candidate to be the basis of our research. So, Rebeca was designed to bridge the gap between formal methods and software engineering. In Rebeca, reactive objects are units of concurrency. Compared to thread-based concurrent programming, Rebeca makes the modeling of concurrency more natural and less error-prone. Moreover, it brings transparency by removing the difference between local and non-local concurrency in a distributed system. The recent widespread use of actors in different areas and applications, like in distributed applications, or safe/sound programming of emerging multicore hardware, clearly demonstrates its power as a computational model [45,47].

To the best of our knowledge, the first attempt to provide compositional verification and model checking support for an imperative actor-based language is the introduction of Rebeca in 2001 [102,103,104]. We defined the language Rebeca and its formal semantics, developed its model checking tools, and provided a compositional verification theory and abstraction techniques. We have been actively and successfully investigating specialized reduction techniques for formal verification of Rebeca models, namely, symmetry, partial order, and slicing, that are all based on the formal semantics of the language [107,105,108,58,56,48,97,96]. Rebeca with its simple message-driven object-based computational model, Java-like syntax, and the associated set of verification tools, is an interesting and easy-to-learn model for students, software engineers, and practitioners.

In this paper, we present a survey of the different analysis techniques that are developed for Rebeca. We¹ first started model checking using back-end model checking tools of Spin [111] and NuSMV [88], and then proceeded to develop direct model checking tools (Section 3). To tackle the state space explosion problem, we developed a theory for abstraction and compositional verification of Rebeca models (Section 4). Then we moved towards investigating and applying reduction techniques in model checking, including symmetry and partial order reduction (Section 5), and slicing (Section 6). Our focus has been on finding specific reduction techniques that exploit the specific features of Rebeca models. In our techniques for symmetry reduction, partial order reduction, slicing and distributed model checking, we perform static analysis on the model and use the results to tackle the state space explosion problem.

The language Rebeca is used in different research and application areas. We established schedulability analysis for real-time actor-based models and used Rebeca to represent our approach (Section 7). We have also extended Rebeca to serve as a policy-based coordination language in modeling self-adaptive systems, and we have established corresponding techniques to check their correctness (Section 8).

¹ Please note that the word *we* in this paper refers to all those who have contributed and are working on designing and developing Rebeca, and the related theories, tools, applications, and extensions.

2 Rebeca Syntax and Semantics

Rebeca [104,105,106] is a modeling language with a formal semantics based on an operational interpretation of the actor model [44,4]. The definition of a Rebeca model consists of a set of reactive classes plus an initial configuration in its **main** section where a set of *rebecs* (reactive objects) are created as instances of reactive classes (see Fig. 1 for the syntax of Rebeca). Each rebec has a single thread of execution. The behavior of a Rebeca model is defined by the fair and interleaved execution of the rebecs.

Rebecs communicate *only* through asynchronous message passing and have unbounded buffers for automatically storing the incoming messages, i.e., there is no statement in Rebeca syntax to explicitly wait for receiving a message. When a rebec is scheduled to run, the message at the head of the queue is taken out and processed. Each message that can be serviced by a rebec has a corresponding message server, which is given in the definition of the reactive class denoting the type of the rebec. Message servers are executed atomically, thus, Rebeca is said to have coarse-grained (“big-step”) interleaving. Although each rebec has one queue, we can model multiple reception queues between different rebecs by adding an extra rebec to represent each reception queue.

A **reactiveclass** definition takes an integer argument to denote an upper-bound on the length of the message queue; this is used in model checking and can be increased in the case of a queue overflow. The body of the **reactiveclass** consists of three parts: the known rebecs section includes a set of rebec identifiers and as such forms the initial communication topology of the system; variables constitute the local state of a rebec; and, message servers (also called methods) define the behavior. Each message server may declare local variables and further contains a sequence of statements, including assignments, if statements, rebec creation (**new**), and method calls. A method call is equivalent to sending an asynchronous message that invokes the corresponding message server (method). By sending rebec variables around (i.e., the variables holding a rebec identifier),

$Model ::= Class^* Main$	$Stmt ::= v = e; \mid v = \mathbf{new} C(\langle e \rangle^*);$
$Class ::= \mathbf{reactiveclass} C(Nat)$	$\mid Call(\langle e \rangle^*);$
$\quad \{KR s \ Vars \ MsgSrv^*\}$	$\mid \mathbf{if} (e) \ MSt \ [\ \mathbf{else} \ MSt \]$
$KRs ::= \mathbf{knownrebecs} \{ \langle Vdcl; \rangle^* \}$	$Call ::= v.M \mid \mathbf{self}.M \mid \mathbf{sender}.M$
$Vars ::= \mathbf{statevars} \{ \langle Vdcl; \rangle^* \}$	$Mst ::= \{ Stmt^* \} \mid Stmt$
$Vdcl ::= T \langle v \rangle^+$	$Main ::= \mathbf{main} \{ Reb^* \}$
$MsgSrv ::= \mathbf{msgsrv} M(\langle T \ v \rangle^*) \{ Stmt^* \}$	$Reb ::= T \ r(\langle T \ r \rangle^*) : (\langle T \ e \rangle^*);$

Fig. 1. BNF grammar for Rebeca classes. Angle brackets $\langle \dots \rangle$ are used as meta parentheses, square brackets $[\dots]$ for optional parts, superscript $+$ for repetition more than once, superscript $*$ for repetition zero or more times, whereas using $\langle \dots \rangle$, with repetition denotes a comma separated list. Identifiers C , T , M , r and v denote class, type, message server, rebec and variable names, respectively; Nat denotes a natural number; and, e denotes an (arithmetic, boolean or nondeterministic choice) expression.

<pre> 1 reactiveclass Sender(4) { 2 knownrebecs { 3 Receiver r; 4 } 5 statevars { 6 int req; 7 boolean pass; 8 } 9 msgsrvv initial() { 10 req = 1; 11 r.receiveReq(req); 12 } 13 msgsrvv sendNextReq() { 14 pass = ?(true,false); 15 if(pass) req = req + 1; 16 if(req == 5) req = 1; 17 r.receiveReq(req); 18 } 19 } 20 main() { 21 Sender s(r):(); 22 Receiver r(s):(); 23 } </pre>	<pre> 1' reactiveclass Receiver(4) { 2' knownrebecs { 3' Sender s; 4' } 5' statevars { 6' int msg; 7' boolean isFinal; 8' } 9' msgsrvv initial() { 10' isFinal = false; 11' } 12' msgsrvv receiveReq(int m) { 13' msg = m; 14' if(msg == 4) 15' isFinal = true; 16' else 17' isFinal = false; 18' s.sendNextReq(); 19' } 20' } </pre>
--	--

Fig. 2. The Rebeca code of the sender/receiver example

the topology can change dynamically. In each reactive class, there is at least one message server, called ‘initial’; this is responsible for initialization tasks (like ‘constructors’ in object oriented programming languages). Each rebec receives this message implicitly upon creation. The system continues running as long as there is at least one message to be processed. To instantiate a **reactiveclass** in the **main** section, one should provide first the bindings of the **knownrebecs** and then the parameters to the initial message server (if any).

2.1 A Sender/Receiver Example

As a running example, we use a simple model of a sender and receiver (shown in Fig. 2). We use slightly different versions of this example in different sections to demonstrate how the techniques in that section can be used in practice. This example is similar to the *alternating bit protocol*, but we simplified it by putting a nondeterministic assignment (line 14) instead of getting a real acknowledgment from the receiver.

There is a rebec in this example that acts as a sender and sends a number of messages (four here) to a receiver (the other rebec). Based on the nondeterministically chosen value of the variable `pass` (line 14), the sender rebec either sends a new message (line 15) or repeats the previous one. The happy scenario is when the receiver receives all four messages, after which `isFinal`, a state variable of the receiver, is set to `true` (line 15’). Every time `receiveReq` is executed by the receiver, a `sendNextReq` is sent back to the sender asking for the next message (line 18’). After receiving the last message by the receiver this scenario starts over again.

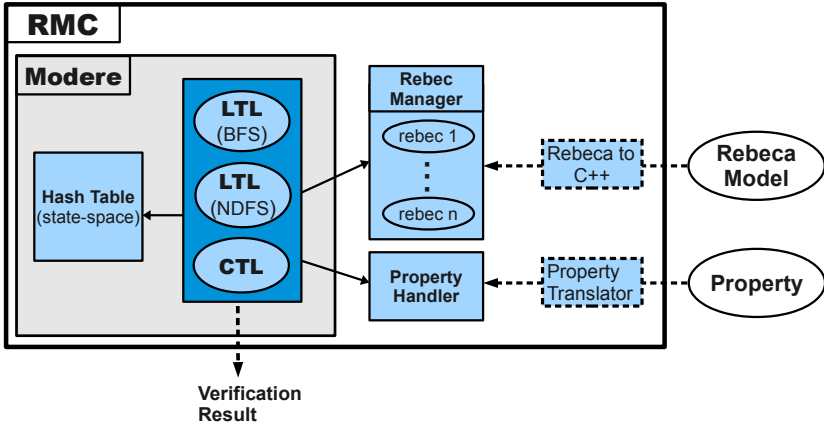


Fig. 3. The architecture of Rebeca Model Checker (RMC) — the solid arrows show calling another component (control dependencies), and dashed arrows show flow of data (input/output of the tool and the data dependencies)

A possible interesting property for this example is $G(F(isFinal == true))$ which checks whether the last message is always finally received by the receiver. The property is an LTL (Linear Temporal Logic) formula [35] where G denotes *globally* and F denotes *finally*. In addition, the example can be verified against deadlock. Deadlock occurs for example if we remove line 18' from the model.

3 Model Checking Tools

Since 2005, Rebeca has a custom-made explicit-state model checker [56,58] implemented in C++. The advantage of a tailor-made tool is that it can take into account the intrinsic features and the nature of the concurrency model of Rebeca. This amounts to a more efficient tool, provided that it is debugged as any other software and it is maintained and kept up-to-date with respect to the state of the art in tools and algorithms. However, model checking Rebeca has been possible since 2003; before the development of the custom model checking tool, this was achieved by translation into the input languages of famous and prominent model checkers [103,110,107,48], namely, SMV [20], Promela (SPIN) [49] and later mCRL2 [42]. The main advantage of using such translations is that we can benefit from the maturity of the back-ends of established tools. In this section, we give an overview of the tools developed for model checking Rebeca directly.

Fig. 3 shows the architecture of Rebeca Model Checker (RMC) which is composed of loosely coupled components. The main component is *Modere*, the Model-checking Engine of Rebeca. A model checking engine is in essence a highly

optimized and memory efficient search algorithm, e.g., DFS or BFS, which is extended to check for temporal properties specified, e.g., in LTL or CTL (Computational Tree Logic) [35]. Modere was first developed as an LTL model checker; Vardi [118] argues that LTL as a linear-time temporal logic is more expressive and intuitive and supports compositional reasoning, unlike CTL as a branching time temporal logic. Nevertheless, CTL is advantageous, for example, in specifying reachability goals. Modere can now check for properties in both LTL and CTL, thus, it can fulfill the needs of a wide range of practitioners. It also has implementations of both DFS and BFS based search algorithms. The modular architecture of the RMC tool-set has allowed us to make these extensions to Modere while reusing the other components, e.g., Rebeca to C++ converter.

The rebec manager component in RMC is model dependent and is created by a Rebeca to C++ translator (see Fig. 3), it contains a C++ equivalent of the reactive class definitions and the instantiation of these classes as specified in the Rebeca model. During model checking, Modere repeatedly puts the rebecs in a specific state and asks the rebec manager to run one rebec, the rebec manager returns the resulting state(s) afterwards. A possible extension point of the tool-set is to replace the rebec manager with a process/object manager for another language with a similar actor-based computational model.

3.1 Bounded Model Checking

Bounded model checking has also been studied for Rebeca using the SMT (Satisfiability Modulo Theories) solver tool Yices [33]. Such a tool checks the satisfiability of a given formula; this is in theory NP-hard, but there are several efficient heuristics for this problem that work in practice and many tools have been developed. This approach is particularly useful to model check Rebeca for data-centric applications.

To use an SMT-solver, first the general concurrency and communication model of Rebeca needs to be defined as a set of formulas [59]. And then given a specific model to be verified, the model needs to be translated into compatible formulas. Finally, the desired property is also turned into formulas, such that the conjunction of these three sets of formulas: “*RebecaConcurrency* \wedge *RebecaModel* \wedge *Property*” is satisfiable if and only if the given property holds for the given model.

Table 1. Defining Rebeca concepts as formulas

Rebeca Concept	type	description
(isActiveRebec $i t$)	bool	Rebec i is active at step t
(isActiveMsgsrv $j i t$)	bool	Message server j of rebec i is active at step t
(qSize $i t$)	int	The size of queue of rebec i at step t
(queue $i j t$)	int	Element i in queue of rebec j at step t .

To encode the general concurrency model of Rebeca, we first need to model the basic concepts of a Rebeca model. A possible encoding for this is shown in Table 1. In this encoding, all messages are represented as integers (cf. the type of queue elements in Table 1). The parameter t in this encoding models the execution steps; for example, to send a message at step t to rebec i , we need to make sure that it exists in the queue of rebec i at step $t + 1$ and furthermore, it should hold that $(\text{qSize } i \ t + 1) = (\text{qSize } i \ t) + 1$.

As the next step, to encode a specific model, we need to set up some constants, like the number of rebecs in the system, as well as translate the class and message server definitions. The latter is achieved again by defining the changes that need to take place as a transition from step t to step $t + 1$.

Encoding temporal properties, e.g., LTL, is done by unfolding the property. For example a property Gp is translated to $p(0) \wedge \dots \wedge p(k)$ where $p(t)$ asserts the satisfiability of p at step t and k is the bound we assume on the number of steps in bounded model checking. If a counter-example is obtained with this bound, we know that the desired property does not hold for the system. But the satisfiability of the desired property up to k steps cannot guarantee the correctness of the model in general. One can increase this bound to obtain more general results as far as the physical (memory and time) restrictions of the hardware allow it.

3.2 Domain Specific Model Checking: SystemC Designs

SystemC [89] is an object-oriented language that has emerged lately as the leading language for system-level modeling. In the project Sysfier [2], a tool is integrated into Rebeca model checking tools to map SystemC designs to Rebeca models and then use Rebeca verification tool-set to verify LTL and CTL properties [10,92]. Many examples are translated from SystemC to Rebeca and are model checked against LTL and CTL properties.

To model SystemC designs, Rebeca is extended by adding global variables and wait statements. Global variables are used in a very limited way to model events and signals. A *wait* statement is used when a process needs to wait for a specific event before it can continue. The simulation kernel of SystemC is mapped to a rebec which plays the role of a synchronizer.

The inherent similarities of the two languages prevents any unwanted overhead or additional states. As a matter of fact, the theories in abstraction and compositional verification of Rebeca and the tools and techniques for state space reduction can be applied to SystemC verification, too. Moreover, the Rebeca model checkers are equipped with different policies based on the semantics of SystemC to reduce the state space. One policy that considerably reduces the size of the generated state space is to mimic the behavior of the SystemC simulation kernel and consider the sequential execution of rebecs instead of all possible interleavings. This policy works when verifying race-free SystemC designs. Another policy is to apply partial order reduction based on SystemC semantics.

4 Compositional Verification and Abstraction

In a broad sense, compositional verification tackles the state-space explosion problem by verifying the constituent components of a system in isolation: since these components are smaller in size, they are more amenable to computerized analysis, e.g., model checking. The correctness properties of the system are then derived from the properties of its individual components [21,73,83].

In general, compositional verification may be exploited more effectively when the model is naturally decomposable [95]. Actor-based models provide such inherently independent modules because there are no shared variables, but explicit non-blocking send operations are the only way of communication.

In compositional verification of Rebeca [106], we follow a top-down approach, where components are sub-models and are the result of decomposing a closed model. To this end, we take one part of the closed model as an open component and the rest of the model is assumed to be the environment. With such decomposition, the rebecs in the selected component will be modeled with their state and behavior, whereas the state-space of the *external* rebecs, i.e., those in the environment, is not modeled because their methods are not executed. External rebecs are only modeled as their potential in sending messages.

In an unrestricted environment, the general so-called *environment problem* arises, which states that the reachable state space of an open component, as described above, may in fact be much larger than that of the original closed model. In fact, putting the messages sent from external rebecs into the queues will immediately overflow the queues. To alleviate this problem, we model a reduced environment which can be considered as a *compositional minimization*. To do so, we consider the set of external messages always enabled, and consider a fair choice between executing these messages and the message on the top of the queue. This way we also avoid explicitly modeling the environment, for example, as another rebec.

In order to prove certain properties, we may need some assumptions about the environment, but in general we do not apply assume/guarantee reasoning. Our compositional verification generates an over-approximation of the behavior of the components. We proved a weak simulation relation between any closed model including the component and the component composed with the abovementioned environment, and hence we can claim that safety properties are preserved [106].

Another view to compositional verification is a bottom-up approach. In [108], we discuss such an approach along the lines of modular verification where the concept of a component is an independent module with a well-defined interface. Such notion of a component represents a re-usable off-the-shelf module. Once verified, this module has a fixed proven specification and then it can be used to build reliable systems. In [108], this modular verification technique for Rebeca is presented. In the both approaches described above, although the strategy in abstraction techniques is the same, the technical details (to keep the theory valid) are quite different.

5 Symmetry and Partial Order Reduction

Two of the most widely used state-space reduction techniques in explicit-state model checking are symmetry [36,51,25,86] and partial order reduction [41,117]. These two are the first reduction techniques implemented in the model checking engine of Rebeca (Modere) [58,56] because they fit naturally the asynchronous object model of Rebeca and yield reasonable reductions.

5.1 Partial Order Reduction

The idea of partial order reduction (POR) is that it is not always necessary to consider all of the possible interleaved sequences of the enabled actions. Instead, the execution of some of them can be postponed to a future state without affecting the validity of the correctness property. This way, the full interleaving of those actions is avoided and the size of the explored state space is reduced.

A popular approach to implementing POR is based on statically detecting *safe* actions, called static POR. This approach is applicable to Rebeca only in absence of dynamic rebec creation or change of topology. The characteristics of safe actions are described in the following. The first characteristic is invisibility, i.e., not changing the satisfiability of the correctness property. In the sender/receiver example in Fig. 2, assume the correctness property is $G(F(isFinal == true))$. The message server `sendNextReq` is then invisible as it does not change the variable `isFinal`. In addition, the initial message servers are also by definition invisible [56], because variables are uninitialized before that.

Two actions are said to be independent if one cannot disable the other; an action that is independent of all other actions is called globally independent. Due to absence of shared variables, assignments are local and hence globally independent; therefore, independence of a message server depends only on its send statements (recall that no `new` statement is allowed in static POR). Sending a message from $r1$ to $r2$ is globally independent if $r1$ is the only rebec that may send messages to $r2$ [56]. In the sender/receiver example, since only `s` sends messages to `r` and vice versa, i.e., there are even no self calls, all rebec actions in this example are globally independent.

An action is safe if it is safe and globally independent. The model checker can execute a safe action without considering its interleaving with other actions. The initial message servers as well as `sendNextReq` correspond to safe actions in our example.

Considering the coarse-grained interleaving of Rebeca, an action corresponds to a message server; therefore, POR amounts to a considerable reduction when applicable. Furthermore, as mentioned above the initial message server is always invisible. This makes direct application of POR in Modere more efficient than translating for example to Promela where you will lose such useful information.

5.2 Symmetry Reduction

The symmetry reduction technique views the state space as a graph: the states are the vertices and the transitions are the edges. The idea then is to partition

<pre> 2 reactiveclass SenderReceiver(4){ 3 knownrebcs { 4 SenderReceiver peer; 5 } 6 statevars { 7 int req; 8 boolean pass; 9 int msg; 10 boolean isFinal; 11 } 12 msgsrvv initial() { 13 req = 1; 14 peer.receiveReq(req); 15 isFinal = false; 16 } 17 msgsrvv sendNextReq() { 18 pass = !(true, false); </pre>	<pre> 20 if(pass) req = req + 1; 21 if(req == 5) req = 1; 22 peer.receiveReq(req); 23 } 24 msgsrvv receiveReq(int m) { 25 msg = m; 26 if(msg == 4) 27 isFinal = true; 28 else 29 isFinal = false; 30 sender.sendNextReq(); 31 } 32 } 33 main() { 34 SenderReceiver sr1(sr2):(); 35 SenderReceiver sr2(sr1):(); 36 } </pre>
---	--

Fig. 4. A symmetric sender/receiver reactive class

the state space into equivalence classes corresponding to isomorphic graphs and use one state as the representative of each equivalence class. The problem in symmetry reduction is that calculating the representative states, known as the ‘constructive orbit problem,’ is NP-hard [23]. This is usually alleviated by first *specifying* or *detecting* the symmetry among higher-level constructs (such as processes or objects) using some static analysis and then applying it in solving the orbit problem. The most popular approach to explicitly specify symmetry in a system is the notion of scalar sets, proposed by Ip and Dill [51], which is also later used by others, e.g., [13], [43].

In Rebeca, since rebecs instantiated from the same reactive class exhibit similar behavior, any symmetry in the communication structure of a model leads to symmetry in the underlying state-space graph; we call this *inter-rebec symmetry*. Detecting such symmetries does not rely on any symmetry-related input from the modeler. In [58] we proposed a polynomial-time solution for detecting structural symmetry without requiring any change in the syntax. The sender/receiver example in Fig. 2 is not symmetric because the rebecs in the model are of different types. We revisit the example in this section by merging the two reactiveclasses thus enabling a rebec to act both as a sender and a receiver, shown in Fig. 4. The composition of the two rebecs, specified in the **main** section, is now symmetric, i.e., by swapping the names of the rebecs we obtain:

```

main {
  SenderReceiver sr2(sr1):();
  SenderReceiver sr1(sr2):();
}

```

which can be changed into the original bindings by reordering the lines. This can be contrasted to the example in Fig. 5, where there is a central receiver with three senders. In this case, swapping the names of the rebecs of type Sender does not yield any symmetry, because this changes the knownrebcs binding of the Receiver rebec.

<pre> 2 reactiveclass Receiver(4) { 3 knownrebecs { 4 Sender s[i:1..3]; 5 } 6 statevars { 7 int msg[i]; 8 boolean isFinal[i]; 9 } 10 ... 11 } </pre>	<pre> 13 reactiveclass Sender(4) { 14 knownrebecs { Receiver r; } 15 ... 16 } 17 main{ 18 Sender s1(r):(); 19 Sender s2(r):(); 20 Sender s3(r):(); 21 Receiver r(s1,s2,s3):(); 22 } </pre>
--	---

Fig. 5. A symmetric star topology

The example in Fig. 5 is still symmetric if we make sure the implementation of the Receiver class is internally symmetric. We extended our automatic symmetry detection further to also consider *intra-rebec* symmetries [57]. In *intra-rebec* symmetry, we propose to use scalar-sets but for a different purpose from its usual use, i.e., we use scalar sets locally for each class to specify the symmetric behavior of that class with respect to its known rebecs (when applicable), rather than specifying the symmetry in the whole system, as in e.g. [52,14]. Our use of scalar sets is in line with the modular modeling encouraged by the actor model. This way we can consider the internal symmetry of rebecs along with the symmetry in their communication structure.

One of the topologies where *intra-rebec* symmetry can be used is a star network. Fig. 5 shows an example of sender and receiver reactive classes that are instantiated in a star topology. The definition of the Receiver class uses a scalar set i in defining the known rebecs. This implies that the definition of the reactiveclass must be symmetrical which is guaranteed by syntactical restrictions; for example, the state variables in this class definition are defined per known rebec. Rebeca compiler will check statically whether the model is symmetric and in that case, it will generate the necessary information for the model-checking engine (Modere) to apply symmetry reduction.

5.3 Applicable Properties

When using POR, the model checker may only consider Linear Temporal Logic without the next operator (LTL-X) [41,117]; as we mentioned above, POR reduces the state space by postponing the execution of certain actions. This intuitively means that the ‘next-state’ behavior of the system is not preserved. When using symmetry, all temporal operators are allowed but the correctness property must also be symmetric. For example, with two instances of the SenderReceiver class (Fig. 4), the property asserting whether only sr1 can reach the goal, i.e., `isFinal = true`, is not symmetric; rather, we must check whether both sr1 and sr2 could reach this goal.

5.4 Related Work

The closest work to our inter-rebec symmetry detection is that of Donaldson, Miller and co-authors, who independently from our work, have proposed several techniques for detecting symmetry in similar models of computation (mainly Promela) [14,29,30,31,32]. To be able to automatically detect symmetries in Promela, they need to assume a communication pattern like that of Rebeca, namely using static channels. Another similar work is by Leuschel and Massart [79] where there is no need to extend the syntax of B to specify symmetry, because a built-in construct, called *deferred sets*, gives rise to symmetric data values in a way similar to scalar sets. Symmetry detection still depends on the proper use of deferred sets by the modeler, whereas in Rebeca, symmetry detection is based on the intrinsic communication mechanism of the language.

Basset [77], is a general framework for testing actor systems compiled to Java bytecode. Basset employs heap symmetry reduction which is based on data symmetries and is thus orthogonal to the structural symmetry reduction technique that we have applied. On the other hand, Basset implements a dynamic POR technique based on the happens-before relation and the causality among message send and receive events. There is no need for such heavy-weight dynamic POR in Rebeca due to the stronger assumptions of coarse-grained interleaving and FIFO ordering of messages. Unlike in Modere, the reported implementation of Basset has not yet combined the use of symmetry and POR.

6 Slicing

Program slicing is a program analysis technique with applications in various software engineering activities such as program understanding, debugging, testing, program maintenance, and complexity measurement [87]. Slicing can be used together with model checking and is orthogonal to a number of other reduction techniques. In [34] a significant reduction is reported by slicing concurrent object-oriented source code, and slicing is recommended because of its automation and low computational cost. In this section, we review slicing of Rebeca codes and its specific features and data dependencies, which are reported in detail in [96] and [97].

Static slicing [120] extracts statements from a program which have a direct or indirect effect on computations of other statements. More specifically, a program slice consists of the parts of a program that potentially affect the values computed at some statement of interest (referred to as the slicing criterion). A slicing criterion is usually denoted by $\langle p, V \rangle$, where p is a program statement and V is a set of variables.

The main challenge in this area is to efficiently build a precise slice. One of the main approaches to slicing is using reachability analysis on a program dependence graph. A program dependence graph consists of nodes representing the statements of a program, and edges representing the control and data dependencies. There is a control dependency between two statement nodes if one statement controls the execution of the other. Data dependency exists between

two statement nodes if the change made to a variable at one statement might reach the usage of the same variable at the other statement. The general approach to slicing is to find all paths in the graph that affect a specific variable in a specific statement. We build all such paths, and hence the slice, by following the dependency edges backward from the given statement.

In a simple sequential program without procedures, the slicing method is trivial. Applying the same simple method to programs with procedures, may build paths that are not *realizable*. By realizable we mean those paths that show a possible execution of the program. In order to remove the unrealizable paths, we need a *context-sensitive* analysis: the computation of a slice must preserve the calling context of the called procedures, and ensure that only paths corresponding to the legal call-return sequences are considered. Context-sensitive slicing can be done by generating summary edges at call sites: summary edges represent the transitive dependence edges of called procedures at call sites [70].

In concurrent programs with shared variables another type of dependence arises: interference. Interference occurs when a variable is defined in one thread and used in a concurrently executing one. Like above, a simple traversal of interference dependence edges during slicing can produce unrealizable paths and make the slice imprecise. To solve this problem, we need to consider the valid execution chronology [71]. Considering the chronological order of statements within a thread is not enough to build precise slices. The reason is that it is in general not possible to determine whether a definition reaches the statement of interest (slicing criterion), or it is always *killed* (disabled) by other definitions. As a result, we may mistakenly consider the statement including that definition in the slice. So, the interference dependency is not transitive.

Slicing object-oriented programs presents new challenges which are not encountered in traditional program slicing [87]. To slice an object-oriented program, features such as classes, dynamic binding, encapsulation, inheritance, message passing and polymorphism need to be considered carefully. Although the concepts of inheritance and polymorphism are strengths of object-oriented programming languages, they pose special challenges in program slicing. Due to inheritance and dynamic binding in object-oriented programs, the process of tracing dependencies becomes more complex than that in a procedural program.

6.1 Slicing Rebeca

In [97,96] we proposed slicing techniques for Rebeca which are based on Rebeca's actor-based computational model. We introduced a specialized control graph for Rebeca to capture its reactive behavior. One can perform data flow analysis on a Rebeca model by iterating over its control flow graph. Rebeca dependence graph is then introduced to represent different dependencies including control, data, intra-rebec, parameter-in, activation, member, and known-rebec dependencies. These two types of graphs are explained below.

A *Rebeca control flow graph (RCFG)* is defined based on the atomic execution of message servers and the reactive behavior of the rebecs. The control flow of the body of each message server is trivial but determining the control flow among

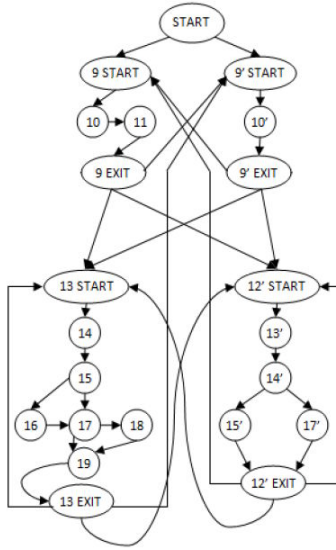


Fig. 6. RCFG of the sender/receiver model (from [96]). Line numbers refer to Fig. 2.

the message servers themselves is difficult. The control flow of a Rebeca model is different from that of multi-threaded concurrent programs due to its reactive and event-based nature. In Rebeca, method calls are different from regular procedure calls or initiating a thread. A method call is performed by sending an asynchronous message; the body of the corresponding message server is guaranteed to execute later atomically. A method call does not transfer the execution control from the *caller* to the *callee* nor does it generate a new concurrent thread. The main issue here is to determine which message servers can potentially execute after a given message server has finished. This depends on the messages on top of the queues of all rebecs at that time. As an over-approximation, one can assume that after the execution of a message server is over, all message servers potentially have a chance for execution. For a more precise approximation, one can use the causality relations.

As Rebeca is a well-structured language, control dependence can be computed during the traversal of the abstract syntax tree. The above approximation implies that the last definition of each variable in a message server can reach the first statements of all message servers. However, there are no shared variables in Rebeca, therefore, the only manifestation of this approximation is that a change in the value of a state variable within a rebec, reaches the other message servers of the same rebec. Figure 6 shows the RCFG of the sender and receiver example.

The *Rebeca Dependence Graph (RDG)* captures the features of Rebeca as follows (see Figure 7): Each reactive class is modeled as a *class-object* node. Each message server is modeled by an *entry* node, a set of nodes representing its statements, and *data dependence* edges and *control dependence* edges modeling the existing dependencies within the body of the message server. The set of message

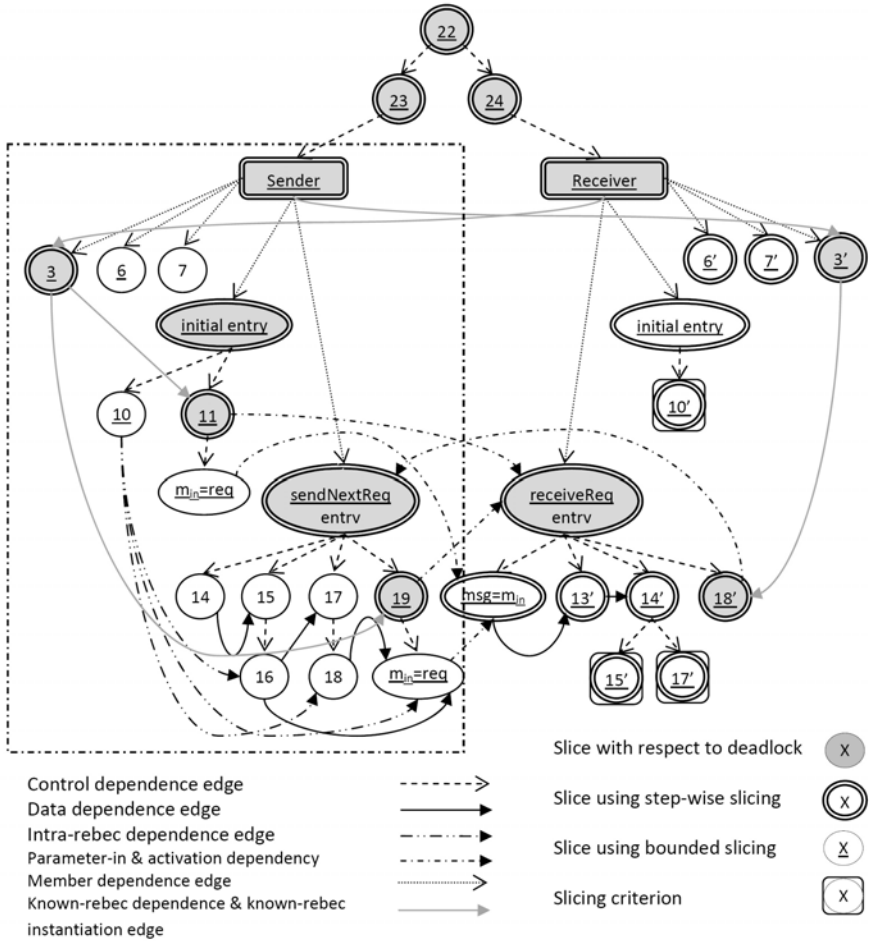


Fig. 7. RDG of the sender/receiver example (from [96]). Line numbers refer to Fig. 2.

servers of a reactive class are connected to the corresponding class-object node by *member dependence* edges. The member dependence edges ensure that a reactive class will be included in a slice if at least one of its message servers or state variables is included in that slice. Putting a message in a queue is represented by an *activation node*. In addition, an *activation edge* is used to connect the activation node to the *entry* node of the related message server and a *known-rebec dependence* edge is used to connect the activation node to the corresponding known rebec. Parameters of messages are modeled using *formal-in* and *actual-in* nodes along with *parameter-in* edges.

State variables are not shared among rebecs, but the message servers of each rebe share the state variables in the rebe. Therefore, there is a dependency between every message server using a variable and other message servers that assign a value to that variable. So, we introduce the notion of *intra-rebe dependency* to represent these kinds of dependencies. Considering the atomic execution of message servers, this dependency exists between the last statement of a message server assigning a value to a variable and the first use of that variable in other message servers (if the value of that variable is not changed in the body of the second message server before the first use).

To compute a slice from the resulting graph, four different algorithms are presented in [97,96]. The first one is the traditional reachability algorithm which is used for static slicing, and the second algorithm is checking a model for deadlock. The third and fourth techniques are based on iterative approximation and refinement of the model. In these techniques, an initial over-approximation of the original model is computed, and the model is subsequently refined based on the results of verification (alike in counter-example guided abstraction refinement - CEGAR [24]). The goal of these techniques is to find the minimal specification that satisfies a property, or otherwise a non-spurious counter-example. The reduced model is verified; if a spurious counter-example is found, then the model is refined to include more variables and the verification-refinement cycle is repeated. The *step-wise* slicing, starts with a reduced model including only the variables that construct the property, and the *bounded* slicing, is based on the nondeterministic assignments in Rebeca and user knowledge.

6.2 Related Work

A thorough general survey of slicing methods is provided in [70], and a survey of slicing techniques for object oriented programs is provided in [87]. In [34] slicing for concurrent object oriented programs is evaluated.

Compared to existing dependence graphs, the Rebeca dependence graph is simpler in several ways due to the asynchronous nature of communication, atomic execution of message servers, absence of shared data, and absence of procedure calls. In addition, Rebeca is an object-based language (as apposed to object-oriented), e.g., inheritance and polymorphism are not included in the language. So, we do not need to deal with the complexities of dependence graphs designed for object-oriented languages.

We introduced a new type of dependency for message servers within a rebe, called *intra-rebe dependency*. This dependency cannot be captured as *inter-procedure dependency* because the sequence of execution of the message servers is not deterministic. It is also different from *interference dependency* because concurrency does not exist within a rebe. This dependency captures the concurrency in Rebeca, and unlike *interference dependence* in multi-threaded concurrency, it is transitive.

7 Schedulability Analysis for Timed Actors

Besides functional analysis of systems, it is also necessary to make sure that systems preserve a certain level of quality-of-service. In standard Rebeca, each rebec by default assumes a “First-Come, First-Served (FCFS)” strategy to run the messages in its queue. This is, however, not optimal when there are other measures like priorities or response time that play a role in the QoS. To optimize QoS, we enable rebecs to specify local scheduling strategies, e.g., based on fixed priorities, earliest deadline first, or a combination of such policies. Rebecs may require certain customized scheduling strategies in order to meet their QoS requirements.

In real-time modeling, a task specification indicates its execution time besides generation of other tasks; further, tasks have deadlines before which they must be scheduled and executed. Analyzing schedulability of a real time system consists of checking whether all tasks are accomplished within their deadlines. In Rebeca, message servers can be considered tasks and deadlines are associated to messages, as sending a message in this setting generates a new task.

We employed automata theory [53,55] to provide a high-level framework for modular schedulability analysis of asynchronous reactive objects with local schedulers. In this framework, reactive objects are modeled abstractly using Timed Automata [8] so that analysis can be done in existing tools for example UPPAAL [75]. At this level of abstraction, a method definition may abstract from the real computation and replace it by passage of time (as explained later in the example in Fig. 9).

In modular analysis, we analyze rebecs individually. To analyze a rebec in isolation, we need to restrict the possible ways in which its methods may be called; to this end, we make use of behavioral interfaces for rebecs. A behavioral interface specifies at a high level and in the most general terms how a rebec may be used. As in modular verification [72], which is based on assume-guarantee reasoning, individually schedulable rebecs can be used in systems *compatible* with their behavioral interfaces. Schedulability of such systems is then guaranteed. Compatibility being subject to state space explosion can be efficiently tested [55].

7.1 Real-Time Classes and Rebecs in Timed Automata

Modeling Behavioral Interfaces. The abstract behavior of a rebec is specified in its behavioral interface. This interface consists of the messages the rebec may receive and send and provides an overview of the rebec behavior in a single automaton. A behavioral interface can also be seen as an abstraction (over-approximation) of the environments that can communicate with the rebec. A behavioral interface abstracts from specific method implementations, the queue in the rebec and the scheduling strategy.

In our example of sender-receiver, there are two interfaces (see Fig. 8). In this section, we use the shortened names ‘receive’ and ‘next’ instead of ‘receiveReq’ and ‘sendNextReq’, respectively. The Sender interface starts by outputting a ‘receive’ message; message communication is written in UPPAAL as ‘invoke[m][r]’

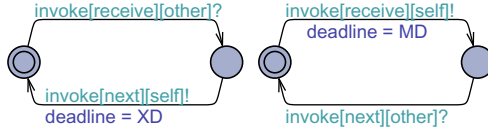


Fig. 8. Sender (left) and Receiver (right) Behavioral Interfaces

where ‘m’ denotes the message name and ‘r’ identifies the receiver. As dual to Sender, the Receiver interface inputs a ‘receive’ message at the beginning, which has a deadline ‘MD’; to specify a deadline for a message we use a global variable which will be used by the scheduler automaton (described below after classes). Considering the symmetric model in Fig. 4, a reactiveclass may implement both of these interfaces.

To formally define a behavioral interface, we assume a finite global set \mathcal{M} for method names. Sending and receiving messages are written as $m!$ and $m?$, respectively. A behavioral interface B providing a set of method names $M_B \subseteq \mathcal{M}$ is a deterministic timed automaton over alphabet Act^B such that Act^B is partitioned into two sets of actions:

- rebec outputs received by the environment: $Act_O^B = \{m? | m \in \mathcal{M} \wedge m \notin M_B\}$
- rebec inputs sent by the environment: $Act_I^B = \{m(d)! | m \in M_B \wedge d \in \mathbb{N}\}$

The integer d associated to input actions represents a deadline. A correct implementation of the rebec should be able to finish method m before d time units. The methods M_B must exist in the classes implementing the interface B . Other methods are sent by the rebec and should be handled by the environment.

Modeling Classes. One can define a class as a set of methods implementing a specific behavioral interface. A class R implementing the behavioral interface B is a set $\{(m_1, A_1), \dots, (m_n, A_n)\}$ of methods, where

- $M_R = \{m_1, \dots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_B \subseteq M_R$;
- for all i , $1 \leq i \leq n$, A_i is a timed automaton representing method m_i with the alphabet $Act_i = \{m! | m \in M_R\} \cup \{m(d)! | m \in \mathcal{M} \wedge d \in \mathbb{N}\} \cup \{t? | t \in \mathcal{T}\}$;

Classes have an *initial* method which is implicitly called upon initialization and is used for the system startup. Method automata only send messages or wait for replies while computations are abstracted into time delays. Receiving messages (and buffering them) is handled by the scheduler automata explained next. Sending a message $m \in M_R$ is called a self call. Self calls may or may not be assigned an explicit deadline. The self calls with no explicit deadline are called *delegation*. Delegation implies that the internal task (triggered by the self call) is in fact the continuation of the parent task; therefore, the delegated task inherits the (remaining) deadline of the task that triggers it. This mechanism is also handled by the scheduler.

Fig. 9 depicts the timed automata modeling the abstract behavior of the methods of a sender-receiver class. This class implements both the Sender and the

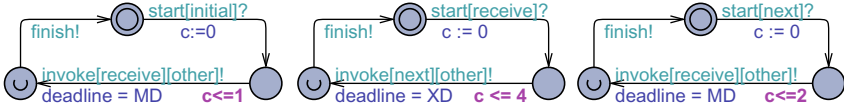


Fig. 9. Methods of a sender-receiver class

Receiver interfaces in Fig. 8, therefore it needs to implement both ‘receive’ and ‘next’ methods (in addition to the ‘initial’ method). To enable starting and stopping method executions, all these method automata start by a synchronization on the ‘start’ channel and end by ‘finish’ channel. In between, in this example, each method only sends a message while the rest of the method is abstracted away into a time delay.

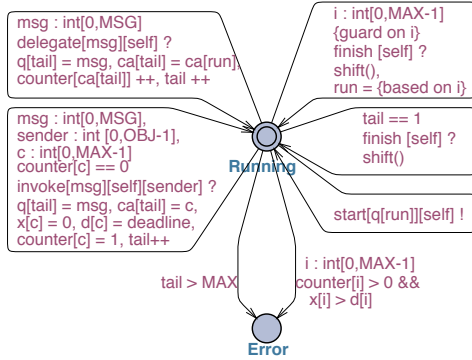


Fig. 10. A general scheduler automaton

Modeling Schedulers. Fig. 10 shows the general structure of a scheduler automaton. The only thing not specified in this picture is the scheduling strategy. This automaton has the functionalities described below.

Queue. The queue is modeled using arrays in UPPAAL. For a message stored in $q[i]$, the deadline is stored at $d[ca[i]]$ and the clock $x[ca[i]]$ keeps track of how long it has been in the queue. Delegation is modeled by reusing ca . The variable $counter[i]$ holds the number of tasks that use clock $x[i]$. A clock is free if its counter is zero. When delegation is used, the counter becomes greater than one.

Input-enabledness. In this general scheduler automaton, there is an edge (left down in the picture) that allows receiving (at any time) a message on the invoke channel (from any sender). To allow any message and sender, select expressions are used. The expression $msg : \mathbf{int}[0,MSG]$ nondeterministically selects a value between 0 and MSG for msg . This is equivalent to adding a transition for each value of msg . Similarly, any sender ($\mathbf{sender} : \mathbf{int}[0,OBJ-1]$) can be selected. The selected message is put at the tail of the queue ($q[tail] = msg$), a free clock

($\text{counter}[c] == 0$) is assigned to it ($\text{ca}[\text{tail}] = c$) and reset ($x[c] = 0$), and the deadline value is copied ($d[c] = \text{deadline}$).

A similar transition accepts messages on the delegate channel. In this case, the clock already assigned to the currently running task (parent task) is assigned to the internal task ($\text{ca}[\text{tail}] = \text{ca}[\text{run}]$). In a delegated task, no sender is specified (it is always `self`).

Context-switch is performed in two steps (without letting time pass). When a method is finished (synchronizing on `finish` channel), it is taken out of the queue (by `shift()`). If it is not the last in the queue, the next method to be executed should be chosen based on a specific scheduling strategy (by assigning the right value to `run`). For a *concrete* scheduler, the guard and update of `run` should be well defined. If `run` is always assigned 0 during context switch, the automaton serves as a First Come First Served (FCFS) scheduler. An Earliest Deadline First (EDF) scheduler can be encoded using a guard like:

```
i < tail && i != run &&
forall (m : int[0,MAX-1])
  ( (m == run) ||
    (x[ca[i]] - x[ca[m]] >= d[ca[i]] - d[ca[m]])
  )
```

and assigning $\text{run} = (i < \text{run}) ? i : i-1$ (because `i` is selected before shifting). The guard $x[a] - x[m] \geq d[a] - d[m]$ makes sure that the remaining deadline of `a`, i.e., $x[a] - d[a]$, is bigger than or equal to the remaining deadline of `m`. The rest ensures that an empty queue cell ($i < \text{tail}$) or the currently finished method (`run`) is not selected.

If the currently running method is the last in the queue, nothing needs to be selected (i.e., if $\text{tail} == 1$ we only need to `shift`). The second step in context-switch is to start the method selected by `run`. Having defined `start` as an urgent channel, the next method is immediately scheduled (if queue is not empty).

Error. The scheduler automaton moves to the Error state if a queue overflow occurs ($\text{tail} > \text{MAX}$) or a deadline is missed ($x[i] > d[i]$). The guard $\text{counter}[i] > 0$ checks whether the corresponding clock is currently in use, i.e., assigned to a message in the queue.

7.2 Modular Schedulability Analysis

An *rebec* is an instance of a class together with a scheduler automaton. To analyze a *rebec* in isolation, we need to restrict the possible ways in which the methods of this *rebec* could be called. Therefore, we only consider the incoming method calls specified in its behavioral interface. Receiving a message from another *rebec* (i.e., an input action in the behavioral interface) creates a new task (for handling that message) and adds it to the queue. The behavioral interface doesn't capture (internal tasks triggered by) `self` calls. In order to analyze the schedulability of a *rebec*, one needs to consider both the internal tasks and the tasks triggered by the (behavioral interface, which abstractly models the acceptable) environment.

We can generate the possible behaviors of a rebec by making a network of timed automata consisting of its method automata, behavioral interface automaton B and a concrete scheduler automaton. The inputs of B written as $m!$ will match with inputs in the scheduler written as $m?$ and the outputs of B written as $m?$ will match outputs of method automata written as $m!$.

An rebec is schedulable, i.e., all tasks finish within their deadlines, if and only if the scheduler cannot reach the Error location with a queue length of $\lceil d_{max}/b_{min} \rceil$, where d_{max} is the longest deadline for any method called on any transition of the automata (method automata or the input actions of the behavioral interface) and b_{min} is the shortest termination time of any of the method automata [53]. We can calculate the best case runtime for timed automata as shown by Courcoubetis and Yannakakis [28].

Once a rebec is verified to be schedulable with respect to its behavioral interface, it can be used as an off-the-shelf component. To ensure the schedulability of a system composed of individually schedulable rebecs, we need to make sure their real use is *compatible* with their expected use specified in the behavioral interfaces. The product of the behavioral interfaces, called B , shows the acceptable sequences of messages that may be communicated between the rebecs. Compatibility is defined as the inclusion of the visible traces of the system in the traces of B [55].

To avoid state-space explosion, we test compatibility. A trace is taken from B and turned into a test case by adding **Fail**, **Pass** and **Inconc** locations. Deviations from the trace either lead to inconclusive verdict **Inconc** (meaning that no conclusions can be drawn from this test) when the step is allowed in B , or otherwise lead to **Fail** (meaning that a counter-example to compatibility is found). The submission of a test case consists of having it synchronize with the system. This makes the system take the steps specified in the original trace. The **Fail** location is reachable if and only if the system is incompatible with B along this trace. This testing method is sound and complete [55].

7.3 Related Work

Schedulability has usually been analyzed for a whole system running on a single processor, whether at modeling [38,7] or programming level [27,69]. We address distributed systems where each rebec has a dedicated processor and scheduling policy. We propose a modular approach to schedulability analysis similar to the ideas of modular model checking [72]. The work in [40] is also applicable to distributed systems but is limited to rate monotonic analysis. Our analysis being based on automata can handle non-uniformly recurring tasks as in Task Automata [38]. In Task automata, however, a task is purely specified as computation times and therefore it cannot create sub-tasks.

RT-Synchronizers [93] are designed for declarative specification of timing constraints over groups of untimed actors. Therefore, they do not speak of schedulability of the actors themselves; in fact, a deadline associated to a message is for the time before it is executed and therefore cannot deal with the execution time of the task itself or sub-task generation.

In our approach, behavioral interfaces are key to modularity. A behavioral interface models the most general message arrival pattern for a rebec. In the literature, a model of the environment is usually the task generation scheme in a specific situation. However, a behavioral interface in our analysis covers all allowable scenarios of using the rebec, which in turn adds to the modularity of our approach; every use of the rebec foreseen in the interface is verified to be schedulable. Comparatively, for instance in TAXYS [27], this model of the environment can also be general enough to cover all uses of the program but it is used to analyze a complete program and is not used modularly.

In [12,54], an extension to our approach is applied to accommodate explicit *release* statements and *replies* of the Creol language. Creol [61,60] is a concurrent object-based language where the core of the language is similar to Rebeca and objects communicate only via asynchronous message passing. Asynchronous message passing in Creol is augmented with return values. Furthermore, Creol has explicit synchronization mechanisms, e.g., after sending a message, the caller may wait for a return value from the callee. A running method can decide to voluntarily *release* the control over processor, e.g., if the return values of a call are not yet available.

8 Extending Rebeca: Analyzing Self-adaptive Models

Software systems are steadily becoming larger, more heterogeneous and long-lived. Flexible and scalable approaches are required for developing today's complex and evolving software-intensive systems. Hard-coded mechanisms make tuning and adapting long-run systems complicated. A prosperous practice is to enable such systems to continually evolve and adapt to situations not anticipated at development time.

In order to obtain adaptation, a major concern is *Flexibility*. Recently, the use of policies has been recognized as a powerful mechanism to achieve flexibility in adaptive and autonomous systems. With policies, one can “*dynamically*” specify the requirements in terms of high level goals. A policy is a rule describing the conditions under which a specified subject must, may or may not perform an action on a specific object.

Since self-adaptive systems are often complex and have a great degree of autonomy, it is more difficult to ensure that they behave as intended. Hence, it is of great practical importance to provide rigorous mechanisms for checking their correctness. To this end, *model-driven approaches* and *formal methods* can play a key role.

PobSAM (Policy-based Self-Adaptive Model) [67] is a flexible formal model to develop, specify and verify self-adaptive systems. It uses Rebeca as the language for specifying the functional behavior of systems. In order to build self-adaptive models, PobSAM adds two layers of *views* and *managers* on top of the actor layer. Analysis methods based on Rebeca model checking tools are proposed to check behavioral correctness, consistency of policies, and safety in the adaptation phase. Theories for checking behavioral equivalence and substitutability of two

components are established. A mapping to Maude [82] is developed (not yet published) for more effective analysis of the models.

8.1 Motivating Example: Smart Home

As a motivating example, we describe a smart home based on the the example in [67]. In a home automation system, sensors are devices that provide a smart home with information about the physical properties of the environment. In addition, actuators are physical devices that can change the state of the world in response to these sensed data. The system processes the data gathered by the sensors, then it activates the actuators to alter the user environment according to the predefined set of policies. Smart homes can have different features, for example: (1) The lighting control can switch the lights on/off automatically, or adjust their intensity based on their placement in a room and according to the predefined policies. (2) Doors/Windows management controls windows and doors automatically. For instance, if windows have blinds, these should be rolled up and down automatically. (3) Heating control allows the inhabitants to adjust the house temperature to their preferred value. The heating control will adjust itself automatically in order to save energy.

The smart home system is required to adapt its behavior according to the changes in the environment. A typical system runs in normal, vacation and fire modes, and in each mode, it enforces different sets of policies to adapt to the current conditions. As some examples the policies defined for the lighting control module while the system runs in normal and fire modes can be as follows:

Defined Policies in the Normal Mode

- P1** Turn on the lights automatically when night begins.
- P2** Whenever someone enters an empty room, the light setting must be set to default.
- P3** When the room is reoccupied within $T1$ minutes after the last person has left the room, the last chosen light setting has to be reestablished.
- P4** The system must turn the lights off, when the room is unoccupied.

Defined policies in the Fire Mode

- P1** Turn on the emergency light.
- P2** Disconnect power outlets.
- P3** When the fire is extinguished, turn off the emergency light.

8.2 PobSAM Outline

A PobSAM model is the composition of three layers:

- The *actors layer* describes the functional behavior of the system and contains the computational entities. Rebeca is used to model these actors.

- The *managers layer* contains the autonomous managers. Managers are responsible for managing actors' behavior according to the predefined policies.
- The *view layer* is composed of a set of views that provide an abstraction of the actors' state for the managers. A view is a state variable, a function or a predicate applied to the state variables of actors.

The managers monitor the actors through views. Views provide managers with the required information about the actors. Each manager has a set of configurations containing adaptation policies and governing policies. The actors' behavior is directed by sending messages to them according to the governing policies. Adaptation policies are used for dynamic adaptation in response to the changing circumstances by switching between configurations.

In our example, actors are used at the functional level and model the sensors and actuators (e.g., the light actuator). The manager layer includes the policies in each configuration (e.g., the light controller would be a manager with the different policies for normal, fire and vacation configurations). The view layer acts as an abstract interface of the actors for the managers (e.g., a variable showing the intensity of each light and a variable showing the total intensity).

In this model, a new mode of operation, called the *adaptation mode*, is introduced to control the adaptation phase. Whenever an event which requires adaptation occurs, the relevant managers are informed. However, adaptation is not done immediately and the managers run in adaptation mode before switching to the next configuration. When the system reaches a safe state, the managers switch to the new configuration. This feature allows us to guide the adaptation process safely. There are two kinds of adaptation, called *loose adaptation* and *strict adaptation*. Under loose adaptation, the manager enforces old governing policies, whereas in strict adaptation, all events are ignored until the system passes the adaptation mode and reaches a safe state.

PobSAM has a formal foundation that employs an integration of algebraic formalisms and actor-based models. While the computational (functional) model of PobSAM is based on the actor-based semantics of Rebeca, the Configuration Algebra (CA) is proposed to specify the configurations of managers. A manager is formally defined as a tuple consisting of the set of possible configurations, the initial configuration, and the set of observable views for the manager. A configuration is defined as a set of governing policies and a set of adaptation policies.

A governing policy consists of a priority, an event, a condition (a Boolean term) and an action. Events are generated when the execution of a message server is completed, when a message is sent, when a new actor is created, and when a specific condition in the system becomes true. The action part of a governing policy is specified using an algebraic theory in which the primitive action is sending a message to an actor (rebec). Action terms may be guarded; complex actions are constructed by sequential or parallel composition or by a nondeterministic choice among multiple actions. Whenever a manager receives an event, it identifies all the governing policies that are activated by that event.

For each of the activated policies, if the policy condition evaluates to true, its action is triggered by sending a message to the relevant actors.

An adaptation policy is a prioritized rule that whenever triggered, drives the manager to the adaptation mode. The manager will switch to the new configuration after a safe state is reached. An adaptation policy consists of the priority of the policy, the triggering event, the condition of triggering the policy, the condition of applying the policy, the adaptation type (loose or strict), and the new configuration. Adaptation takes place in two phases. The adaptation policy implies that when the specified event occurs, and the triggering condition holds, if there is no other triggered adaptation policy with a higher priority, then the manager evolves to the strict or loose adaptation modes based on its type. When the condition of applying adaptation becomes true, the manager will perform adaptation and switch to the specified configuration.

8.3 Formal Analysis

We can perform different kinds of analysis on PobsAM models. In general, properties to be checked about an adaptive system can be categorized as adaptation properties, functional properties or a composition of both. Correctness properties of the functional layer (actors) of PobsAM models are application-specific. Correctness properties of the managers layer are related to the adaptation concerns (i.e., adaptation policies) or behavioral concerns (i.e., governing policies). Particularly, as policies direct the system behavior, it is required to understand and control the overall effect of governing policies on the system behavior. Governing policies often interact with each other and can cause undesirable effects. Hence, it is crucial to provide mechanisms to detect different kinds of policy conflicts. Furthermore, the correctness of the adaptation process of the PobsAM models, especially its stability, is an important property that needs to be verified.

In [68], we model PobsAM using Rebeca where actors and managers are modeled as rebecs, and views are modeled using global variables (as explained in Section 3.2 global variables are added to Rebeca for modeling system-level designs and can be used in a controlled way here, too). To enforce governing policies, a message server named `enforce` is considered for each manager rebec, which receives and handles events by interpreting the governing policies of the current configuration of the manager. While a manager is in normal or loose adaptation mode, it handles events by enforcing the triggered governing policies based on the priority of the policies. Governing policies are expressed as a set of rules in the body of `enforce`. The conditional part of a governing policy is defined as a guarded expression. The policy context is defined in terms of the global state variables associated with the view layer. Moreover, a new generic classification of the conflicts that may exist among governing policies is introduced, and LTL patterns are proposed to express each type of these conflicts. A number of correctness properties of the adaptation process are also introduced. We use the model checking tools of Rebeca to detect policy conflicts and check the correctness of the adaptation phase. In addition to model checking, an approach based on static analysis of adaptation policies is presented to check system stability: if

an adaptation by a manager leads to another adaptation, and this continues in a cycle then it causes an unstable state for the system which can be detected by a graph analysis technique.

Later, in [66], a behavioral equivalence theory is presented which helps in substitution of components and compositional reasoning. In dynamic environments such as the ubiquitous computing world, many systems must cope with variable resources, system faults and changing user priorities. In such environments, the system required to continue running with minimal human intervention, and the component assessment and integration process must be carried out automatically. Component assessment is identifying a component with the desired behavior that can replace another one. A possible solution to this problem relies on detecting the behavioral equivalence of components. Generally, we categorize behavioral equivalence of two components as context-independent or context-specific. Two components that are context-independent equivalent behave equivalently in any environment, whereas equivalence of two context-specific equivalent components depends on the environments in which they are running.

In [66], we present a context-independent behavioral equivalence theory to reason about managers, configurations, policies and policy actions. We develop semantic theories based on the notion of *splitting bisimulation* [11] and present sound and complete axiomatizations for this kind of bisimulation with respect to policy actions and governing policies. Furthermore, we introduce a new type of bisimilarity, called *prioritized splitting bisimulation*, to describe the behavioral equivalence of adaptation policies, configurations and managers. In [65], we develop an equational theory to analyze the context-specific behavioral equivalence of manager components based on a notion of behavioral equivalence, called state-based bisimulation. The view layer (i.e., the context) of the system is specified by a labeled state transition system. We extend our Configuration Algebra with new operators to consider the interaction of managers and the context and present the axioms of those operators. An important advantage of this equational theory is that it analyzes the behavioral equivalence of the manager layer using the view layer and independently from the actor layer.

8.4 Related Work

Dynamic adaptation is a very diverse area of research and different communities are concerned with this issue including autonomic computing, component-based systems, software architecture, coordination models, agent-based systems, etc. Structural adaptation has been given strong attention in the research community, and formal techniques have been extensively used to model and analyze dynamic structural adaptation (see [15]). *Structural adaptation* (or dynamic re-configuration) is usually modeled using graph-based approaches (e.g. [112,84]) or ADL-based approaches (e.g. [80,90]).

Behavioral adaptation focuses on modifying the functionalities of the computational entities. Formal modeling and verification of adaptive systems at behavioral level is a young research area [18] and only a few research groups have

already focused on this topic. As part of the RAPIDware project, Zhang et al. [123] proposed a model-driven approach for developing adaptive systems. In this approach, different contexts in which an adaptive program may run are specified by a formalism like temporal logic. The local properties of the program in each context are described formally. Then, a state-based model of the program in each context as well as the adaptation models for the adaptations of the program from one context to another are built. Different behavioral variants of a program are modeled as Petri Nets in [123]. Furthermore, they extend LTL with an “adapt” operator called A-LTL to specify adaptation requirements before, during and after adaptation [122] and introduce a model checking approach to verify the program formally. In another work [124], they propose a modular approach to verify adaptive programs.

Schneider et al. [100] present a method to describe adaptation behavior at an abstract level. After deriving transition systems from the system description, the system properties are verified using model checking techniques. In their later work [1], they propose a framework, MARS, for model-based development of adaptive embedded systems in which a model consists of a set of modules. A module may have different guarded configurations which are selected dependent on the current situation of the modules environment. The system is specified using Synchronous Adaptive Systems (SAS) [99] and is verified using theorem proving, model checking and specialized verification methods.

RAPIDware and MARS are on a different level of abstraction comparing to PobSAM. In these works, the system is described using a semantic-level state-based formalism while PobSAM uses high-level policies to control the system behavior and provide a high-level language to specify policies formally. Moreover, unlike PobSAM, configurations and the adaptation logic are fixed in RAPIDware and MARS. The ability to change configurations and the adaptation logic is vital to be able to model evolving adaptive systems. An act of adaptation in [122] results in a completely new program, but adaptation in PobSAM influences only the managers layer and the actors keeps running normally during adaptation. Thus, the adaptation semantics of PobSAM differs from that of A-LTL, however, both approaches consider safe adaptation.

A close area of research is coordination in which the interaction of objects can be controlled to achieve adaptation. While coordination models aim at decoupling interactions from computation and controlling interactions, PobSAM is concerned with controlling objects through controlling their behavior and decouples the *behavioral choices* and adaptation issues from the computational environment. ARC (Actor-Role-Coordinator)[94] and PAGODA (Policy And GOal based Distributed Architecture) [115,116] are two actor-based coordination models in which meta-actors control interactions of actors. ARC controls objects interactions by manipulating message delivery, for instance via rerouting and reordering messages. In PAGODA, each coordinator is provided with a set of policies to coordinate actors where a simple policy may reorder messages, serialize requests and maintain a history of events.

9 Conclusion and Future Work

Several actor languages have been developed [91,37,93,121,119], and some of these languages are supported by model checking or testing tools [39,101,77]. In this paper, we focused on the imperative actor-based modeling language Rebeca, which has been designed in 2001 with the goal of providing a language for modeling concurrent and distributed systems with formal verification support. Throughout the paper, the language Rebeca and the supporting tools and techniques for analyzing Rebeca models are explained. Here, we will summarize our main design decisions in the language, its extensions and analysis techniques. We will also address some of the ongoing and future work. This list is far from complete.

The Language and its Extensions. The general design strategy of Rebeca has been to keep the language a pure actor-based modeling language with no synchronous communication. Rebeca is designed based on an operational view of the actor model introduced in [6,5,81]. The kernel of the Rebeca language is kept simple and only supports asynchronous non-blocking message passing. This has allowed us to provide powerful analysis methods based on specialized abstraction and reduction techniques. On the other hand, we extended Rebeca for a few specific domains, e.g., for hardware-software co-design (Section 3.2) [63,92], and for globally asynchronous, locally synchronous systems (GALS) [108,109]. To be used as a hardware-software co-design language, we added *global variables* in order to model events and signals in a system design, and *wait statements* to model the situations when a process in the system is waiting for a specific event. Our reduction techniques are extended to cover this extension of Rebeca. The same extension of Rebeca is used in designing self-adaptive policy-based systems (explained in Section 8). In the extension for modeling GALS, we added a formal notion of components to the language. Components interact only by asynchronous messages, while within each component, the reactive objects may communicate by synchronous messages. This offers a general framework which integrates, in a formally consistent manner, both synchrony and asynchrony. Our formal verification approach is adjusted to reason about the open components. Certain properties are proven to be preserved when the model checked components are composed with other arbitrary components, and so, they can be plugged in a model relying on their behavior.

Analysis. Actor programming avoids the bugs inherent in shared-memory programming, but problems in incorrect sequential code within an actor, and problems in sequence of message passing still exist. These can cause deadlocks, race conditions, or bugs in the desired protocol. The biggest problem in analyzing actors is the growing number of sent messages (can be seen as events) that are not yet handled; this can quickly cause state-space explosion. In our tools, we allow the user to check the queue overflow condition and increase the size of the queue. In certain models the size of the queue is not bounded; in such situations, we need to have the option of running the system despite queue overflow.

To handle an overflow, different policies can be taken, e.g., to overwrite the old messages or purge the new ones.

Rebeca has FIFO queues for the pending messages, which pose stronger ordering constraints in comparison to message bags used in many other actor languages. This preserves the happens-before relation [74,77] while at the same time we consider all the possible interleavings for the execution of the rebecs. In our model, we have fewer message interleavings: for example between any pair of actors, the messages are processed in sending order, which is not the case for the models using bags. We consider atomic execution of methods, which is in line with the macro-step semantics of [5]. The combination of atomic execution and FIFO message queues causes even less message interleavings, for example, in the case where within a method we have more than one messages sent to the same rebec. These situations can be found by a simple static analysis of the code and if necessary be taken care of by a fine-grained execution of methods, or having a rebec in the middle that plays the role of a bag for the messages.

In analyzing Rebeca, we generally do not need to deal with the complications of programming languages, like complex data structures, or implementation details like managing the thread pools.

Future Work. The semantics and the established theories for Rebeca include dynamic creation of rebecs and dynamic topology, but the tools have to be extended to support these features of the language.

An ongoing work is a distributed implementation of the model checker. The BFS algorithm is especially suitable for parallel model checking [9]. This extension will distribute the state space across multiple computers, which will result in the ability to handle much bigger systems. In this work, we are investigating the applicability of call dependency graphs of Rebeca code which are similar to event diagrams of Clinger [26] but are derived using static analysis. Another approach to improve efficiency is using heuristics in model checking Rebeca. We are working on the application of best-first search algorithms using heuristics based on information from the message queues. Preliminary experimental results show the efficacy of the technique for some models [50].

A possible extension of Modere is to replace the rebec manager with a process/object manager for another language with a similar actor-based concurrency model. An ongoing work is integrating a Creol [62,60] interpreter with Modere. Creol is based on concurrent objects (similar to actors). Creol has fine-grained interleaving and assumes no order on executing messages from the message bags. The semantics of Creol is implemented in the rewrite engine of Maude and as a result, execution and simulation of Creol models are currently possible. To the best of our knowledge, there exists however no efficient model checking tool for Creol. We expect that the state-space reduction techniques already developed in Modere could also be applicable to Creol (possibly with some adjustments) due to the similar concurrency model.

Acknowledgement. We wish to thank all the present and past members of the Rebeca group for their enthusiasm and hard work. In particular, we thank

Hamideh Sabouri and Narges Khakpour for their help in writing the sections on slicing and self-adaptive models (respectively). Furthermore, we would like to thank Luca Aceto, Farhad Arbab and Mohammad Reza Mousavi for their comments on this paper. The work of the second author is supported by the EU FP7-231620 project called HATS.

References

1. Adler, R., Schaefer, I., Schuele, T., Vecchié, E.: From model-based design to formal verification of adaptive embedded systems. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 76–95. Springer, Heidelberg (2007)
2. Afra: a SystemC verifier, <http://ece.ut.ac.ir/FML/afra.htm>
3. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge (1990)
4. Agha, G.: The structure and semantics of actor languages. In: de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1990. LNCS, vol. 489, pp. 1–59. Springer, Heidelberg (1991)
5. Agha, G., Mason, I., Smith, S., Talcott, C.L.: A foundation for actor computation. *Journal of Functional Programming* 7, 1–72 (1997)
6. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: Towards a theory of actor computation. In: Cleaveland, R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 565–579. Springer, Heidelberg (1992)
7. Altisen, K., Gößler, G., Sifakis, J.: Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems* 23(1-2), 55–84 (2002)
8. Alur, R., Dill, D.: A theory of timed automata. *Theoretical Computer Science* 126, 183–235 (1994)
9. Barnat, J., Cerná, I.: Distributed breadth-first search ltl model checking. *Formal Methods in System Design* 29(2), 117–134 (2006)
10. Behjati, R., Sabouri, H., Razavi, N., Sirjani, M.: An effective approach for model checking systemc designs. In: Billington, J., Duan, Z., Koutny, M. (eds.) Proc. 8th International Conference on Application of Concurrency to System Design (ACSD 2008), pp. 56–61. IEEE (2008)
11. Bergstra, J.A., Middelburg, C.A.: Preferential choice and coordination conditions. *J. Log. Algebr. Program.* 70(2), 172–200 (2007)
12. de Boer, F., Chothia, T., Jaghoori, M.M.: Modular schedulability analysis of concurrent objects in Creol. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 212–227. Springer, Heidelberg (2010)
13. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric SPIN. *International Journal on Software Tools for Technology Transfer (STTT)* 4(1), 92–106 (2002)
14. Bošnački, D., Donaldson, A.F., Leuschel, M., Massart, T.: Efficient approximate verification of promela models via symmetry markers. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 300–315. Springer, Heidelberg (2007)
15. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proc. 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004, pp. 28–33 (2004)
16. Chang, P.H., Agha, G.: Supporting reconfigurable object distribution for customized Web applications. In: The 22nd Annual ACM Symposium on Applied Computing (SAC 2007), pp. 1286–1292 (2007)

17. Chang, P.H., Agha, G.: Towards context-aware web applications. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 239–252. Springer, Heidelberg (2007)
18. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
19. Cheong, E., Lee, E.A., Zhao, Y.: Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks. In: Proc. 3rd International Conference on Embedded Networked Sensor Systems, SenSys 2005, pp. 302–302 (2005)
20. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An openSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
21. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
22. Clarke, E.M.: The birth of model checking. In: Proc. Symposium on “25 Years of Model Checking”, Federated Logic Conference (FLOC 2006) affiliated with CAV 2006, pp. 1–26 (August 2006)
23. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)
24. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counter-example-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 752–794 (2003)
25. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design* 9(1/2), 77–104 (1996)
26. Clinger, W.D.: Foundations of actor semantics. Tech. rep., Cambridge, MA, USA (1981)
27. Closse, E., Poize, M., Pulou, J., Sifakis, J., Venter, P., Weil, D., Yovine, S.: TAXYS: A tool for the development and verification of real-time embedded systems. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 391–395. Springer, Heidelberg (2001)
28. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design* 1(4), 385–415 (1992)
29. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 481–496. Springer, Heidelberg (2005)
30. Donaldson, A.F., Miller, A.: Extending symmetry reduction techniques to a realistic model of computation. *Electronic Notes in Theoretical Computer Science* 185, 63–76 (2007)
31. Donaldson, A.F., Miller, A., Calder, M.: Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electr. Notes Theor. Comput. Sci.* 128(6), 161–177 (2005)
32. Donaldson, A.F., Miller, A., Calder, M.: Spin-to-Grape: A tool for analysing symmetry in Promela models. *Electronic Notes in Theoretical Computer Science* 139(1), 3–23 (2005)

33. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for dpll(t). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
34. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V., Robby, Wallentine, T.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 73–89. Springer, Heidelberg (2006)
35. Emerson, E.A.: Temporal and Modal Logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 996–1072. Elsevier Science Publishers, Amsterdam (1990)
36. Emerson, E.A., Sistla, A.: Symmetry and model checking. Formal Methods in System Design 9(1-2), 105–131 (1996)
37. Erlang Programming Language Homepage, <http://www.erlang.org>
38. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
39. Fredlund, L.Å., Svensson, H.: Mcerlang: a model checker for a distributed functional programming language. SIGPLAN Not 42(9), 125–136 (2007)
40. Garcia, J.J.G., Gutierrez, J.C.P., Harbour, M.G.: Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In: Proc. 12th Euromicro Conference on Real-Time Systems, pp. 15–24. IEEE (2000)
41. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991)
42. Groote, J.F., Mathijssen, A., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.: The formal specification language mcl2. In: Brinksma, E., Harel, D., Mader, A., Stevens, P., Wieringa, R. (eds.) MMOSS. Dagstuhl Seminar Proceedings, vol. 06351. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl (2006)
43. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to UPPAAL. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 46–59. Springer, Heidelberg (2004)
44. Hewitt, C.: Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT (April 1972)
45. Hewitt, C.: What Is Commitment? Physical, Organizational, and Social (Revised). In: Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., Matson, E. (eds.) COIN 2006. LNCS (LNAI), vol. 4386, pp. 293–307. Springer, Heidelberg (2007)
46. Hewitt, C.: Orgs for scalable, robust, privacy-friendly client cloud computing. IEEE Internet Computing 12(5), 96–99 (2008)
47. Hewitt, C.: Actscript(tm): Industrial strength integration of local and nonlocal concurrency for client-cloud computing. CoRR abs/0907.3330 (2009)
48. Hojjat, H., Sirjani, M., Mousavi, M., Groote, J.: Sarir: A Rebeca to mCRL2 translator (tool paper). In: Proc. 7th International Conference on Application of Concurrency to System Design (ACSD 2007) (July 2007)
49. Holzmann, G.J.: The model checker SPIN. Software Engineering 23(5), 279–295 (1997)
50. IceRose homepage - projects, <http://en.ru.is/icerose/applying-formal-methods/projects>

51. Ip, C.N., Dill, D.L.: Better verification through symmetry. *Formal Methods in System Design* 9(1-2), 41–75 (1996)
52. Ip, C.N.: *State Reduction Methods for Automatic Formal Verification*. Ph.D. thesis, Department of Computer Science, Stanford University (1996)
53. Jaghoori, M.M., de Boer, F.S., Chotia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming* 78(5), 402–416 (2009)
54. Jaghoori, M.M., Chotia, T.: Timed automata semantics for analyzing Creol. In: *Proc. Foundations of Coordination Languages and Software Architectures (FOCLASA 2010)*. EPTCS, vol. 30, pp. 108–122 (2010)
55. Jaghoori, M.M., Longuet, D., de Boer, F.S., Chotia, T.: Schedulability and compatibility of real time asynchronous objects. In: *Proc. 2008 Real-Time Systems Symposium (RTSS)*, Barcelona, pp. 70–79. IEEE Computer Society (2008)
56. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: the model-checking engine of Rebeca. In: Haddad, H. (ed.) *Proc. ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 23–27, pp. 1810–1815. ACM (2006)
57. Jaghoori, M.M., Sirjani, M., Mousavi, M., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. *Acta Informatica* 47, 33–66 (2010)
58. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Movaghar, A.: Efficient symmetry reduction for an actor-based model. In: Chakraborty, G. (ed.) *ICDCIT 2005*. LNCS, vol. 3816, pp. 494–507. Springer, Heidelberg (2005)
59. Jahania, M.: *Using SAT-Solvers to model check Rebeca for data-centric applications* - Master thesis, Sharif University of Technology (2008)
60. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling* 6(1), 35–58 (2007)
61. Johnsen, E.B., Owe, O., Arnestad, M.: Combining active and reactive behavior in concurrent objects. In: Langmyhr, D. (ed.) *Proc. of the Norwegian Informatics Conference (NIK 2003)*, pp. 193–204. Tapir Academic Publisher (November 2003)
62. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science* 365(1-2), 23–66 (2006)
63. Kakooe, M.R., Shojaei, H., Ghasemzadeh, H., Sirjani, M., Navabi, Z.: A new approach for design and verification of transaction level models. In: *International Symposium on Circuits and Systems (ISCAS 2007)*, pp. 3760–3763 (2007)
64. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: *PPPJ 2009: Proc. 7th International Conference on Principles and Practice of Programming in Java*, pp. 11–20. ACM, New York (2009)
65. Khakpour, N., Jalili, S., Sirjani, M., Goltz, U.: Context specific behavioral equivalence of policy-based self-adaptive systems. In: *Proc. 13th International Conference on Formal Engineering Methods (ICFEM 2011)* (to appear, 2011)
66. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: Formal modeling of evolving adaptive systems. In: *Science of Computer Programming - Special issue of FACS 2009* (2009) (accepted)
67. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: PobSAM: Policy-based managing of actors in self-adaptive systems. *Electr. Notes Theor. Comput. Sci.* 263, 129–143 (2010)
68. Khakpour, N., Khosravi, R., Sirjani, M., Jalili, S.: Formal analysis of policy-based self-adaptive systems. In: *Proc. 25nd Annual ACM Symposium on Applied Computing (SAC 2010)*, pp. 2536–2543 (2010)

69. Kloukinas, C., Yovine, S.: Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In: Proc. 15th Euromicro Conference on Real-Time Systems (ECRTS 2003), Portugal, pp. 287–294. IEEE CS (2003)
70. Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs. Ph.D. thesis, Universitat Passau, Fakultät für Mathematik und Informatik (April 2003)
71. Krinke, J.: Context sensitive slicing of concurrent programs. ACM SIGSOFT Software Engineering Notes, 178–187 (2003)
72. Kupferman, O., Vardi, M.Y., Wolper, P.: Module checking. Information and Computation 164(2), 322–344 (2001)
73. Lamport, L.: Composition: A way to make proofs harder. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 402–407. Springer, Heidelberg (1998)
74. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (1978)
75. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. Journal on Software Tools for Technology Transfer (STTT) 1(1-2), 134–152 (1997)
76. Lauterburg, S., Karmani, R., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: Rosenblum, D., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 308–322. Springer, Heidelberg (2010)
77. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Basset: a tool for systematic testing of actor programs. In: SIGSOFT FSE, pp. 363–364 (2010)
78. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers 12(3), 231–260 (2003)
79. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: Proc. of the International Symmetry Conference, Edinburgh, UK, pp. 71–85 (2007)
80. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proc. Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (1996)
81. Mason, I.A., Talcott, C.L.: Actor languages: Their syntax, semantics, translation, and equivalence. Theoretical Computer Science 220(2), 409–467 (1999)
82. Maude Homepage, <http://maude.cs.uiuc.edu>
83. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Science of Computer Programming 37(1–3), 279–309 (2000)
84. Metayer, D.L.: Describing software architecture styles using graph grammars. Software Engineering, IEEE Transactions on 24(7), 521–533 (1998)
85. Microsoft: Asynchronous agents library, [http://msdn.microsoft.com/en-us/library/dd492627\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd492627(VS.100).aspx)
86. Miller, A., Donaldson, A.F., Calder, M.: Symmetry in temporal logic model checking. ACM Comput. Surv. 38(3) (2006)
87. Mohapatra, D., Mall, R., Kumar, R.: An overview of slicing techniques for object-oriented programs. Informatica (Slovenia), 253–277 (2006)
88. NuSMV user manual, <http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html>
89. Open SystemC Initiative: IEEE 1666: SystemC Language Reference Manual (2005), www.systemc.org
90. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: International Conference on Software Engineering, pp. 177–186 (1998)

91. Ptolemy homepage, <http://ptolemy.berkeley.edu/ptolemyII>
92. Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A., Sirjani, M.: Sysfier: Actor-based formal verification of systemc. *ACM Trans. Embed. Comput. Syst.* 10, 19:1–19:35 (2011)
93. Ren, S., Agha, G.: RTsynchronizer: language support for real-time specifications in distributed systems. *ACM SIGPLAN Notices* 30(11), 50–59 (1995)
94. Ren, S., Yu, Y., Chen, N., Marth, K., Poirot, P.E., Shen, L.: Actors, roles and coordinators — A coordination model for open distributed and embedded systems. In: Ciancarini, P., Wiklicky, H. (eds.) *COORDINATION 2006*. LNCS, vol. 4038, pp. 247–265. Springer, Heidelberg (2006)
95. de Rover, W.P., Langmaack, H., Pnueli, A. (eds.): *COMPOS 1997*. LNCS, vol. 1536. Springer, Heidelberg (1998)
96. Sabouri, H., Sirjani, M.: Actor-based slicing techniques for efficient reduction of Rebeca models. *Sci. Comput. Program.* 75(10), 811–827 (2010)
97. Sabouri, H., Sirjani, M.: Slicing-based reductions for Rebeca. *Electr. Notes Theor. Comput. Sci.* 260, 209–224 (2010)
98. Scala Programming Language Homepage, <http://www.scala-lang.org>
99. Schaefer, I., Poetzsch-Heffter, A.: Using abstraction in modular verification of synchronous adaptive systems. In: Autexier, S., Merz, S., van der Torre, L.W.N., Wilhelm, R., Wolper, P. (eds.) *Trustworthy Software*. OASICS, vol. 3. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl (2006)
100. Schneider, K., Schuele, T., Trapp, M.: Verifying the adaptation behavior of embedded systems. In: *Proc. 2006 International Workshop on Self-Adaptation and Self-Managing Systems, SEAMS 2006*, pp. 16–22. ACM, New York (2006)
101. Sen, K., Agha, G.: Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
102. Sirjani, M., Movaghar, A.: An actor-based model for formal modelling of reactive systems: Rebeca. *Tech. Rep. CS-TR-80-01*, Tehran, Iran (2001)
103. Sirjani, M., Movaghar, A., Iravanchi, H., Jaghoori, M., Shali, A.: Model checking Rebeca by SMV. In: *Proc. Workshop on Automated Verification of Critical Systems (AVoCS 2003)*, Southampton, UK, pp. 233–236 (April 2003)
104. Sirjani, M., Movaghar, A., Mousavi, M.: Compositional verification of an object-based reactive system. In: *Proc. Workshop on Automated Verification of Critical Systems (AVoCS 2001)*, Oxford, UK, pp. 114–118 (April 2001)
105. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Model checking, automated abstraction, and compositional verification of Rebeca models. *Journal of Universal Computer Science* 11(6), 1054–1082 (2005)
106. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Modeling and verification of reactive systems using Rebeca. *Fundamenta Informatica* 63(4), 385–410 (2004)
107. Sirjani, M., Shali, A., Jaghoori, M., Iravanchi, H., Movaghar, A.: A front-end tool for automated abstraction and modular verification of actor-based models. In: *Proceedings of Fourth International Conference on Application of Concurrency to System Design (ACSD 2004)*, pp. 145–148. IEEE Computer Society (2004)
108. Sirjani, M., de Boer, F.S., Movaghar, A.: Modular verification of a component-based actor language. *Journal of Universal Computer Science* 11(10), 1695–1717 (2005)

109. Sirjani, M., de Boer, F.S., Movaghar, A., Shali, A.: Extended Rebeca: A component-based actor language with synchronous message passing. In: Proceedings of Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), pp. 212–221. IEEE Computer Society (2005)
110. Sirjani, M., Movaghar, A., Iravanchi, H., Jaghoori, M.M., Shali, A.: Model checking in Rebeca. In: Arabnia, H.R., Mun, Y. (eds.) Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, vol. 4, pp. 1819–1822. CSREA Press (2003)
111. Spin: Spin User Manual, <http://spinroot.com/spin/Man/Manual.html>
112. Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 179–193. Springer, Heidelberg (2000)
113. Talcott, C.L.: Composable semantic models for actor theories. *Higher-Order and Symbolic Computation* 11(3), 281–343 (1998)
114. Talcott, C.L.: Actor theories in rewriting logic. *Theoretical Computer Science* 285(2), 441–485 (2002)
115. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. *Electr. Notes Theor. Comput. Sci.* 150, 143–157 (2006)
116. Talcott, C.L.: Policy-based coordination in PAGODA: A case study. *Electr. Notes Theor. Comput. Sci.* 181, 97–112 (2007)
117. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
118. Vardi, M.Y.: Branching vs. linear time: Final showdown. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001)
119. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices* 36(12), 20–34 (2001)
120. Weiser, M.: Program slicing. In: Proc. 5th International Conference on Software Engineering, pp. 439–449 (1981)
121. Yonezawa, A.: ABCL: An Object-Oriented Concurrent System. Series in Computer Systems. MIT Press (1990)
122. Zhang, J., Cheng, B.H.C.: Specifying adaptation semantics. *ACM SIGSOFT Software Engineering Notes* 30(4), 1–7 (2005)
123. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proc. 28th International Conference on Software Engineering, ICSE 2006, pp. 371–380. ACM, New York (2006)
124. Zhang, J., Goldsby, H.J., Cheng, B.H.: Modular verification of dynamically adaptive systems. In: Proc. 8th ACM International Conference on Aspect-Oriented Software Development, AOSD 2009, pp. 161–172. ACM, New York (2009)

Mathematical Models of Object-Based Distributed Systems

Carlos Henrique C. Duarte*

BNDES, Av. República do Chile 100, Rio de Janeiro, RJ, 20001-970, Brazil
carlos.duarte@computer.org
<http://chcduarte.webs.com>

Abstract. We propose an alternative characterisation of object-based distributed systems in terms of algebraic structures and topological spaces. Some examples are given in order to attest the adequacy of this approach to the subject. We also illustrate a method of transference of results from these mathematical theories that can further contribute to the advancement of distributed systems theory.

Keywords: Algebraic Structures, Topological Spaces, Distributed Systems, Software Development.

1 Introduction

Distributed systems are hard to design and understand because we lack intuition for them [24]. Although this was pointed out almost two decades ago, it appears that the theory and practice of distributed systems development has not evolved sufficiently since then so as to completely unravel the inherent complexity of the notion of distribution. On the contrary, diverse formalisms and technologies proliferate, usually adding more complexity to this problem.

A plausible and frequently adopted approach for obtaining a better intuition concerning distributed systems consists in developing an abstract and faithful representation of such systems, notably by adopting a class of mathematical structures that allows us to represent and analyse any property of interest. Here, such mathematical structures are called distributed system *formal models*.

It has been a longstanding tradition in Computer Science, and more generally in Logic, to adopt algebraic structures as the underlying mathematical entities against which the satisfiability of logically formulated properties is inspected [26]. The formal models usually adopted in distributed systems development are not different: (fair) transition systems [21], I/O automata [17], edge reversing graphs [5], event structures [19] and many others are defined as algebraic structures.

By no means we need to restrict ourselves to the algebraic character of mathematical structures in the study of distributed systems. In this paper, in particular, our aim

* The definitions and results in this paper can be regarded as an attempt to obtain more general axiomatic theories than those developed by Carolyn Talcott and her colleagues over the past two decades on the theory of asynchronous object-based distributed systems.

is precisely to propose a novel characterisation of distributed systems in terms of a diverse mathematical theory: topological spaces. In fact, this approach is not entirely new: Alpern and Schneider provided a topological characterisation of safety and liveness properties of concurrent and distributed systems in [3], while Herlihy, Shavit, Saks and Zaharoglou gave in [14,23] a topological characterisation of the class of decision problems that can be solved using asynchronous wait-free deterministic distributed shared memory computation. Here, our purpose is to widen this approach in order to cover not only distributed computing but also distributed systems development in general, including the specification and verification of their structural and behavioural properties.

The proposed characterisation of distributed systems yields a method of transference of results from the underlying mathematical theories. Frequently adopted by mathematicians, such method consists in a scheme to transpose verified results concerning a well established theory to another one which is still in development. In this paper, we also illustrate this method, which we believe can further contribute to the advancement of distributed systems theory.

We regard our algebraic and topological characterisation of distributed systems to be the main original contribution reported in this paper. The corresponding theoretical results are important towards establishing a framework for the analysis and simulation of distributed system behaviours, facilitating their comprehension. They also establish a general formal foundation for the definition of logical systems devoted to the compositional development of object-based distributed systems.

The remainder of the paper is organised as follows: Section 2 introduces the relevant distributed system notions; Section 3 presents the underlying algebraic structures that are used throughout the paper; while Section 4 develops a topological characterisation and analysis of distributed systems. We conclude the paper commenting on related and future work.

2 Distributed System Notions

Distributed systems can be identified in many different contexts. A computer network of a corporation can be regarded as a distributed system. Software applications providing support to electronic commerce, distance education and electronic government, through widely distributed computer networks such as the Internet, can also be considered as examples of this family of systems. Moreover, postal systems in which the dispatch, processing and delivery of letters is manually performed are inherently distributed.

It is very hard to propose an exact definition of distributed systems considering that our aim is to capture with such a definition the aforementioned examples. Consequently, we will consider sufficient throughout this text to say that a *distributed system* is a set of at least loosely coupled autonomous objects potentially situated at distinct localities.

Here, the term *object* corresponds to an abstraction, which can represent distinct types of entities, such as humans, intelligent agents, software components or processing units. The assumption that objects are at least *loosely coupled* captures, on the one hand, the intuition that completely disconnected sets of objects define in fact separated (sub-)systems, and, on the other hand, that fully coupled objects cannot present any autonomous behaviour.

The term *location* denotes a reference to the physical or virtual position of each object and is perhaps what fundamentally distinguishes our definition from others available in the literature. We find it convenient to presume that objects are situated (even if we frequently forget this), since, whenever this is identified as a requirement in a particular development, it is a sufficient condition to characterise the system as *inherently distributed*, in opposition to systems which are made distributed due to a decision adopting a distribution technology.

With the notions defined above, it is already possible to express many distributed system *conceptual models*, capturing distinct sets of choices concerning the modes of object creation, configuration, interaction and failure that regulate the structure and behaviour of a family of distributed systems. In the present paper, we recurrently rely on examples of dynamically configured asynchronous message passing and statically configured distributed shared memory systems.

We adopt the Actor Model [1] as a reference to message passing systems. Actors are independent units of interaction and computation. They interact solely via asynchronous point-to-point message passing. The delivery of messages is guaranteed and, as a result of consuming a message, an actor may change its local state, create finitely many objects and dispatch a finite number of messages to the actors which became known at creation time or through message consumption.

Concerning distributed shared memory systems, we adopt Unity [6] as a reference model. Objects in Unity are defined in terms of memories denoted by variables and also by multiple assignments. They interact among themselves and with their environment by reading and writing on shared memories, possibly synchronising on such occurrences. Unity adopts a weak fairness assumption requiring that continuously enabled assignments eventually occur. It is only due to such occurrences that variables defining object states can change.

Conceptual models such as Actors and Unity are formulated to enable the specification and verification of distributed systems. We use here the financial systems domain to contrast the respective modelling approaches. Two types of objects are postulated to exist: persons, active objects capable of receiving and disbursing amounts of money, and accounts, passive objects over which credit and debt operations can be performed. Typical relationships between these objects are those of deposit and withdrawal. We also allow persons and accounts to exchange money directly among themselves.

Persons and accounts are seen here as object types of the same kind. We consider that they are both money repositories and take advantage of this consideration to propose a unique specification for their kind. We present in Fig. 1 their specification according to the Actor Model and in Fig. 2 according to Unity. We should point out that the *get* and *put* actions in these specifications should be respectively read as a reception and a disbursement of money whenever repositories are seen as persons. The same rationale applies to accounts, credits and debts.

We use a temporal logic language in these specifications [10], instead of the usual declarative ones [1,6], to specify local object behaviours. The conditions in each specification are usually called *local constraints*, since they restrict the set of life cycles admissible for each object. For instance, (i) if we get some amount of money from a repository, this amount will be deduced from the respective balance in a moment strictly

Actor AMONEYREP
data types Addr, Float
attributes *bal* : Float
messages *get?*(Addr, Float), *get!*(Addr, Float), *put?*(Addr, Float), *put!*(Addr, Float)
axioms $m, n : \text{Addr}; k, x, y : \text{Float}$
 $(\text{get? } n \ x) \wedge \text{bal} = k \rightarrow (\neg((\text{get? } m \ y) \vee (\text{put? } m \ y))) \hat{\mathbf{U}}(\text{send } n \ (\text{get! self } x) \wedge \text{bal} = k - x)$ (1.1)
 $(\text{put? } n \ x) \wedge \text{bal} = k \rightarrow (\neg((\text{get? } m \ y) \vee (\text{put? } m \ y))) \hat{\mathbf{U}}(\text{send } n \ (\text{put! self } x) \wedge \text{bal} = k + x)$ (1.2)
 $\text{get? } n \ x \rightarrow \text{bal} \geq x$ (1.3)

Fig. 1. Actor-based specification of money repositories

Process UMoneyRep
data types Float
memories *bal* : Float
actions *get*(Float), *put*(Float)
axioms $k, x, y : \text{Float}$
 $\text{get}(x) \wedge \text{bal} = k \rightarrow (\neg(\text{get}(y) \vee \text{put}(y))) \hat{\mathbf{U}}(\text{bal} = k - x)$ (2.1)
 $\text{put}(x) \wedge \text{bal} = k \rightarrow (\neg(\text{get}(y) \vee \text{put}(y))) \hat{\mathbf{U}}(\text{bal} = k + x)$ (2.2)
 $\text{get}(x) \rightarrow \text{bal} \geq x$ (2.3)

Fig. 2. Unity-based specification of money repositories

in the future¹ and no other event is allowed to happen until then (1.1 and 2.1); (ii) this is only allowed to happen if the current balance is greater or equal to the required amount (1.3 and 2.3).

The differences in modelling style are apparent in our specifications: whereas actors interact by receiving, consuming and sending back asynchronous messages from/to their clients, Unity processes are synchronous, in the sense that events and variable changes are perceived to happen simultaneously by the participant objects. We should stress that, even using a particular specification formalism, the respective objects comply with the semantics of the corresponding conceptual models. For instance, a withdrawal represented in the standard Unity notation by a guarded assignment of a value x to an account balance shared variable is represented in specification UMoneyRep by the action identifier $\text{get}(x)$ and two axioms are used to express its pre and post conditions: axiom (2.3) defines the assignment guard while axiom (2.1) defines its results.

In order to address the configuration and correlation of local object behaviours, coordination specifications are adopted. Instead of proposing textual specifications for the aforementioned types of distributed objects in particular operation contexts, we perform this task diagrammatically: Fig. 3 details a possible message-based scenario for the behaviour of a financial system, whereas Fig. 4 uses a shared-memory state-based notation. These specifications are formulated using UML [20].

¹ It is important to mention that the standard semantics of these models also prevent two local events of the same object to happen concurrently.

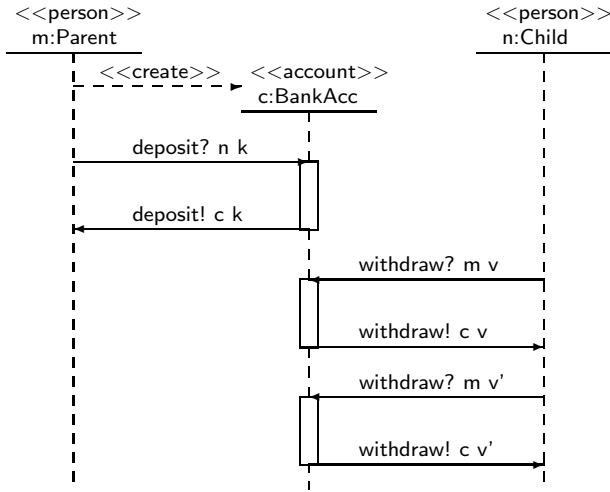


Fig. 3. Message-based scenario for a financial system behaviour

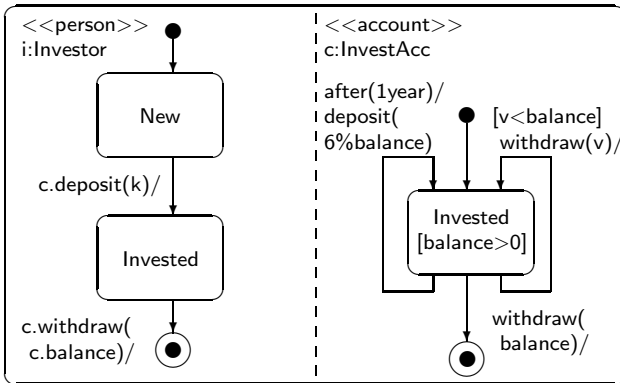


Fig. 4. Shared-memory state-based specification for financial system behaviours

3 Algebra

3.1 Time Frames

A central issue in distributed systems formal modelling consists in the identification of the structure and role of time, since different families of distributed systems appear to require distinct time structures in order to express the relatively autonomous behaviour of their constituent objects. In view of this diversity, we define here time structures in a very general manner and study afterwards how the usually adopted time frames can be derived. The role of time in distributed systems modelling is discussed in Section 3.2.

Definition 1 (Time Frame). A *time frame* is a(n algebraic) structure $\langle \langle \mathbf{T}, \leq \rangle, L \rangle$:

- \mathbf{T} is a non-empty set defining a *time domain*;
- $\leq \subseteq \mathbf{T} \times \mathbf{T}$ is a(n order) relation, with $\langle \mathbf{T}, \leq \rangle$ satisfying the following axioms:
 - (R1) Reflexivity: For every $x \in \mathbf{T}$, $x \leq x$;
 - (R2) Anty-symmetry: For every $\{x, y\} \subseteq \mathbf{T}$, $x \leq y \wedge y \leq x \rightarrow x = y$;
 - (R3) Transitivity: For every $\{x, y, z\} \subseteq \mathbf{T}$ $x \leq y \wedge y \leq z \rightarrow x \leq z$; (transitivity)
- L is a set of *linear time flows*: every $\lambda \in L$ is a function with $\text{dom } \lambda \in \mathcal{P}_+(\mathbf{T})$ and $\text{cod } \lambda \in \mathcal{S}_+(\mathbf{L})^2$, for some fixed $\langle \mathbf{L}, \preceq \rangle^3$ satisfying (R1-3) and
 - (LIN) Linearity: For every $\{x, y, z\} \subseteq \mathbf{L}$, $x \preceq y \rightarrow z \preceq x \vee y \preceq z$; and each λ satisfying the following axioms:
 - (OO) Injectivity: For every $\{x, y\} \subseteq \text{dom } \lambda$ $\lambda(x) = \lambda(y) \rightarrow x = y$;
 - (OT) Surjectivity: For every $y \in \text{cod } \lambda$, $\exists x \cdot x \in \text{dom } \lambda \wedge \lambda(x) = y$;
 - (MO) Monotonicity: For every $\{x, y\} \subseteq \text{dom } \lambda$, $x \leq y \rightarrow \lambda(x) \preceq \lambda(y)$.

The discrete time frames usually adopted in concurrent and distributed systems development can be classified according to their linear or branching nature. Branching time has been considered a convenient framework not only for the implementation of model checking tools but also for the specification of one kind of *enabledness property* which expresses what may be happening at the current time instant [10]. On the other hand, linear time is the mainstream assumption, due to its simplicity, but nevertheless allows one to express enabledness as the satisfaction of sufficient pre-conditions for a specific imminent occurrence [15].

Definition 1 easily captures linear or branching time frames. In particular, the assumption of linearity is captured whenever L is postulated to be a singleton. The fact that \mathbf{L} and \mathbf{T} are left unspecified in such kind of postulate makes it possible to express discrete or dense time extra assumptions. We do not, however, require that $\langle \mathbf{T}, \leq \rangle$ obeys only such restrictions taking into account inherently distributed physical systems for which the idealised time frame is imaginary.

In fact, the adoption of linearisation functions in L can be viewed precisely as a mechanism to map higher-dimensional time structures into the forth dimension of space-time, which corresponds to our standard linear intuition concerning time flows. Incidentally, we capture the existence of *initial time instants* in all time flows by postulating that each $\lambda \in L$ is bounded below.

3.2 Distributed System Formal Models

The role of time in distributed systems development consists precisely in allowing the association of autonomous objects to corresponding behaviours and placements. The following definition captures this intuition:

Definition 2 (Object Structure). An *object structure* is a 5-tuple $\langle \mathcal{T}, \mathcal{V}, \mathcal{S}, B, P \rangle$ defined in terms of the disjoint non-empty sets \mathbf{T} , \mathbf{Ev} and \mathbf{Loc} , where:

² We denote by $\mathcal{P}(X)$ the set of all subsets of X and by $\mathcal{S}(X)$ the set of all (open and closed) intervals defined in terms of X . Moreover, whenever we use the symbol $+$ as a subscript, the elements of these sets are non-empty.

³ By abuse of notation, we put $\text{dom } L \stackrel{\text{def}}{=} \mathbf{T}$ and $\text{cod } L \stackrel{\text{def}}{=} \mathbf{L}$.

- $\mathcal{T} = \langle \langle \mathbf{T}, \leq \rangle, L \rangle$ is a time frame;
- \mathcal{V} is an *event structure*, defined in terms of an *event domain* \mathbf{Ev} ;
- \mathcal{S} is a *location structure*, defined in terms of a *location domain* \mathbf{Loc} ;
- $B: \mathbf{Ev} \rightarrow \mathcal{P}(\mathcal{I}_+(\text{cod } L))$ defines a *behaviour*⁴;
- $P: \mathbf{Loc} \rightarrow \mathcal{P}(\mathcal{I}_+(\text{cod } L))$ defines a *placement*;

provided that the following axiom is satisfied:

(MAX) Maximality: For every $e \in \mathbf{Ev}$ and $s \in \mathbf{Loc}$, $((\cap B(e)) = \{\}) \wedge ((\cap P(s)) = \{\})$.

Structures as defined above are said to be partial order based in [22], since they rely on the definition of underlying partial-order relations as their time frames, \mathcal{T} . The time flows derived from these frames, L , are used in the assignment of a time dependent interpretation to events and locations, through B and P respectively. We leave the event and location structures \mathcal{V} and \mathcal{S} partially unspecified, since they are quite dependent upon the application domain at hand: events may carry values and locations may have hierarchical structure, for instance to represent realistic patterns of interaction and nested geographic regions respectively.

Concerning our financial information system specifications, the events that populate \mathbf{Ev} are represented by *get* and *put* (along with their variations formed by the use of the suffixes *?* and *!*). Therein, we could have postulated that objects are explicitly located. In that case, the values populating \mathbf{Loc} would define a domain on their own, specified according to an enumeration such as the following one:

$$\mathbf{Loc} \stackrel{\text{def}}{=} \text{HOME} \mid \text{ATM} \mid \text{AGENCY}$$

In this context, it makes sense to talk about a new requirement concerning transaction costs depending on the client location. For instance, transactions performed at an agency would be ten times more expensive than the others. In order to treat this requirement in our specifications, not only would we have to define \mathbf{Loc} as above, but we would also have to add a new parameter to each event and modify all the axioms to deduce the cost from the account balance whenever performing any transaction. Note that systems with location dependent requirements are regarded as inherently distributed.

Our example can be further extended so that we can illustrate the definition above. Suppose now that the aforementioned client is in fact a bank employer with a daily regular behaviour pattern: every day, he (i) leaves home early in the morning and comes back at night; (ii) stays in his agency during the whole day, and (iii) withdraws some money before working and deposits the remaining amount after work, both using an ATM. Considering the previous definitions, the following could be a possible object structure for this behaviour:

$$\begin{aligned} P(\text{HOME}) &= \{[0', 8'], [20', 24']\} & B(\text{withdraw}) &= \{[8'30'']\} \\ P(\text{AGENCY}) &= \{[9', 19']\} & B(\text{deposit}) &= \{[19'30'']\} \\ P(\text{ATM}) &= \{[8'30''], [19'30'']\} \end{aligned}$$

This example shows that, even if we forget locations, our previous definition is far more general than those found in the literature. For example, events may be durative. This is

⁴ Note that, since $\text{cod } L$ is totally ordered, an interval S of $\text{cod } L$ is a subset of $\text{cod } L$ such that, whenever $\{x, y\} \subseteq S$, $\forall z \in \text{cod } L \cdot x \leq z \leq y \rightarrow z \in S$.

accomplished by requiring that each event be assigned by B to the set of time intervals during which the event occurs in a time flow. The usually adopted instantaneous events [10] can be obtained by postulating that each occurrence is represented by a singleton, something captured by an axiom such as (INS) below. The local structures proposed in [22] are obtained by additionally requiring each object structure to obey sequentiality (SEQ) in a linear time structure. If we further require discreteness, as in [7], this can be captured by the assumption of yet another axiom, (DISC). Concerning localities, (XOR) may be postulated to ensure that two of them are never occupied at the same instant by a single object:

- (INS)** Instantaneity: For every $e \in \mathbf{Ev}$ and $x \in \mathcal{P}(\mathcal{I}_+(\text{cod } L))$,
 $x \in B(e) \rightarrow (\forall y, z \cdot y \in x \wedge z \in x \rightarrow y = z)$;
- (SEQ)** Sequentiality: For every distinct $\{e_1, e_2\} \subseteq \mathbf{Ev}$,
 $(\cup B(e_1)) \cap (\cup B(e_2)) = \{\}$;
- (DISC)** Discreteness: For every $\{x, y\} \subseteq \mathbf{T}$,
 $x \leq y \rightarrow \exists z \cdot (x \leq z \wedge \exists u \cdot x \leq u \wedge u \leq z) \wedge \exists w \cdot (w \leq y \wedge \exists u \cdot u \leq y \wedge w \leq u)$;
- (XOR)** Orthogonality: For every distinct $\{s_1, s_2\} \subseteq \mathbf{Loc}$,
 $(\cup P(s_1)) \cap (\cup P(s_2)) = \{\}$;

Given two object structures, we could have naively proceeded to attempt to define diverse modes of interaction. For instance, we could have stated that two objects synchronise at an event occurrence if the event belongs to their structures and its interpretations according to B in each of these structures have a time instant in common. However, this definition would not be sufficiently general to encompass the postulation of durative events. Worse, the involved structures could have time frames with distinct order-theoretic characters. In fact, such diversity reflects the nonexistence of universally valid time frames, a consequence of the acceptance of the theory of relativity governing inherently distributed systems.

Considering this rationale, the definition of distributed system (formal) models must be relational, in view of the necessity to establish correspondences between the time frames of each pair of objects, as well as the boundary of each (sub-)system in relation to its environment:

Definition 3 (Distributed System Model). A *distributed system model* is a 4-tuple $\langle \mathcal{O}, E, I, C \rangle$, where:

- \mathcal{O} is an *object universe*, a countable set of object structures;
- $E : \mathcal{O} \rightarrow \mathbf{Bool}$ is an *environment* identification function;
- $I : \mathcal{O} \rightarrow \mathbf{Bool}$ is an *internal object* identification function;
- $C : \mathcal{O} \rightarrow \mathcal{O}$ is a partial map which induces a family of *time correlation* functions: for each $o = \langle \mathcal{T}, \mathcal{V}, \mathcal{S}, B, P \rangle$, $o \in \text{dom } C$, such that, if there is $o' = \langle \mathcal{T}', \mathcal{V}', \mathcal{S}', B', P' \rangle$ with $C(o) = o'$, then there are also $\kappa_o : \mathbf{T} \rightarrow \mathbf{T}'$ and $\kappa_{o'} : \mathbf{T}' \rightarrow \mathbf{T}$ both obeying (MO).

such that the following axiom is satisfied:

- (SEP)** Separation: For every $o \in \mathcal{O}$, $E(o) \rightarrow \neg I(o)$.

The structural interface of a distributed system model – \mathcal{O} , E and I – is rather conventional and attempts to generalise the structures proposed in [2,25]: \mathcal{O} captures an

universe of distributed objects, E identifies the objects which belong to the *environment* (those *external* to the modelled system(s)), whereas I spots the *internals*, objects which cannot directly interact with the environment. Such interfaces will play a central role in the definition of model composition below.

It is important to point out that, since we do not make any assumption concerning (the existence of) real physical locations of distributed objects (if we had, we could also have proposed a metric establishing the distance between distributed objects), we cannot presume a fixed relationship between their respective time frames. In particular, since objects may have a strictly logical character, they are not obliged to follow relativistic correlations between time and space. Of course, in most cases object locations are given in terms of spatial coordinates and, in these cases, the respective structures will have to respect further spatio-temporal constraints.

The relational character of C is a consequence of the assumptions above. This map is important, however, not only to correlate object structure time frames but also to avoid violations in temporal causality chains (time travel), something prevented by the order preserving requisite (monotonicity) posed on this relationship.

3.3 Interaction and Composition

We are now able to formalise the distinction between synchronous and asynchronous modes of interaction, in the sense of [13]. Provided an object structure θ , we denote by \mathbf{Ev}_θ the set of events of the underlying event structure \mathcal{V}_θ . Given $E \subseteq \mathbf{Ev}_\theta$, we say that an *occurrence* D of the events in E is *totally synchronous* if D belongs to the interpretation of each event in E , that is $\forall e \cdot e \in E \rightarrow D \in B(e)$. This mode of interaction is unrealistic and used only in idealised models of distributed systems. Therefore, *partial synchrony* is usually adopted instead, which is defined by the existence of a common time point t in the given occurrences $D \in B(e)$ of each event e , that is $\exists t \cdot \forall e \cdot e \in E \wedge D \in B(e) \rightarrow t \in D$. Clearly, these definitions concern only to intra-object interaction, but they can be extended to inter-object interaction by considering that the involved objects all belong to the same distributed system model and are time correlated. Consequently, we say that objects interact in (*partially*) *synchronous* mode if (partial) event synchronisation is adopted as the sole mode of interaction.

The definition of synchrony above provides a sufficient characterisation of this notion but not a necessary one. A sufficient and necessary characterisation is obtained by asserting that objects rely on an upper bound on interaction delays, clock drifts or relative object speeds [13]. The usual way of guaranteeing this interaction pattern is to require that objects communicate using a blocking scheme. Objects rely on this kind of synchronisation information by referring to facilities provided by their (operational) environment, whose existence is presumed or defined in linguistic terms during the system development. In Section 3.4, we address the specification of such facilities.

The messages exchanged between persons and accounts illustrate point-to-point asynchronous interaction. In particular, the semantics of these events can be defined in terms of time intervals that capture the periods of time in which messages remain in transit. On the other hand, *deposit* and *withdraw* actions performed simultaneously by investors and their accounts are instances of synchronous interaction.

The semantics of interaction outlined above captures the view that only time correlated objects of the same model are able to interact. Now we wish to generalise this view to take into account a design method in which distinct distributed systems are modelled by isolated structures and are composed afterwards. The composition of distributed system models is defined below:

Definition 4 (Composability). Two distributed system models $\mathcal{M}_i = \langle \mathcal{O}_i, E_i, I_i, C_i \rangle$, $i \in \{0, 1\}$, are *composable* if and only if, given that $\mathcal{O} = \mathcal{O}_0 \cup \mathcal{O}_1$, the following axiom is satisfied:

(DISJ) Disjointedness: For every $o \in \mathcal{O}$, $\neg(I_0 \uparrow_{\mathcal{O}}(o) \wedge I_1 \uparrow_{\mathcal{O}}(o))$ ⁵;

Definition 5 (Model Composition). A *distributed system model* $\mathcal{M} = \langle \mathcal{O}, E, I, C \rangle$ is the *composition* of composable distributed system models $\mathcal{M}_i = \langle \mathcal{O}_i, E_i, I_i, C_i \rangle$, $i \in \{0, 1\}$, if the following axioms are satisfied:

(GEN) Generalisation: $\mathcal{O} = \mathcal{O}_0 \cup \mathcal{O}_1$;

(EXT) Scoping: For every $o \in \mathcal{O}$, $E(o) \rightarrow E_0 \uparrow_{\mathcal{O}}(o) \vee E_1 \uparrow_{\mathcal{O}}(o)$;

(IN) Internalisation: For every $o \in \mathcal{O}$, $I_0 \uparrow_{\mathcal{O}}(o) \vee I_1 \uparrow_{\mathcal{O}}(o) \rightarrow I(o)$;

(COR) Correlation: For every $i \in \{0, 1\}$ and $o \in \mathcal{O}$, $C_i(o) = o' \rightarrow C(o) = o'$.

Much in the way that identifying an object in different models is an important part of the composition activity, particularly in order to ensure the satisfiability of the axioms above, the formulation of time correlation functions is also part of this activity. Note, however, that no new such function is required to exist in a composed model, since its components are not obliged to maintain between themselves time correlated objects.

A model composition example is obtained by considering as sub-systems a parent with an external bank account object, a child with its external account and the account itself, a sub-system composed by this single object. By composing these sub-systems pairwise, presuming that all their objects are time correlated and that the account becomes an internal object in the composition, we obtain a configuration having as a possible behaviour that described by Fig. 3. Note that the composition of these sub-systems is not unique, since we can also define a composed system with all the objects as internals, a so-called *closed system* [1].

3.4 Distributed System Representations

We study in this section distributed system representations, such as the specifications presented in Section 2. We are not, however, concerned here with the syntax of the adopted formal languages, since there are many available alternatives. A possibility is the use of the languages proposed in [22], which are layered: there is a local temporal language for defining objects and another language for expressing global system properties. Another possibility is the adoption of a single language with different usage contexts [9]: the local view of each object is captured by object specifications, whereas

⁵ $f \uparrow_G / f \downarrow_H$ denote the generalisation / particularisation of the boolean valued function f to the set G / H , with $\text{dom } f \subseteq G$, $H \subseteq \text{dom } f$ and $f \uparrow_G(x) \stackrel{\text{def}}{=} \begin{cases} f(x), & \text{for } x \in \text{dom } f \\ \text{FALSE}, & \text{otherwise.} \end{cases}$

system configurations and global properties are represented in the context of coordination specifications.

The semantics of such representations can be given in terms of object structures and (composed) distributed system models. We have already sketched how message and action symbols can be interpreted using such structures. The semantics of states and memory values can be given in the same predicative way.

Some semantically rich symbols are often found in such representations:

self: The immutable unique identification of an object;

loc: The current location (set) of an object;

acq: The current set of *acquaintances* of an object – the (identities of the) objects that became known at creation time or through object interaction;

Distributed system classifications are formulated in terms of the potential knowledge of these notions by the respective objects. If **self** is available, the system is said to be *identified*, or else it is said to be *anonymous* [4]. Whenever **loc** is available, the system is said to be *location aware*. If the interpretation of this symbol is not constant, the system objects are said to be *mobile* [8]. Moreover, if **T** is accessible by some objects, then we are dealing with *temporised* systems. By referring directly or indirectly to **T**, object computation and communication can be defined to happen more or less synchronously. We postpone the presentation of the classification corresponding to **acq** to Section 4.2.

It is important to distinguish if the above notions belong to the underlying models or are definable in terms of other notions. For this purpose, it becomes necessary to introduce the semantic notions of interpretation, satisfaction, truth and consequence [12]. Given a collection of specification symbols Δ (a signature) whose generated language is denoted by $lang(\Delta)$, we postulate the existence of an interpretation of Δ , a map $[\cdot]$ that, provided an object structure o , assigns each symbol in Δ to an event or location of o . Note that Δ may be partitioned in action/message and state symbols and that $lang(\Delta)$ may have other logical symbols such as the above, which are not part of signatures. The satisfaction $o \models_{\lambda}^t p$ of a sentence $p \in lang(\Delta)$ in a moment $t \in \text{dom } \lambda$ of a time flow $\lambda \in L$, a component of \mathcal{T} of a structure $o = \langle \mathcal{T}, \mathcal{V}, \mathcal{S}, B, P \rangle$, is defined by recursion on the structure of $lang(\Delta)$. The base cases of this definition in terms of Δ are:

- $o \models_{\lambda}^t s$ iff $\exists d \cdot d \in B([s]^o) \wedge t \in d$, if $[s]^o \in \mathbf{Ev}_o$;
- $o \models_{\lambda}^t s$ iff $\exists d \cdot d \in P([s]^o) \wedge t \in d$, if $[s]^o \in \mathbf{Loc}_o$;

A sentence $p \in lang(\Delta)$ is locally true in o and λ ($o \models_{\lambda} p$) whenever $o \models_{\lambda}^t p$ for every $t \in \text{dom } \lambda$. Whenever this is the case for every λ , p is simply true in o ($o \models p$). A sentence is valid if it is true in any such structure for Δ ($\models p$). A consequence relation $\Psi \models p$ between a set of sentences Ψ and a sentence p can be defined by stating that $o \models p$ whenever $o \models q$, for every $q \in \Psi$ and any admissible object structure o for Δ .

4 Topology

Since this section relies on topology, we present below the relevant definitions.

The aggregate of elements of a family of subsets F_X of a given set X is a topological space T_X whenever it satisfies the following axioms:

(T1) $\{\} \in F_X$ and $X \in F_X$;

(T2) $X_1 \cap \dots \cap X_n \in F_X$ for any $n \in \mathbb{N}$ and every $X_i \in F_X$, $1 \leq i \leq n$;

(T3) $\bigcup_{X_i \in P} X_i \in F_X$ for every $P \subseteq F_X$;

Elements of X are called points, elements of F_X are entitled opens and F_X is named a topology on X .

Given a topological space $T_X = (X, F_X)$, the complement of A under X is denoted by A' (that is, $A' \stackrel{\text{def}}{=} X - A$). Closed sets are complements of opens in F_X .

4.1 Behavioural Properties

Here we recast the topological characterisation of safety and liveness properties of concurrent and distributed systems of [3] in terms of our own formal framework.

We first provide necessary and sufficient logical characterisations for safety and liveness. Given a signature Δ and an object structure $o = \langle \mathcal{T}, \mathcal{V}, \mathcal{S}, B, P \rangle$ for Δ , $p \in \text{lang}(\Delta)$ is said to be a safety (liveness) property if and only if:

(SAFE) Safety: For every $\lambda \in L_{\mathcal{T}}$,

$$(\neg(o \models_{\lambda} p) \Rightarrow (\exists t \in \text{dom } \lambda \cdot \forall \lambda' \in L_{\mathcal{T}} \cdot \lambda \sqsubseteq_t^o \lambda' \rightarrow \neg(o \models_{\lambda'} p)))^6$$

(LIVE) Liveness: For every $\lambda \in L_{\mathcal{T}}$, $(\exists \lambda' \in L_{\mathcal{T}}; t \in \text{dom } \lambda \cdot \lambda \sqsubseteq_t^o \lambda' \wedge o \models_{\lambda'}^t p)$;

The first axiom states that, if we are not dealing with a safety property, it is possible to identify an instant in which a “bad thing” falsifies the property. On the other hand, the second axiom states that a liveness property guarantees the occurrence of a “good thing” at some instant. Examples of these properties are respectively that a withdrawal cannot happen if the account does not hold the required funds and that investors eventually demand repayment of the invested amounts.

Taking advantage of the necessary and sufficient characters of the above definitions and of the fact that object behaviours and placements are completely determined by their underlying time flows when related by \sqsubseteq , from now on we deal with distributed system properties by relying on the corresponding sets of time flows. That is, we use $P \stackrel{\text{def}}{=} \{\lambda \in L_{\mathcal{T}} \mid o \models_{\lambda} p\}$ instead of p .

Now we adopt these sets to provide a topological characterisation for safety and liveness properties. First note that \sqsubseteq_x^o obeys (R1) and (R3) for each $x \in \mathbf{T}$, that is, \sqsubseteq_x^o is a pre-order. Given $X \subseteq L_{\mathcal{T}}$, we define the following operators:

- $\text{Int}(X) \stackrel{\text{def}}{=} \{\lambda' \in L_{\mathcal{T}} \mid \exists \lambda \in X; x \in \text{dom } \lambda \cdot \lambda \sqsubseteq_x^o \lambda'\}$;
- $\text{Cl}(X) \stackrel{\text{def}}{=} \{\lambda \in L_{\mathcal{T}} \mid \exists \lambda' \in X; x \in \text{dom } \lambda' \cdot \lambda \sqsubseteq_x^o \lambda'\}$;

$\text{Int}(X)$ is a set of time flows with common prefixes in X . As in [3], these sets are considered to be opens. They correspond exactly to liveness properties here. $\text{Cl}(X)$ is a closed set and corresponds to a safety property. It is not difficult to see that opens obey (T1), (T2) and (T3). Therefore, their family $\text{Int}(X)$, $X \subseteq L_{\mathcal{T}}$, defines a topology.

⁶ We denote by $\lambda \sqsubseteq_t^o \lambda'$ the dominance of a function λ by another one λ' of the same type in an object structure o up to t . It is defined by:

$$\lambda \sqsubseteq_x^o \lambda' \stackrel{\text{def}}{=} \forall y \cdot y \leq x \rightarrow \left(\forall e \cdot \lambda^{-1}(y) \in B(e) \rightarrow \lambda'^{-1}(y) \in B(e) \wedge \forall s \cdot \lambda^{-1}(y) \in P(s) \rightarrow \lambda'^{-1}(y) \in P(s) \right)$$

Notice that the indexed sequences of program states used in [3], determined by our time flows λ and λ' in (LIVE), are respectively required to be finite and infinite. Consequently, liveness properties are characterised as dense sets therein. We do not consider this to be a reasonable requirement in a general temporal setting (so long as we maintain that “something good” eventually happens) and, as a result of this abstraction, obtain a characterisation in which the following holds:

Lemma 1. Liveness properties are closed under arbitrary intersections.

This is a direct consequence of the set-theoretic definition of Int . Consequently, by duality, safety properties are closed under arbitrary unions.

The main result concerning safety and liveness can be formulated as follows:

Theorem 1 (Behavioural Characterisation). Every property is an intersection of a safety and a liveness property.

Proof: Given a signature Δ and an object structure $o = \langle \mathcal{F}, \mathcal{V}, \mathcal{S}, B, P \rangle$ for Δ , it suffices to show, for $P \subseteq L_{\mathcal{F}}$ representing $p \in lang(\Delta)$, that $P \subseteq Int(X) \cap Cl(Y)$ for some $X \cup Y \subseteq L_{\mathcal{F}}$. But, for each $\beta \in P$, $\beta \in Int(\{\beta\}) \cap Cl(\{\beta\})$. Therefore:

$$\begin{aligned} P &\subseteq \bigcup_{\beta \in P} (Int(\{\beta\}) \cap Cl(\{\beta\})) && \subseteq (\bigcup_{\beta \in P} Int(\{\beta\})) \cap (\bigcup_{\beta \in P} Cl(\{\beta\})) \\ &\subseteq Int(\bigcup_{\beta \in P} \{\beta\}) \cap Cl(\bigcup_{\beta \in P} \{\beta\}) && \subseteq Int(P) \cap Cl(P) \end{aligned}$$

■

4.2 Structural Properties

In this section, we show that structural properties of distributed systems can also be characterised in topological terms.

Given an object structure $o = \langle \mathcal{F}, \mathcal{V}, \mathcal{S}, B, P \rangle$ and a countable value domain $Addr$, we postulate the existence of $\{self(Addr), acq(Addr)\} \subseteq \mathbf{E}v_o$ and that these events respectively capture the semantics of **self** and **acq** if available in the adopted representation language⁷. The *configuration* of the respective object for $\lambda \in L_{\mathcal{F}}$ and $t \in \text{dom } \lambda$ is defined by the following function:

$$Conf_{\lambda}(t) \stackrel{\text{def}}{=} \{i : Addr \mid t \in \cup B(acq(i))\}$$

This notion can be generalised to each object $o \in \mathcal{O}$ of a distributed system model $\mathcal{M} = \langle \mathcal{O}, E, I, C \rangle$ as follows:

$$Conf_{\lambda}^o(t) \stackrel{\text{def}}{=} \{i : Addr \mid t \in \cup B_o(acq(i)) \wedge C(o) = o' \wedge \kappa_o(t) \in \cup B_{o'}(self(i))\}$$

Without loss of generality, we deal with objects themselves in place of their identifications, since these are unique and immutable. Concerning each distributed system $X \subseteq \mathcal{O}$ that \mathcal{M} represents, we say that it has a *static configuration* if and only if, for each $o \in X$, $Conf_{\lambda}^o(t)$ is constant over $t \in \text{dom } \lambda$, for every $\lambda \in L_{\mathcal{F}}$. Otherwise, it is said to have a *dynamic configuration*.

⁷ Moreover, that $\forall x : Addr \cdot self(x) \rightarrow acq(x)$.

Going back to Section 3.3, it is easy to see that the example system composed by three objects has a static configuration whenever the unique allowed interaction pattern is that described by Fig. 3. On the other hand, if child or parent objects are allowed to interact with the external environment, the system will have a dynamic configuration.

It is the open and closed character of some distributed system configurations that is subject to a topological definition. Towards this, we need two further generalisations of distributed system configuration notions, respectively without any reference to the passage of time and covering sets of objects instead of single objects. The corresponding definitions are presented below:

$$Conf(o) \stackrel{\text{def}}{=} \bigcup_{\substack{t \in \text{dom } \lambda \\ \lambda \in L_{\mathcal{F}}}} Conf_{\lambda}^o(t); \quad Conf(X) \stackrel{\text{def}}{=} \bigcup_{o \in X} Conf(o)$$

Given $X \subseteq \mathcal{O}$, $Conf(X)$ is considered to be an open. Again, opens obey (T1), (T2) and (T3) and the family $Conf(X)$, $X \subseteq \mathcal{O}$, defines a topology. Moreover,

Lemma 2. Configurations are closed under arbitrary intersections.

The following is a structural counterpart to our behavioural characterisation:

Theorem 2 (Structural Characterisation). Every object universe is an union of open and closed disjoint sets of objects.

Proof: Given a model $\mathcal{M} = \langle \mathcal{O}, E, I, C \rangle$, let the set of internals be $J \stackrel{\text{def}}{=} \{o \in \mathcal{O} | I(o)\}$, the externals set be $F \stackrel{\text{def}}{=} \{o \in \mathcal{O} | E(o)\}$ and $R \stackrel{\text{def}}{=} \mathcal{O} - J - F$. Hence:

1. The open set containing all the objects directly reachable from the environment is $Conf(F)$. Therefore, $Conf(F)'$ is the closed set corresponding to J ;
2. The open set of objects corresponding to R is $Conf(J) \cap Conf(F)$;
3. The closed set corresponding to F is $(Conf(J) \cup (Conf(J) \cap Conf(F)))'$.

These three sets are disjoint and their union corresponds to \mathcal{O} . ■

It is interesting to mention that the set R above corresponds precisely to the *receptionist* objects of [1] and, more generally, to the objects reachable from the environment. These are the central objects in each distributed system model.

It is not surprising to reach the conclusion above, since it captures the standard intuition in distributed systems development that internals are substitutable and the environment uncertain. The theorem shows in particular that, if we ignore the empty sets that determine trivial distributed system models and adopt the terminology mentioned in Section 3.3, each single closed system corresponds precisely to a closed object set in our topology: it determines the set of internals of the model, whereas the sets of externals and receptionists are empty in this case.

4.3 Transference of Results

Now we show how results on topological spaces can be transferred directly to distributed system theory. We focus our attention in the problem of determining the class of distributed system models in which external objects not able to reach the modelled systems have been eliminated therein.

Due to the nature of this problem, we are obliged to introduce other topological notions. We say that $f : A \rightarrow B$ is continuous if the inverse images of closed sets under f are closed. In a topological space $T_X = (X, F_X)$, a sub-set $Y \subseteq X$ is said to be connected if there is no way to define Y as a union of two disjoint nonempty open sets. The connected component of $p \in X$ is a sub-set $C_p \subseteq X$ such that $p \in C_p$, C_p is connected and, if $p \in C$ for some connected $C \subseteq X$, $C \subseteq C_p$. The connected components of T_X are the respective sets that partition X . The following result is used in the sequel:

(CC) The connected components of a topological space are closed sets;

Let us spell out what we mean by an operation of restriction (modulo equivalence). This and other operations on models can be represented by means of injective set inclusion functions on objects whose inverse images map safety properties and internals into similar entities and also preserve separation and correlation. Note that the inverse image of these functions map closed sets (safety properties and internal objects) into closed sets, characterising them to be continuous.

Theorem 3 (Model Restriction). Each distributed system model can be restricted to an equivalent model without disconnected externals.

Proof: Given $\mathcal{M} = \langle \mathcal{O}, I, E, C \rangle$, let $A = \{A_i | A_i \subseteq \mathcal{O}\}$ be the family of connected components of the topological space $(\mathcal{O}, \{Conf(X) | X \subseteq \mathcal{O}\})$. Due to Theorem 2, we know that $\mathcal{O} = J \cup R \cup F$ for some closed J and F and some open R sets as defined therein. By (CC), we know that each A_i is a closed set. If $|A| = 0$, \mathcal{M} is the trivial model which is clearly equivalent to itself. Alternatively, if $|A| \geq 1$, due to the dual of Lemma 2, $D = \bigcup_{A_i \subseteq F} A_i$ is the closed set of disconnected externals.

Put $\mathcal{O}' \stackrel{\text{def}}{=} \mathcal{O} - D$ and $\mathcal{M}' \stackrel{\text{def}}{=} \langle \mathcal{O}', I \downarrow_{\mathcal{O}'}, E \downarrow_{\mathcal{O}'}, C \downarrow_{\mathcal{O}'} \rangle$. Clearly, the function $f : \mathcal{O}' \rightarrow \mathcal{O}$ defined by the identity on \mathcal{O}' satisfies all the requirements to be considered a restriction. Consequently, \mathcal{M}' is the representative of the class of models equivalent to \mathcal{M} . ■

The result above illustrates how to take advantage of topology results, namely (CC), to develop distributed systems theorems. If it is proven to be decidable for some class of models, it can be used to develop static analyses of distributed system representations that may suggest simplifications to software engineers. With a bit of ingenuity, it can also be extended to formalise distributed garbage collection.

5 Concluding Remarks

In the present paper, we proposed a novel characterisation of object-based distributed systems in terms of algebraic structures and topological spaces. Although there seems to exist a growing consensus concerning the importance of topological methods in distributed computing, we are not aware of other research efforts that address their whole development process in this way.

The proposed algebraic structures are sufficiently general to express most distributed system notions, such as diverse modes of object creation, configuration, interaction and timing usually found in the literature. In particular, they are a generalisation of the algebraic notions first developed in [2,25]. The topological analysis of these structures

allowed us to recast here behavioural results of [3], to develop similar results concerning distributed system configurations and also to exemplify how topological results can be transferred to distributed systems theory. We consider these to be the main original contribution of our research.

The reported research yields a general foundation for the definition of logical systems devoted to the compositional development of object-based distributed systems. It is also important towards establishing a semantically rich framework for the analysis and simulation of distributed system behaviours, facilitating their comprehension.

We expect to refine in the future the connections of the reported research with our previous work on object-based mobility [8], specification [9] and implementation [11]. It is in perspective an extension of this work towards applying the theory of dynamical systems to distributed systems development. Another interesting direction for future work is to formalise the method of transference of results from the aforementioned mathematical theories to distributed systems theory using Institutions [12].

Acknowledgement. The author gladly acknowledges that the remarks from an anonymous referee contributed to improve the readability of this paper.

References

1. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press (1986)
2. Agha, G., Mason, I., Smith, S., Talcott, C.: A foundation for actor computation. *Journal of Functional Programming* 7(1), 1–72 (1997)
3. Alpern, B., Schneider, F.B.: Defining liveness. *Information Processing Letters* 21(4), 181–185 (1985)
4. Attiya, H., Snir, M., Warmuth, M.: Computing on an anonymous ring. *Journal of the ACM* 35(4), 845–876 (1988)
5. Barbosa, V., Gafni, E.: Concurrency in heavily loaded neighborhood-constrained systems. *ACM Transactions on Programming Languages and Systems* 11, 584–592 (1989)
6. Chandy, K.M., Misra, J.: *Parallel Program Design, A Foundation*. Addison-Wesley (1988)
7. Denker, G., Ehrich, H.D.: Specifying distributed information systems: Fundamentals of an object-oriented approach using distributed temporal logic. In: Bowman, H., Derrick, J. (eds.) *Prof. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems Conference (FMOODS 1997)*, vol. 2, pp. 89–104. Chapman and Hall (1997)
8. Duarte, C.H.C.: A proof-theoretic approach to the design of object-based mobility. In: Bowman, H., Derrick, J. (eds.) *Proc. 2nd IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 1997)*, pp. 37–53. Chapman and Hall (July 1997)
9. Duarte, C.H.C., Maibaum, T.: A rely-guarantee discipline for open distributed systems design. *Information Processing Letters* 74(1-2), 55–63 (2000)
10. Duarte, C.H.C., Maibaum, T.: A branching-time logical system for open distributed systems development. *Electronic Notes on Theoretical Computer Science* 67 (2002)
11. Duarte, C.H.C., Talcott, C.: Clara: An actor language for high performance distributed computing. In: *Proc. 12th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2000)*, pp. 20–37 (October 2000)
12. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. *Journal of the ACM* 39(1), 95–146 (1992)

13. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In: Distributed Systems, pp. 97–145. Addison-Wesley (1993), ch. 5 of [18]
14. Herlihy, M.P., Shavit, N.: The topological structure of asynchronous computation. *Journal of the ACM* 46, 856–923 (1999)
15. Lamport, L.: The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16(3), 872–923 (1994)
16. Lefschetz, S.: *Algebraic Topology*. American Mathematics Society (1942)
17. Lynch, N.: *Distributed Algorithms*. Morgan Kaufmann (1996)
18. Mullender, S. (ed.): *Distributed Systems*, 2nd edn. Addison-Wesley (1993)
19. Nielsen, M., Plotkin, G., Winskel, G.: Petri-nets, event-structures and domains - part I. *Theoretical Computer Science* 13, 85–108 (1981)
20. O. M. G. Unified Modelling Language Specification. Object Management Group — OMG, Version 1.3 (June 1999)
21. Plotkin, G.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, University of Aarhus (1981)
22. Ramanujam, R.: Locally linear time temporal logic. In: Proc. 11th IEEE Symposium on Logic in Computer Science, pp. 118–127. IEEE Computer Society Press (1996)
23. Saks, M., Zaharoglou, F.: Wait-free k -set agreement is impossible: The topology of public knowledge. *Siam Journal on Computing* 29, 1449–1483 (2000)
24. Schneider, F.B.: What good are models and what models are good. In: Distributed Systems, pp. 17–26. Addison-Wesley (1993), ch. 12 of [18]
25. Talcott, C.: Composable semantic models for actor theories. *Higher-Order and Symbolic Computation* 11(3), 281–343 (1998)
26. Tarski, A.: *Logics, Semantics and Metamathematics*. Oxford Publishing Company (1956)

From Explicit to Symbolic Types for Communication Protocols in CCS

Hanne Riis Nielson¹, Flemming Nielson¹, Jörg Kreiker², and Henrik Pilegaard²

¹ DTU Informatics, Technical University of Denmark, Denmark
{riis,nielson}@imm.dtu.dk

² Institut für Informatik, Technische Universität München, Germany
joba@model.in.tum.de, henrik@pilegaard.org

Abstract. We study communication protocols having several rounds and expressed in value passing CCS. We develop a type-based analysis for providing an *explicit* record of all communications and show the usual subject reduction result. Since the explicit records can be infinitely large, we also develop a type-based analysis for providing a finite, *symbolic* record of all communications. We show that it correctly approximates the explicit record and prove an adequacy result for it.

1 Introduction

Motivation. Modern IT Systems are complex and contain a number of challenges: concurrency, distribution and secure communication over insecure networks to name but a few. One approach to designing and analysing such systems is through the use of high-level models of the underlying computational paradigms — these can then be scrutinized using a number of formal approaches or by studying the performance of concrete prototypes. The number of such paradigms is vast but in our view the focus on distributed and concurrent processes is evident in calculi such as Actors (e.g. [1]), Obliq [6], and Klaim (e.g. [12]).

Adapting static analysis techniques to apply to such calculi is no trivial pursuit¹ although many useful approaches have emerged. Some of us have been actively involved in pursuing the use of Type and Effect Systems [13] together with process calculi and in the development of Flow Logic for programming languages and process calculi [16,17]. Apart from ensuring semantic correctness and algorithmically tractable ways of obtaining “best” analysis results we also consider it important to strive for approaches that find the proper balance between precision (using state-of-the-art methods and techniques) and technical detail (to allow to be assimilated by the community).

The problem. In this paper we study the key challenge of secure communication over an insecure network. One of the key challenges of communication protocols

¹ The first two authors worked with Carolyn Talcott and Chris Hankin during the *Atlantique*-project in the mid 1990'es on developing static analysis for a fragment of Obliq but this work did not go much further than indentifying the key obstacles.

is that the desired communication is indeed exchanged between the intended participants. Often communication takes place in a number of rounds where it is essential that information pertaining to one round does not interact with other rounds. In short, the private messages exchanged between the participants in one round should not erroneously show up in another round. We show how to develop a static analysis for verifying this property.

To express the communication protocols we use a simple value passing variant of Milner’s CCS (as in Chapter 4 of [11] but with replication instead of recursion as well as polyadic name passing) rather than the more complex calculi mentioned above. For a concrete example consider the value passing CCS expressions $P_1 = !(\nu n)(\nu m)(n\langle m \mid n(x) \rangle)$ and $P_2 = (\nu n)!(\nu m)(n\langle m \mid n(x) \rangle)$. In both cases, a number of rounds is carried out, as indicated by the replication operator $!$, and in each round a fresh name m is output on a channel n and subsequently input on the same channel and bound to a variable x . The difference between the two processes is that in the case of P_1 a fresh channel is used in each round whereas in the case of P_2 the same global channel is used in all rounds. Consequently, only in P_1 is there no interaction between rounds whereas this is possible in P_2 . Our type based analysis will be able to pinpoint this difference between P_1 and P_2 .

To facilitate the development of a static analysis that is able to pinpoint this difference between P_1 and P_2 we introduce a value passing variant of CCS, which incorporates round information on the replication operators and on names and variables introduced within the “scope” of such replication operators. In this notation the examples from above read

$$\begin{aligned} P_1 &= !^{a:1} (\nu n^{a:?}) (\nu m^{a:?}) (n^{a:?} \langle m^{a:?} \mid n^{a:?} (x^{a:?}) \rangle) \\ P_2 &= (\nu n^\epsilon) !^{a:1} (\nu m^{a:?}) (n^\epsilon \langle m^{a:?} \mid n^\epsilon (x^{a:?}) \rangle) \end{aligned}$$

The superscripts are called *round indicators* and the identifiers in front of the colons are called *round identifiers*. They indicate syntactically, which unfolding instance of a replication a name or variable belongs to.

Overview. In Section 2, our value passing CCS is equipped with a fairly standard reaction semantics and a structural congruence – however, while we allow to unfold replication we shall not allow to fold it back.

Our first static analysis, in Section 3, is a type system that gives an *explicit* record of the data communicated and bound to variables. It is precise in recording the round information that pertains to the data and variables. This is achieved by ensuring that names and variables bound within the scope of several replication operators are all indexed by sequences of values corresponding to sequences of round indicators as already illustrated in the syntax of P_1 and P_2 above. We prove the correctness of this analysis using a standard subject reduction result. The analysis suffices for pinpointing the difference between P_1 and P_2 .

The main drawback of this analysis is its use of an infinite set of rounds and therefore the explicit record is necessarily infinite. In Section 4, we therefore show how to develop a type system that gives a *symbolic* record of the solutions prescribed by our analysis. We design it so as to accurately track the identities

Table 1. The action prefixes, π , of value passing CCS

$\pi \in \mathbf{Act} ::=$	Action prefixes
τ	Internal action
$n(\bar{u})$	Output action
$n(\bar{x})$	Input action

of round information (whereas more powerful techniques would be needed for expressing affine relationships between various rounds, e.g. [9]). To this end we use simple equations between symbolic round indicators, thereby effectively partitioning the round indicators into equivalence classes belonging to the same round. We then show that the analysis results presents an overapproximation, expressed using a suitable concretisation function, of the explicit types prescribed by the previous type system. Our adequacy result shows that our analysis is able to prove that programs adhere to a novel notion of *round consistency* defined in Definition 1 in Section 2.

We sketch a worked example in Section 5 based on the *Diffie-Hellman* key exchange protocol [7], and conclude in Section 6 by discussing related work and the challenges posed by distribution and discussing possible extensions of our approach to more expressive calculi and more demanding analyses.

2 Value Passing CCS

Communication protocols often consist of a number of rounds. In the presence of several concurrent instances of the same process, one needs to ensure that there is no cross-over talk between instances belonging to different rounds. In Definition 1, we shall capture this by a notion of *round consistency*. In order to concentrate on this critical aspect, we shall base our developments on a puristic process algebraic model of concurrent systems. In the following we shall therefore present a polyadic value-passing variant of Milner's CCS (Chapter 4 of [11]) that shall serve as the basis for the developments of later sections.

2.1 Syntax

The fundamental data entities of the language are *names*, $n, m \in \mathbf{Name}$. Furthermore, the language has *variables*, $x, y \in \mathbf{Var}$, that act as placeholders for names. When an entity can be either a name or a variable we use metavariables $u, v \in (\mathbf{Name} \cup \mathbf{Var})$.

The operational activity primitives of the language are *actions* as defined by Table 1. Such an action may be either some internal activity, τ , the polyadic output, $n(\bar{u})$, of a data tuple, $\bar{u} = u_1 \cdots u_k$, over a *channel*, n , or the polyadic input, $n(\bar{x})$, of polyadic data received via a channel, n , into a tuple of variables, $\bar{x} = x_1 \cdots x_k$.

The programs of the language are *processes* as defined by Table 2. The *guarded sum*, $\sum_{i=1}^k \pi_i . P_i$ can choose non-deterministically between k processes that are

Table 2. The processes, P , of value passing CCS

$P \in \mathbf{Proc} ::=$	Processes
$\sum_{i=1}^k \pi_i . P_i$	Guarded sum
$P_1 \mid P_2$	Concurrent composition
$(\nu n) P$	Scope restriction
$!^{\beta:k} P$	Process replication

each a sequential composition of an *action prefix*, π_i , and a *continuation*, P_i . In the nullary and unary special cases this construct specialises to the *nil* process, $\mathbf{0}$, and the ordinary sequential composition, $\pi . P$, respectively. Processes, P_1 and P_2 executing in parallel are composed by *concurrent composition*, $P_1 \mid P_2$. A name, n , is made private to a subsystem, P , by *scope restriction*, $(\nu n) P$. Finally, a process, P , is replicated as many times as necessary by *process replication*, $!^{\beta:k} P$, where we explain the purpose of the superscript below.

In order for the type systems of Sections 3 and 4 to ensure *round consistency* (formally defined in Definition 1), it must be possible to unambiguously relate names and variables to a unique *round*, i.e. a particular recursive process instance. Therefore, each process replication, $!^{\beta:k} P$, is annotated by a *round indicator*, $\beta : k$, where $\beta \in \mathbf{Id}$ denotes a *round identifier* that is unique for this particular process replication and $k \in \mathbb{N}$ indicates that the next replication instance will be the k th.

In turn, these indicators are used to annotate names and variables and, thereby, associate them to a particular recursive instance. More specifically, each name, n , is really a composition, $n^e \in \mathbf{Nid} \times \mathbf{Ind}$, where $n \in \mathbf{Nid}$ is a *name identifier* and $e \in \mathbf{Ind} = (\mathbf{Id} \times (\mathbb{N} \cup \{?\}))^*$ is a *round expression*. The same is true for variables, where $\mathbf{Var} = \mathbf{Vid} \times \mathbf{Ind}$.

The importance of the *round expression*, being a sequence of indicators corresponding to a nesting of replicated processes, is that it uniquely identifies the originating recursive instance. We shall write ϵ for the empty sequence. Furthermore, whenever e is a round expression and β is a round indicator having a unique occurrence in e , we shall write $e.\beta$ for the corresponding value in $\mathbb{N} \cup \{?\}$.

Table 3. Directed congruence, \Rightarrow , of value passing CCS

PARUNIT	$P \mid \mathbf{0} \equiv P$
PARCOMM	$P_1 \mid P_2 \equiv P_2 \mid P_1$
PARASSOC	$P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$
SUMREORD	$\sum_{i=1}^k \pi_i . P_i \equiv \sum_{i=1}^k \pi_{\sigma(i)} . P_{\sigma(i)}$
SCPEXTR	$(\nu n)(P_1 \mid P_2) \equiv P_1 \mid (\nu n)P_2$ if $n \notin \text{fn}(P_1)$
SCPUNIT	$(\nu n)\mathbf{0} \Rightarrow \mathbf{0}$
SCPREORD	$(\nu n^e)(\nu m^f)P \equiv (\nu m^f)(\nu n^e)P$ if $n \neq m$
REPUNFOL	$!^{\beta:k} P \Rightarrow P\{\beta:k/\beta:?\} \mid !^{\beta:k+1} P$
α -EQUIV	$P_1 \equiv_{[\alpha]} P_2 \Rightarrow P_1 \equiv P_2$

Name and variables bound within a replicated process, e.g. $(\nu n) P_1$ occurring within $!^{\beta:k} P$, are initially indexed with a '?' in the corresponding position, i.e. $n = n^{e\beta:?\text{f}}$, as will become clear when we define our notion of well-formed and strongly well-formed processes.

Running Example. Our running example, the two processes P_1 and P_2 , was already presented in this annotated form in the Introduction.

2.2 Reaction Semantics

We now define a *reaction semantics* for the language in the style of the chemical abstract machine [3]. As is customary for this style of semantics we define two binary relations on processes as explained in the sequel.

Usually, a symmetric *structural congruence relation*, \equiv , relates processes that are considered identical up to trivial syntactic restructuring. We shall use the directed variant, \Rightarrow , defined as the least substitutive relation defined by the axioms and rules of Table 3, where we write $P_1 \equiv P_2$ as a shorthand for $P_1 \Rightarrow P_2$ and $P_2 \Rightarrow P_1$. The desirability of having a directed version is due to the presence of round expressions. The directed relation prevents arbitrary new round identifiers to be introduced by a backwards application of rules SCPUNIT and REPUNFOL. The structural congruence relation normally includes a notion of α -equivalence, \equiv_α , and therefore names have no syntactic representation that is stable under evaluation. This means that names are not suitable for the representation of static information, and when we later define the type systems, stable representations are mandatory. We shall therefore make the assumption that each statically occurring name identifier, n , is associated with a corresponding *canonical name identifier*, $[n]$, that is invariant under α -renaming. This leads to a more restrictive notion of *canonical α -equivalence*, $\equiv_{[\alpha]}$, under which renaming can only take place between name identifiers with the same canonical representation.

The *reaction relation*, \longrightarrow , relates processes that evolve into one another by a single semantic transition. It is defined as the least relation adhering to the axioms and rules of Table 4, where we write $\{\bar{m}/\bar{x}\}P$ for the process that is as P except that every free occurrence of x_1, \dots, x_k is replaced by m_1, \dots, m_k . The axiom TAU allows a process to advance by the consumption of a τ . The axiom REACT allows two concurrent processes to advance by the consumption of matching input and output actions, thereby causing a synchronising data-exchange between them. The rules PAR and RES assert that reaction may take place in an immediate sub-context of parallel composition and restriction, respectively. Finally, the STRUCT rule tells us that we may rely on the directed congruence to bring processes on the form required by TAU or REACT.

Well-formed Processes. In the sequel, we assume processes to be *well-formed*:

- Different occurrences of replications use different round identifiers.
- Whenever a round expression, e , is used for annotating variable and name identifiers we have $e.\beta = ?$ if and only if e is “in scope” of a replication $!^{\beta:k}$ (for some k).

Table 4. Reaction relation, \longrightarrow , of value passing CCS

TAU	$\tau.P + M \longrightarrow P$
REACT	$(n(\bar{x}).P_1 + M) \mid (n(\bar{m}).P_2 + N) \longrightarrow \{\bar{m}/\bar{x}\}P_1 \mid P_2 \quad \text{if } \bar{m} = \bar{x} $
PAR	$\frac{P_1 \longrightarrow P_2}{P_1 \mid P_3 \longrightarrow P_2 \mid P_3}$
RES	$\frac{P_1 \longrightarrow P_2}{(\nu n)P_1 \longrightarrow (\nu n)P_2}$
STRUCT	$\frac{P_1 \cong P_2 \wedge P_2 \longrightarrow P_3 \wedge P_3 \cong P_4}{P_1 \longrightarrow P_4}$

A process is *strongly well-formed* if *additionally* it is the case that:

- All replications $!\beta:k$ have $k = 1$.
- Whenever a round expression, e , is used for annotating variable and name identifiers we have $e.\beta = ?$ (for all round identifiers, β , occurring in e).
- All names and variables occur in their canonical form.

Well-formedness is preserved under reduction unlike strong well-formedness.

Round Consistency. The notion of *round consistency* defined below is parametric in a round identifier, β . It states, that all bindings of a variable to a name, which are both defined under the replication annotated by β , occur only between instances of the same round, e.g., the i -th instance of x will only be bound to i -instances of name identifiers. Effectively, this will exclude harmful cross-talk.

Definition 1 (Round Consistency). *Let P be a well-formed process and β be a round identifier. Process P is β -round consistent iff for all processes P' and P'' such that*

- $P \longrightarrow^* P' \longrightarrow P''$
- $P' \longrightarrow P''$ is due to a communication involving $n^e(x^f)$ and $n^e(n^g)$.

it holds that $f.\beta = g.\beta$.

Running Example. Note that process P_1 of our running example is in fact \mathbf{a} -round consistent, while P_2 is not. The latter is the case, because there may be, e.g., a binding of $x^{a:2}$ to $n^{a:3}$.

3 Explicit Types

We now develop an analysis that gives a global record of all the communications taking place within a process in value passing CCS. The basic idea is to consider each $!\beta:k P$ as being a shorthand for the infinite parallel composition $P\{\beta:k/\beta:?\} \mid \cdots \mid P\{\beta:k+n/\beta:?\} \mid \cdots$ where $P\{\beta:i/\beta:?\}$ denotes the result of replacing

all occurrences of $\beta : ?$ in P by $\beta : i$. This transformation results in an infinite process without any replication operators (and where names and variables could be renamed to get rid of round expressions). Next, we perform a type based analysis mimicking a standard *OCFA* analysis, e.g., like in [5], of this potentially infinite process to obtain an *explicit* record of all communications that may take place and of all variable bindings that may arise. Finally, the challenge of Section 4 then is to obtain a finite specification using symbolic round expressions and constraints over them.

Analysis Domains. The analysis is defined over the following domains:

- The explicit channel environment, $\hat{K} \subseteq \hat{\mathcal{K}} = \mathbf{Name}^+$, will contain an explicit record of all tuples $\langle \mathbf{n}, \mathbf{m}_1, \dots, \mathbf{m}_k \rangle$ such that an output of $\mathbf{m}_1, \dots, \mathbf{m}_k$ might be performed over the channel \mathbf{n} during the execution of the analysed program.
- The explicit environment, $\hat{R} \subseteq \hat{\mathcal{R}} = \mathbf{Var} \times \mathbf{Name}$, will contain an explicit record of all pairs $\langle \mathbf{x}, \mathbf{n} \rangle$ such that an input over some channel might give rise to the name \mathbf{n} being bound to the variable \mathbf{x} .

We shall arrange that all variables and names only occur in their canonical form.

Acceptability Judgement. The acceptability judgement takes the following form:

$$(\hat{K}, \hat{R}) \vdash P$$

It expresses that the *explicit record* given by (\hat{K}, \hat{R}) correctly describes the behaviour of the *well-formed* process P as well as of all its descendants, i.e. all processes P' such that $P \longrightarrow^* P'$, and it is defined in Table 5. We explain its key components in the sequel and shall ensure that all variables and names in P occur in their canonical form; hence for an “arbitrary” process P we shall perform the analysis on $[P]$.

The inference rules for summation and parallel composition are straightforward: we just make sure to analyse each component. Similarly, the rules for fresh names and for τ are straightforward. The rule for replication generates an infinite obligation to correctly type all possible instances leaving us with the axiom schemes for output and input.

The axiom scheme for output first determines the set $\langle \hat{R} \rangle(u_i^{e_i})$ of values that can be output in the i 'th position of the current output. In case $u_i^{e_i}$ is a variable \mathbf{x} , $\langle \hat{R} \rangle(u_i^{e_i})$ is simply the set of values $\hat{R}(\mathbf{x}) = \{\mathbf{n} \mid \langle \mathbf{x}, \mathbf{n} \rangle \in \hat{R}\}$. In case $u_i^{e_i}$ is a name \mathbf{n} , $\langle \hat{R} \rangle(u_i^{e_i})$ is the singleton set $\{\mathbf{n}\}$. The axiom scheme then takes the combination of all such values, using the Cartesian product $RR = \langle \hat{R} \rangle(u_1^{e_1}) \times \dots \times \langle \hat{R} \rangle(u_k^{e_k})$, and records those as being part of what is communicated over the channel n^e ; to be specific, $RR \subseteq \hat{K}(n^e)$ abbreviates $\{n^e\} \times RR \subseteq \hat{K}$.

For input, the set $\hat{K}_i(n^e) = \{\mathbf{m}_i \mid \langle n^e, \mathbf{m}_1, \dots, \mathbf{m}_k \rangle \in \hat{K}\}$ determines those values that can be received in the i 'th position of the current input over the channel n^e . The axiom scheme records all those values as being part of what the corresponding input variable can be bound to. As above, $\hat{K}_i(n^e) \subseteq \hat{R}(x_i^{e_i})$ is a shorthand for $\{x_i^{e_i}\} \times \hat{K}_i(n^e) \subseteq \hat{R}$.

Table 5. The acceptability judgement, $(\hat{K}, \hat{R}) \Vdash P$, of the *explicit* analysis

$$\begin{array}{c}
\frac{\forall i \leq k : (\hat{K}, \hat{R}) \Vdash \pi_i \wedge (\hat{K}, \hat{R}) \Vdash P_i}{(\hat{K}, \hat{R}) \Vdash \sum_{i=1}^k \pi_i . P_i} \quad \frac{(\hat{K}, \hat{R}) \Vdash P_1 \quad (\hat{K}, \hat{R}) \Vdash P_2}{(\hat{K}, \hat{R}) \Vdash P_1 \mid P_2} \\
\\
\frac{(\hat{K}, \hat{R}) \Vdash P}{(\hat{K}, \hat{R}) \Vdash (\nu n) P} \quad \frac{\forall i \geq k : (\hat{K}, \hat{R}) \Vdash P\{\beta^{:i} / \beta^{:?}}\}}{(\hat{K}, \hat{R}) \Vdash !^{\beta^{:k}} P} \\
\\
(\hat{K}, \hat{R}) \Vdash \tau \\
\\
\frac{\langle \hat{R} \rangle (u_1^{e_1}) \times \dots \times \langle \hat{R} \rangle (u_k^{e_k}) \subseteq \hat{K}(n^e)}{(\hat{K}, \hat{R}) \Vdash n^e \langle u_1^{e_1}, \dots, u_k^{e_k} \rangle} \quad \frac{\forall i \leq k : \hat{K}_i(n^e) \subseteq \hat{R}(x_i^{e_i})}{(\hat{K}, \hat{R}) \Vdash n^e \langle x_1^{e_1}, \dots, x_k^{e_k} \rangle}
\end{array}$$

Correctness Properties. We now establish the subject reduction result, which establishes the “internal consistency” of the type system, leaving the adequacy result to the next section.

Theorem 1 (Subject Reduction). *Let P and Q be well-formed processes. If there exists (\hat{K}, \hat{R}) such that $P \longrightarrow Q$ and $(\hat{K}, \hat{R}) \Vdash [P]$ then $(\hat{K}, \hat{R}) \Vdash [Q]$.*

Proof. The theorem is proved by induction on $P \longrightarrow Q$ and makes uses of the following standard lemmas. \square

Lemma 1 (Substitution). *Let P be a well-formed process such that there exists (\hat{K}, \hat{R}) with $(\hat{K}, \hat{R}) \Vdash [P]$ and let $[(x, n)] \in \hat{R}$. Then $(\hat{K}, \hat{R}) \Vdash [P\{n/x\}]$.*

Lemma 2 (Structural). *Let P and Q be well-formed processes such that there exists (\hat{K}, \hat{R}) with $(\hat{K}, \hat{R}) \Vdash [P]$. If $P \Rightarrow Q$ then $(\hat{K}, \hat{R}) \Vdash [Q]$.*

Running Example. Recall the two processes:

$$\begin{aligned}
P_1 &= !^{a:1} (\nu n^{a:?}) (\nu m^{a:?}) (n^{a:?} \langle m^{a:?} \mid n^{a:?} (x^{a:?}) \rangle) \\
P_2 &= (\nu n^e) !^{a:1} (\nu m^{a:?}) (n^e \langle m^{a:?} \mid n^e (x^{a:?}) \rangle)
\end{aligned}$$

The least analysis result that is acceptable for P_1 is given by:

$$\hat{K}_1 = \{(n^{a:i}, m^{a:i}) \mid i \geq 1\} \quad \hat{R}_1 = \{(x^{a:i}, m^{a:i}) \mid i \geq 1\}$$

Similarly, the least analysis result that is acceptable for P_2 is given by:

$$\hat{K}_2 = \{(n^e, m^{a:j}) \mid j \geq 1\} \quad \hat{R}_2 = \{(x^{a:i}, m^{a:j}) \mid i, j \geq 1\}$$

This clearly shows the ability of our type based *explicit* analysis to pinpoint the important difference in behaviour between whether or not local channels or one common global channel is used in the various rounds. Also, it establishes the a -round consistency of P_1 and that this property might not hold for P_2 .

4 Symbolic Types

Type inference for the explicit types of Section 3 is hard due to the construction of infinite solutions and infinite proof obligations. In this section, we develop a symbolic variant of the explicit system yielding an implementable inference algorithm. Explicit and symbolic types will be formally related in Theorem 2 using a *concretisation* function. In combination with Theorem 1, this will establish the soundness of the symbolic type system. We will conclude this section showing our adequacy result in Theorem 3, which shows that the analysis is precise enough to ensure *round consistency*.

Analysis Domains. The symbolic analysis domains are derived from the explicit ones by substituting natural numbers occurring in round expressions by symbolic *constraint identifiers* as the ι in $\beta : \iota$. The domain of constraint identifiers is written **CId** and ι, j, κ represent elements of **CId**. Furthermore, we record *equality constraints* among constraint identifiers, that we shall write as $\iota = j$. Formally, we define the following symbolic variants of **Name** and **Var**, where **CInd** are *symbolic round expressions*:

$$\begin{aligned} \mathbf{CInd} &= (\mathbf{IId} \times \mathbf{CId})^* \\ \mathbf{CName} &= \mathbf{NId} \times \mathbf{CInd} \\ \mathbf{CVar} &= \mathbf{VId} \times \mathbf{CInd} \end{aligned}$$

We shall use symbols \tilde{e} , \tilde{f} and \tilde{g} to denote symbolic round expressions. The symbolic channel environment \tilde{K} and the symbolic environment \tilde{R} are then defined like their explicit counterparts with the addition of equality constraints, which effectively define an equivalence relation on constraint identifiers:

$$\begin{aligned} \tilde{K} \subseteq \tilde{\mathcal{K}} &= \mathbf{CName}^+ \times \mathcal{P}(\mathbf{CId} \times \mathbf{CId}) \\ \tilde{R} \subseteq \tilde{\mathcal{R}} &= \mathbf{CVar} \times \mathbf{CName} \times \mathcal{P}(\mathbf{CId} \times \mathbf{CId}) \end{aligned}$$

Once more we shall arrange that all variables and names only occur in their canonical form.

The relation between symbolic and explicit types is formally specified in terms of *concretisation*. Intuitively, an explicit type is a concretisation of a symbolic one, whenever the concrete natural numbers denoting rounds in an explicit round expression satisfy the constraints specified in the symbolic type. For example, the concretisation of the symbolic channel environment $\{(\mathbf{n}_{a:\iota}, \mathbf{m}_{a:j}, \{\iota = j\})\}$ will be the explicit channel environment $\{(\mathbf{n}_{a:i}, \mathbf{m}_{a:i}) \mid i \geq 1\}$.

The formal definition of concretisation

$$\gamma : \tilde{\mathcal{K}} \times \tilde{\mathcal{R}} \rightarrow \hat{\mathcal{K}} \times \hat{\mathcal{R}}$$

takes the form $\gamma(\tilde{K}, \tilde{R}) = (\gamma_1(\tilde{K}), \gamma_2(\tilde{R}))$. We first define an auxiliary function, γ'_1 , which is, for simplicity, only defined on tuples of length 2 of symbolic names (instead of sequences of arbitrary length):

$$\begin{aligned} \gamma'_1(n^{\tilde{e}}, m^{\tilde{f}}, C) &= \{(n^e, m^f) \mid e, f \in (\mathbf{IId} \times \mathbb{N})^*, \tilde{e} \# e, \tilde{f} \# f, \\ &\quad \forall \beta : \forall (\iota = j) \in C : \tilde{e}.\beta = \iota \wedge \tilde{f}.\beta = j \Rightarrow e.\beta = f.\beta\} \end{aligned}$$

Table 6. The acceptability judgement, $(\tilde{K}, \tilde{R}) \Vdash^g P$, of the *symbolic* analysis

$$\begin{array}{c}
\frac{\forall i \leq k : (\tilde{K}, \tilde{R}) \Vdash^g \pi_i \wedge (\tilde{K}, \tilde{R}) \Vdash^g P_i}{(\tilde{K}, \tilde{R}) \Vdash^g \sum_{i=1}^k \pi_i \cdot P_i} \qquad \frac{(\tilde{K}, \tilde{R}) \Vdash^g P_1 \quad (\tilde{K}, \tilde{R}) \Vdash^g P_2}{(\tilde{K}, \tilde{R}) \Vdash^g P_1 \mid P_2} \\
\\
\frac{(\tilde{K}, \tilde{R}) \Vdash^g P}{(\tilde{K}, \tilde{R}) \Vdash^g (\nu n) P} \qquad (\tilde{K}, \tilde{R}) \Vdash^g \tau \qquad \frac{(\tilde{K}, \tilde{R}) \Vdash^{g\beta:??} P}{(\tilde{K}, \tilde{R}) \Vdash^g !^{\beta:k} P} \\
\\
\frac{\begin{array}{l} \forall \tilde{g}, \tilde{f}, \tilde{f}_1, n_1^{\tilde{g}_1}, \dots, \tilde{f}_k, n_k^{\tilde{g}_k} : \\ \tilde{g} \# g \wedge \tilde{f} \# e \wedge \bigwedge_i \tilde{f}_i \# e_i \wedge \bigwedge_i (u_i^{\tilde{f}_i}, n_i^{\tilde{g}_i}, C_i) \in \langle \tilde{R} \rangle \\ \Rightarrow (n^{\tilde{f}}, n_1^{\tilde{g}_1}, \dots, n_k^{\tilde{g}_k}, C) \in \langle \tilde{K} \rangle \\ \text{where } C = \text{CLOSE}_{\{\tilde{f}, \tilde{g}_1, \dots, \tilde{g}_k\}} (\bigcup_i C_i \cup \tilde{f} \doteq e\{\tilde{g}/g\} \cup \bigcup_i \tilde{f}_i \doteq e_i\{\tilde{g}/g\}) \end{array}}{(\tilde{K}, \tilde{R}) \Vdash^g n^e \langle u_1^{e_1}, \dots, u_k^{e_k} \rangle} \\
\\
\frac{\begin{array}{l} \forall \tilde{g}, \tilde{f}, \tilde{f}_1, n_1^{\tilde{g}_1}, \dots, \tilde{f}_k, n_k^{\tilde{g}_k} : \\ \tilde{g} \# g \wedge \tilde{f} \# e \wedge \bigwedge_i \tilde{f}_i \# e_i \wedge (n^{\tilde{f}}, n_1^{\tilde{g}_1}, \dots, n_k^{\tilde{g}_k}, C) \in \langle \tilde{K} \rangle \\ \Rightarrow \bigwedge_i (u_i^{\tilde{f}_i}, n_i^{\tilde{g}_i}, C_i) \in \langle \tilde{R} \rangle \\ \text{where } C_i = \text{CLOSE}_{\{\tilde{f}_i, \tilde{g}_i\}} (C \cup \tilde{f} \doteq e\{\tilde{g}/g\} \cup \tilde{f}_i \doteq e_i\{\tilde{g}/g\}) \end{array}}{(\tilde{K}, \tilde{R}) \Vdash^g n^e \langle x_1^{e_1}, \dots, x_k^{e_k} \rangle}
\end{array}$$

Here we used $\tilde{e} \# e$ to indicate that the sequence of round identifiers occurring before the colon's are equal in \tilde{e} and e , e.g. $(\beta : \iota) \# (\beta : 7)$. Next, the mapping γ_1 is the extension of γ'_1 :

$$\gamma_1(\tilde{K}) = \bigcup_{(n^{\tilde{e}}, m^{\tilde{f}}, C) \in \tilde{K}} \gamma'_1(n^{\tilde{e}}, m^{\tilde{f}}, C)$$

The extension to arbitrary sequences of symbolic names and the definition of γ_2 are straightforward.

Acceptability Judgement. The acceptability judgement, defined by Table 6, takes the form

$$(\tilde{K}, \tilde{R}) \Vdash^g P$$

where P is assumed to be *strongly well-formed* and where the round expression, $g \in (\mathbf{Id} \times \{?\})^*$, on the double turnstile records the replications met during the traversal of the syntax tree of the analysed program.

Intuitively, the idea is to “push” the generation of the infinitely many proof obligations from the rule for replication down to the “leaves” of the process, where they are dealt with symbolically. Hence, whenever we pass a replication, it is recorded in the turnstile annotation, yielding the replication context, in which a program is analysed. Consequently, the rules for τ , summation, parallel

composition and name restriction are much as in the explicit case. This leaves us with the axiom schemes for output and input.

To deal with output and input independent of particular choices of constraint identifiers, we define the closure, $\langle \tilde{R} \rangle$, under consistent renaming of constraint identifiers; furthermore, it contains all pairs of equal symbolic names (just like for $\langle \hat{R} \rangle$ in the explicit case). Formally, we have

$$\langle \tilde{R} \rangle = \{(x^{\tilde{e}\theta}, n^{\tilde{f}\theta}, C\theta) \mid (x^{\tilde{e}}, n^{\tilde{f}}, C) \in \tilde{R}, \quad \theta : \mathbf{CId} \rightarrow \mathbf{CId}\} \\ \cup \{(n, n, \emptyset) \mid n \in \mathbf{CName}\}$$

where $\theta : \mathbf{CId} \rightarrow \mathbf{CId}$ denotes a substitution of symbolic constraint identifiers for symbolic constraint identifiers; this corresponds to the fact that all symbolic constraint identifiers occurring in $(x^{\tilde{e}}, n^{\tilde{f}}, C)$ are in fact *implicitly* universally quantified.

In the case of output, we transfer data from the environment to the channel environment by looking up recorded bindings of symbolic variables to symbolic names. We do this by determining all potential bindings of variable (or name) identifiers u_i in $\langle \tilde{R} \rangle$ and putting the corresponding tuples into \tilde{K} . To ensure that \tilde{K} remains finite we do not add a tuple to \tilde{K} if it is a consistent renaming of a tuple already in \tilde{K} . To express this succinctly we define

$$\langle \tilde{K} \rangle = \{(m^{\tilde{e}\theta}, \dots, n^{\tilde{f}\theta}, C\theta) \mid (m^{\tilde{e}}, \dots, n^{\tilde{f}}, C) \in \tilde{K}, \quad \theta : \mathbf{CId} \rightarrow \mathbf{CId}\}$$

To construct the equality constraints we make use of the \doteq operator that tracks relations among constraint identifiers of symbolic round expressions:

$$\tilde{e} \doteq \tilde{f} = \{\iota = j \mid \exists \beta : \tilde{e}.\beta = \iota \wedge \tilde{f}.\beta = j\}$$

The symbolic substitution $\tilde{\Theta} = \{\tilde{g}/g\}$ is defined much as for the explicit case: $\tilde{e}\tilde{\Theta}$ denotes the result of replacing all occurrences of $\beta : ?$ in \tilde{e} by $\beta : \iota$, whenever $\tilde{g}.\beta = \iota$. It takes care of tracking equalities across the channel name and the output names/variables. Since the quantification over round expressions may have introduced a number of auxiliary variables, we reduce the constraint set C tracked in \tilde{K} to those constraint identifiers actually occurring in the symbolic round expressions of the looked up names. To this end, we define the closure operator, $\text{CLOSE}_{\{\tilde{e}\dots\tilde{f}\}}(C)$, as the combination of symmetric, transitive closure on the relation on constraint identifiers induced by C and the restriction to constraint identifiers in $\tilde{e}\dots\tilde{f}$. Informally, we set

$$\text{CLOSE}_{\{\tilde{e}\dots\tilde{f}\}}(C) = (C^{+,sym}) \cap (\text{ind}(\tilde{e}\dots\tilde{f}) \times \text{ind}(\tilde{e}\dots\tilde{f}))$$

where $\text{ind}(\tilde{e}\dots\tilde{f}) = \{\tilde{g}.\beta \mid \tilde{g} \text{ occurs in } \tilde{e}\dots\tilde{f}, \beta \in \mathbf{IId}\}$ is the set of constraint identifiers occurring in $\tilde{e}\dots\tilde{f}$.

The case of input works dually to the output transferring data from the symbolic channel environment to the symbolic environment, where the same operations on round expressions are applied.

Correctness Properties. We first prove a Soundness Theorem (Theorem 2 below) that relates the explicit and the symbolic type system using concretisation as defined above. Together with the Subject Reduction Theorem of the explicit type system (Theorem 1 of Section 3) it establishes the soundness of the symbolic system.

Theorem 2 (Soundness). *Let P be a strongly well-formed process and (\tilde{K}, \tilde{R}) an analysis estimate such that $(\tilde{K}, \tilde{R}) \Vdash^\epsilon P$. We then have $\gamma(\tilde{K}, \tilde{R}) \Vdash P$.*

Proof. We prove the following stronger result by structural induction in P . Let P be a strongly well-formed process, $g \in (\mathbf{Id} \times \{?\})^*$ a round expression, and let (\tilde{K}, \tilde{R}) be chosen such that $(\tilde{K}, \tilde{R}) \Vdash^g P$. We then have $\gamma(\tilde{K}, \tilde{R}) \Vdash P\{^h/g\}$ for all $h \in (\mathbf{Id} \times \mathbb{N})^*$ such that $h\#g$. \square

We next prove a Type Adequacy Result showing that our symbolic analysis is able to ensure *round adequacy* as defined in Definition 1.

Theorem 3 (Adequacy). *Let P be a strongly well-formed process and let (\tilde{K}, \tilde{R}) be chosen such that $(\tilde{K}, \tilde{R}) \Vdash^\epsilon P$. If for all $(x^{\tilde{g}}, n^{\tilde{f}}, C) \in \tilde{R}$ it holds that $(\tilde{g}.\beta = \tilde{f}.\beta) \in C$ then P is β -round-consistent.*

Proof. Let P and (\tilde{K}, \tilde{R}) be as stated and let $P', P'', \mathbf{n}^e, \mathbf{x}^f$ be arbitrary such that $P \longrightarrow^* P' \longrightarrow P''$, where $P' \longrightarrow P''$ is due to a communication involving $\mathbf{n}^e(\mathbf{x}^f)$ and $\mathbf{n}^e\langle \mathbf{m}^g \rangle$. As we assume $(\tilde{K}, \tilde{R}) \Vdash^\epsilon P$, we can apply Theorem 2 to obtain $\gamma(\tilde{K}, \tilde{R}) \Vdash P$ and, by Theorem 1, $\gamma(\tilde{K}, \tilde{R}) \Vdash \lfloor P' \rfloor$. As $P' \longrightarrow P''$ is due to a communication involving $\mathbf{n}^e(\mathbf{x}^f)$ and $\mathbf{n}^e\langle \mathbf{m}^g \rangle$, this implies

$$\gamma(\tilde{K}, \tilde{R}) \Vdash \lfloor \mathbf{n}^e(\mathbf{x}^f) \rfloor \quad (1)$$

$$\gamma(\tilde{K}, \tilde{R}) \Vdash \lfloor \mathbf{n}^e\langle \mathbf{m}^g \rangle \rfloor \quad (2)$$

By (2), we can deduce $\lfloor \mathbf{n}^e\langle \mathbf{m}^g \rangle \rfloor \in \gamma_1(\tilde{K})$, which, by (1), leads to $\lfloor (\mathbf{x}^f, \mathbf{m}^g) \rfloor \in \gamma_2(\tilde{R})$. Given the assumption about $(x^{\tilde{g}}, n^{\tilde{f}}, C) \in \tilde{R}$, we can thus conclude $f.\beta = g.\beta$ establishing β -round consistency of P . \square

Running Example. Recall the processes P_1 and P_2 of our running example. The least symbolic analysis results that are acceptable for these processes are:

$$\begin{aligned} \tilde{K}_1 &= \{(\mathbf{n}_{a:l}, \mathbf{m}_{b:j}, \{\iota = j\})\} & \tilde{R}_1 &= \{(\mathbf{x}_{a:l}, \mathbf{m}_{b:j}, \{\iota = j\})\} \\ \tilde{K}_2 &= \{(\mathbf{n}_\epsilon, \mathbf{m}_{a:l}, \emptyset)\} & \tilde{R}_2 &= \{(\mathbf{x}_{a:l}, \mathbf{m}_{a:j}, \emptyset)\} \end{aligned}$$

Using the adequacy result of Theorem 3, this shows the \mathbf{a} -round consistency of the process P_1 .

5 The Diffie-Hellman Key Agreement Protocol

As an application we shall consider the Diffie-Hellman Key Agreement protocol [7]. It assumes two principals A and B that want to establish a shared secret.

They do so by first agreeing publicly on an element g . Then they individually select random values r_A and r_B , calculate the values g^{r_A} and g^{r_B} and exchange them; after that they will share the secret $g^{r_A r_B} (= (g^{r_A})^{r_B} = (g^{r_B})^{r_A})$.

We can easily encode this simple protocol in our calculus. To do so we assume the existence of two global channels c_{AB} and c_{BA} used for communication between the principals. We shall first consider a variant of the protocol where A and B establish secrets in a number of rounds using different values of g :

$$(\nu c_{AB}^\epsilon) (\nu c_{BA}^\epsilon) !^{a:1} (\nu g^{a:?:}) \quad (\nu r_A^{a:?:}) (c_{AB}^\epsilon \langle (g^{a:?:})^{r_A^{a:?:}} \rangle \mid c_{BA}^\epsilon (x_A^{a:?:})) \\ \mid (\nu r_B^{a:?:}) (c_{BA}^\epsilon \langle (g^{a:?:})^{r_B^{a:?:}} \rangle \mid c_{AB}^\epsilon (x_B^{a:?:}))$$

An alternative version of the protocol uses the same value of g for all the rounds

$$(\nu c_{AB}^\epsilon) (\nu c_{BA}^\epsilon) (\nu g^\epsilon) !^{a:1} (\nu r_A^{a:?:}) (c_{AB}^\epsilon \langle (g^\epsilon)^{r_A^{a:?:}} \rangle \mid c_{BA}^\epsilon (x_A^{a:?:})) \\ \mid (\nu r_B^{a:?:}) (c_{BA}^\epsilon \langle (g^\epsilon)^{r_B^{a:?:}} \rangle \mid c_{AB}^\epsilon (x_B^{a:?:}))$$

Protocols where the principals need to generate several consecutive shared secrets – such as the Diffie-Hellmann Key Agreement Protocol – demonstrate the usefulness of, and the need for, the notion of round-consistency. It is easy to see, that round-consistency ensures the *absence of replay attacks*. In fact, our analysis is able to establish a -round consistency for the first version of the protocol. However, the second version is not a -round consistent, and indeed, a replay attack exists exploiting the “global” channel g .

The probably most well-known attack on the Diffie-Hellmann protocol is a man-in-the-middle attack, which we can detect, too. In order to do so, we need to encode potential attackers into our model of the protocol as was done for the π -calculus in [5]. This is, however, a standard technique not using round information and we refrain from elaborating on it here.

6 Conclusion

Summary. In this paper we have shown how to develop a value passing CCS with explicit information about rounds. We introduced the notion of *round consistency* to formally capture well-formed behaviour of round-based protocols/processes. Based on this we developed a type system for making an *explicit* record of all communication (including round information) that can take place in the system proving subject reduction as usual. To cater for the possibility of implementing the system we also developed a type system giving a *symbolic* record — and showed this to be a correct overapproximation to the explicit system in Theorem 2. Our key result shows that the symbolic analysis is indeed able to statically determine the round consistency of a protocol. We illustrated the usefulness of this analysis on an example: a communication protocol based on *Diffie-Hellman* key exchange, where also the notion of round consistency is naturally motivated as characterising the absence of replay attacks.

Perspective. The present research is part of an initiative to bring more powerful static analysis techniques within the reach of being applied to process calculi. In particular to extend the current repertoire of techniques in the Flow Logic approach [16,17].

A lot of inspiration for the overall approach comes from Type Systems. The use of constraints as part of types is reminiscent of [19], where polymorphic languages with subtyping are studied. Moreover, our $\text{CLOSE}_{\{\cdot\}}(\cdot)$ operator reminds of the transitive reduction of [19]. Similar techniques are used also in analyses based on Type and Effect Systems [13].

The work by Venet [20] and later by Feret [8] established a benchmark in the world of static analysis of process calculi using techniques from Abstract Interpretation. In fact, Feret’s approach works on a large subset of the π -calculus and is able to infer strongly relational properties, including sophisticated numerical domains. These are properties that are out of scope of our approach presented here. Nonetheless, we claim a number of benefits of our approach as compared to theirs:

- **Portability:** We rely on a fully standard syntax of the considered language. In contrast, guarded replication is a syntactic restriction used by [8], which considerably simplifies reasoning about the relation among replication instances. It works by tying together reception and replication, such that a new instance is only unfolded on reception of a message. We deal with the more general approach, where all instances are implicitly there at any time. Furthermore, instead of intricate instrumented semantics, which is the basis of the abstract interpretation in [8], our analysis works for the standard reaction semantics. This enables us to transfer our results much more easily to other calculi (as validated in our work [14] on adapting an analysis for π to a Fusion like [18] calculus). In this work, we show that also round information can be conservatively added to existing analyses testifying to the flexibility of our approach.
- **Lightweight:** In contrast to heavy abstract interpretation machinery using sophisticated abstract domains, a simple type-based, syntax-driven scheme suffices for our approach.
- **Separation of Concerns:** The analysis in [8], is full-fledged in terms of being *relational* and *numerical*. This generates quite some complexity. We chose the opposite approach of separating dimensions. Due to the use of our syntax-driven approach, it becomes a mere technical exercise of combining the various dimensions. We have explored the “relational” dimension in [14] and the dimension incurred by fusion-like name binding in [2]. Here, we add yet another dimension, rounds.
- **Implementation:** Additionally, separation of concerns facilitates the implementation of analyses. Code can be re-used easily, and specification and solutions of analyses are separated enabling the use of off-the-shelf solvers, e.g., the Succinct Solver [15].

Future Work. Looking at our concrete application, that of communication protocols, the notion of round consistency as defined here is a rather strong requirement. Therefore one may consider to relax it to reason about particular name

and variable identifiers at particular program points only. Similarly, one may want to impose round-consistency only on subprocesses of the overall process.

Another possible dimension is the inclusion of *terms* instead of names. This will allow for the more precise modelling and analysis of security protocols, very much in the spirit of analyses of the LySa calculus [4]. Our ultimate goal is thus to present a general framework that allows for the automatic combination of dimensions for a given calculus. To this end we would target modern service-oriented languages like COWS [10], possibly equipped with terms to encode security.

We believe that lightweight tailored solutions that are freely combinable produce more (re-)usable, understandable, and reliable results than a fully integrated general purpose approach. Indeed, simple analyses in terms of only one dimension may suffice for many applications avoiding the overhead of a full-blown analysis like the one in [8]. The long term goal is to ensure that the methods developed here will scale up to more process calculi embodying different and more complex computational paradigms, including Actors, Obliq, Klaim and COWS.

Acknowledgement. This work has been partially supported by MT-LAB, a VKR Centre of Excellence.

References

1. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for Actor computation. *J. Funct. Program.* 7(1), 1–72 (1997)
2. Bauer, J., Nielson, F., Riis Nielson, H., Pilegaard, H.: Relational analysis of correlation. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 32–46. Springer, Heidelberg (2008)
3. Berry, G., Boudol, G.: The chemical abstract machine. *Theor. Comput. Sci.* 96(1), 217–248 (1992)
4. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Riis Nielson, H.: Static validation of security protocols. *J. Comput. Secur.* 13(3), 347–390 (2005)
5. Bodei, C., Degano, P., Nielson, F., Riis Nielson, H.: Static analysis for the π -calculus with applications to security. *Information and Computation* 168, 68–92 (2001)
6. Cardelli, L.: A language with distributed scope. In: POPL, pp. 286–297 (1995)
7. Diffie, W., Hellman, M.E.: New directions in cryptography. *IEEE Transactions on Information Theory* IT-22(6), 644–654 (1976)
8. Feret, J.: Dependency analysis of mobile systems. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 314–329. Springer, Heidelberg (2002)
9. Karr, M.: Affine relationships among variables of a program. *Acta Inf.* 6, 133–151 (1976)
10. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
11. Milner, R.: *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press (1999)

12. Nicola, R.D., Gorla, D., Hansen, R.R., Nielson, F., Riis Nielson, H., Probst, C.W., Pugliese, R.: From flow logic to static type systems for coordination languages. *Sci. Comput. Program.* 75(6), 376–397 (2010)
13. Nielson, F., Riis Nielson, H.: Type and Effect Systems. In: Olderog, E.-R., Steffen, B. (eds.) *Correct System Design*. LNCS, vol. 1710, pp. 114–136. Springer, Heidelberg (1999)
14. Nielson, F., Riis Nielson, H., Bauer, J., Rosenkilde Nielsen, C., Pilegaard, H.: Relational analysis for delivery of services. In: Barthe, G., Fournet, C. (eds.) *TGC 2007*. LNCS, vol. 4912, pp. 73–89. Springer, Heidelberg (2008)
15. Nielson, F., Riis Nielson, H., Sun, H., Buchholtz, M., Hansen, R.R., Pilegaard, H., Seidl, H.: The Succinct Solver Suite. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 251–265. Springer, Heidelberg (2004)
16. Riis Nielson, H., Nielson, F.: Flow Logic: a multi-paradigmatic approach to static analysis. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) *The Essence of Computation*. LNCS, vol. 2566, pp. 223–244. Springer, Heidelberg (2002)
17. Riis Nielson, H., Nielson, F., Pilegaard, H.: Flow logic for process calculi. *ACM Computing Surveys* (to appear 2010)
18. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: *LICS*, pp. 176–185 (1998)
19. Smith, G.: Polymorphic type inference with overloading and subtyping. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) *TAPSOFT 1993*. LNCS, vol. 668, pp. 671–685. Springer, Heidelberg (1993)
20. Venet, A.: Automatic determination of communication topologies in mobile systems. In: Levi, G. (ed.) *SAS 1998*. LNCS, vol. 1503, pp. 152–167. Springer, Heidelberg (1998)

Abstract LR-Parsing

Kyung-Goo Doh^{1,*}, Hyunha Kim^{1,*}, and David A. Schmidt^{2,**}

¹ Hanyang University, Ansan, South Korea

² Kansas State University, Manhattan, Kansas, USA

Abstract. We explain and illustrate *abstract parsing*, a static-analysis technique based on abstract interpretation, LR-parsing, and partial evaluation for validating PHP-like scripts that generate HTML/XML-style documents. A validated script is guaranteed to generate documents that are well formed with respect to the document language’s LR(k)-grammar. In this way, abstract parsing resembles compiler data-type checking: a validated script will “not go wrong” and output a malformed, dynamically generated document.

After presenting abstract parsing for LR(k)-grammars, we handle these important extensions: (i) String-replacement operations are analyzed by composing the finite-state automaton defined by a string replacement with the finite-state control of the LR(k)-parser. (ii) Conditional-test expressions are implemented by filter automata, which are also composed with the parser’s finite-state control. (iii) Dynamically supplied and potentially malicious user input is predicted by characterizing it with an LR(k)-grammar and analyzing the strings generated by the grammar. (iv) Synthesized-attribute grammars are employed to calculate the semantics of the dynamically generated documents.

1 Introduction

Scripting languages use strings as a “universal data structure” to communicate documents, data structures, and programs. For example, a PHP script might assemble within one long string an entire HTML page or an XML document or an SQL query. An incorrectly assembled string-document might later cause failure when it is supplied as input to its intended processor (a web browser or database engine). Worse still, the string-document might contain textual input supplied by a malicious user and initiate a cross-site-scripting or injection attack [20].

To prevent such failures and attacks, the well-formedness of dynamically generated string-documents should be checked with respect to the document’s

* doh@hanyang.ac.kr Supported by R01-2006-000-10926-0, the Basic Research Program of the Korea Science and Engineering Foundation and by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/Korea Science and Engineering Foundation(KOSEF) R11-2008-007-01003-0.

** das@ksu.edu Supported by NSF CNS-0939431.

context-free *reference grammar* (for HTML or XML or SQL) before the string-document is supplied to its processor. Better still, the document generator script *itself* should be analyzed to validate that all its generated string-documents are well formed with respect to the reference grammar, much like an application program is type checked in advance of execution.

In this paper, we marry techniques from LR-parsing theory, abstract interpretation, and partial evaluation to formulate a static analysis that validates that the string-documents generated by a script are grammatically well formed with respect to the reference grammar. We call the analysis *abstract parsing* because it is an abstract interpretation of the script conducted simultaneously with the LR-parsing of the string-documents generated by the script.

The meaning of each potentially generated string-document is not a set of strings or a regular expression but is (an approximation of) the *parse stack* that the LR-parser would generate when it parsed the string-document — the parse stack encodes both the string and its context-free structure, thus providing greater precision than techniques that approximate the string via regular expressions.

The paper proceeds as follows. After presenting abstract parsing for LR(0) and general LR(k) grammars, we handle these important extensions:

- String-replacement operations are analyzed by *composing* the finite-state automaton defined by a string replacement with the finite-state control of the LR(k)-parser.
- Conditional-test expressions are implemented by filter automata, which are also composed with the parser’s finite-state control.
- Dynamically supplied, potentially malicious, user input is predicted and processed by characterizing it by an LR(k)-grammar and analyzing the strings generated from by grammar.
- Attribute grammar technology is added to calculate the semantic properties of dynamically generated string output.

2 Background Example

Say that a script must generate output strings that conform to this grammar,

$$S \rightarrow a \mid [S]$$

where S is the only nonterminal. (HTML, XML, and SQL are such bracket languages.) The grammar is LR(0), but it can be difficult to enforce even for simple programs, like the one in Figure 1, left column. Perhaps the example program should print only well-formed S -phrases — the occurrence of x at “`print x`” is a “hot spot,” where we must analyze x ’s possible values. Three approaches have been proposed to do this:

1. An analysis based on *type checking* assigns types (reference-grammar nonterminals) to the program’s variables and uses them to validate that the program is well typed. The occurrences of x should be data-typed as S , but r has no data type that corresponds to a nonterminal.

<code>x = 'a'</code>	$X0 = a$
<code>r = ']'</code>	$R =]$
<code>while ...</code>	$X1 = X0 \sqcup X2$
<code>x = '[' . x . r</code>	$X2 = [\cdot X1 \cdot R$
<code>print x</code>	$X3 = X1$

(Read `.` as an infix string-append operation.)

Fig. 1. Sample program and its flow equations

2. An analysis based on *regular expressions* [2–6, 14, 15, 19] solves the program-flow equations shown in Figure 1’s right column in the domain of regular expressions, determining that the hot spot’s ($X3$ ’s) values conform to the regular expression, $[* \cdot a \cdot]^*$, but this does not validate the assertion. Improvement in precision can be obtained with *parenthesis grammars* [13, 17], which generate good regular-expression approximations of simple bracket grammars but fail to express general context-free structure.
3. A *grammar-based analysis* [18] treats the flow equations as a set of grammar rules, and a language-inclusion check tries to prove that all $X3$ -generated strings are S -generable. A useful instance of this technique is due to Møller and Schwarz, who check language inclusion with the more restrictive but useful SGML DTD [9] for HTML documents [16].

Our approach solves the program-flow equations in Figure 1 in the domain of *parse stacks* — $X3$ ’s meaning is the *set of parse stacks* of the strings that might be denoted by x . Our technique simultaneously unfolds and LR-parses the strings defined by $X3$, computing a parse stack that expresses both the structure in the flow equations and that of the reference grammar.

The technique is implemented by a partial-evaluation-style specialization of the program’s flow equations applied to the LR-parser. When the specialized, residual flow equations are “executed” (solved with a least-fixed semantics), they generate (sets of) parse stacks as their answers.

Of course, a program might generate infinitely many different strings and therefore the analysis might compute an infinite set of parse stacks. We finitely approximate an infinite set of parse stacks by exploiting this key feature of LR-parse theory: Each parse stack is exactly a finite path through the LR-parser’s finite-state control automaton and can be approximated by the smallest subgraph of the automaton that covers the path. The smallest-subgraph approximation is computed merely by *folding the parse stack on its repeating state(s)*.

3 Abstract LR(0)-Parsing

We present the technique via the example program in Figure 1. For the example grammar, $S \rightarrow a \mid [S]$, Figure 2 gives the LR(0)-parse-controller automaton and a parse of the string, $[[a]]$. The parse-controller automaton is presented

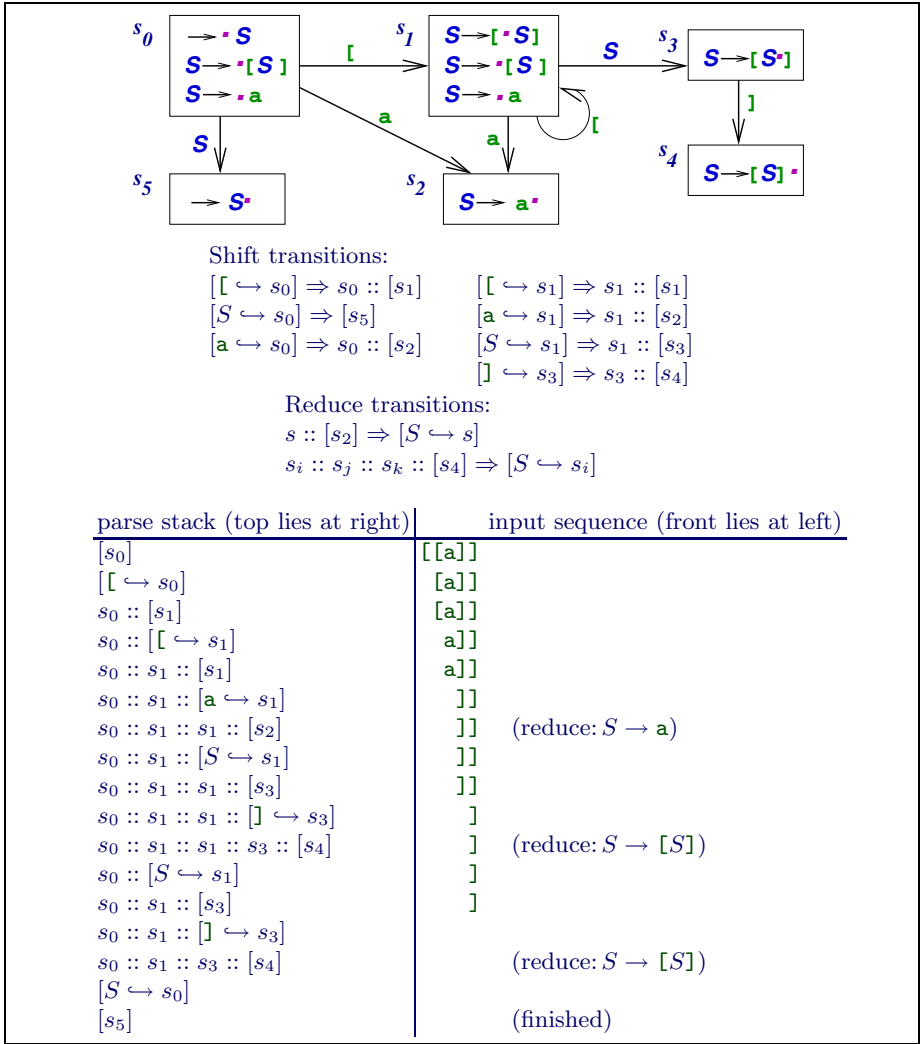


Fig. 2. Parse controller for $S \rightarrow [S] \mid a$ and an example parse of $[[a]]$

graphically, and its transitions are coded as shift/reduce rewriting rules, which we use to parse the string. The current state, $[s_i]$, of the parse appears as the top state in the parse stack, $s_0 :: s_1 :: \dots :: [s_i]$. Input symbols, i , are supplied to state, s , in the format, $[i \hookrightarrow s]$. The parser’s start state is $[s_0]$.

Say that we must validate that the program in Figure 1 prints only S -structured phrases. To analyze the program’s hot spot at $X3$, we must $LRparse(X3, s_0)$, which we portray as a function call, $X3[s_0]$ — we treat the program-flow equations in Figure 1 as functions defined in combinator notation and we specialize (apply) a flow equation to the state used to parse it.

The flow equation, $X3 = X1$, generates this call step:

$$X3[s_0] = X1[s_0]$$

which demands a parse of the strings generated at point $X1$ from parse state s_0 :

$$X1[s_0] = X0[s_0] \cup X2[s_0]$$

The union of the parses of strings at $X0$ and $X2$ from s_0 must be computed. (*Important:* this computes a set of parse stacks. In this example, all the sets are singletons, and we omit the set braces to reduce notational clutter.) We consider first $X0[s_0]$:

$$X0[s_0] = \mathbf{a}[s_0] = [\mathbf{a} \hookrightarrow s_0] \Rightarrow s_0 :: [s_2] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5].$$

That is, a parse of ' \mathbf{a} ' from s_0 generates the one-element stack, s_5 (actually, $\{[s_5]\}$) — all strings denoted by $X0$ are S -phrases. Next,

$$\begin{aligned} X2[s_0] &= ([\cdot X1 \cdot R] [s_0]) = [[\hookrightarrow s_0] \oplus (X1 \cdot R)] \\ &\Rightarrow (s_0 :: [s_1]) \oplus (X1 \cdot R) \\ &= s_0 :: (X1 \cdot R)[s_1] = s_0 :: (X1[s_1] \oplus R) \end{aligned}$$

The \oplus is a “continuation operator”: For parse stack, st , and combinator expression, E , define $st \oplus E = tail(st) :: E[head(st)]$. That is, stack st 's top state feeds to E . (More generally, for a set of stacks, S , define $S \oplus E = \{tail(st) :: E[head(st)] \mid st \in S\}$.)

Next, $X1[s_1] = X0[s_1] \cup X2[s_1]$ computes to $s_1 :: [s_3]$ (as explained below, the recursion generated by $X2[s_1]$ is resolved by least-fixed-point iteration), so

$$\begin{aligned} X2[s_0] &= s_0 :: (X1[s_1] \oplus R) = (s_0 :: s_1 :: [s_3]) \oplus R = s_0 :: s_1 :: R[s_3] = s_0 :: s_1 :: [\hookrightarrow s_3] \\ &\Rightarrow s_0 :: s_1 :: s_3 :: [s_4] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5] \end{aligned}$$

That is, $X2[s_0]$ built the stack, $s_0 :: s_1 :: s_3 :: [s_4]$, denoting a parse of $[S]$, which reduced to S , giving s_5 .

Here is the list of residual equations generated from the partial evaluation of the initial call, $X3[s_0]$:

$$\begin{aligned} X3[s_0] &= X1[s_0] \\ X1[s_0] &= X0[s_0] \cup X2[s_0] \\ X0[s_0] &= [s_5] \\ X2[s_0] &= s_0 :: (X1[s_1] \oplus R) \\ X1[s_1] &= X0[s_1] \cup X2[s_1] \\ X0[s_1] &= s_1 :: [s_3] \\ X2[s_1] &= s_1 :: (X1[s_1] \oplus R) \\ R[s_3] &= s_3 :: [s_4] \quad (\text{generated while } X2[s_1] \text{ is solved}) \end{aligned}$$

Each $X_i[s_j] = E_{ij}$ is a *first-order equation* whose answer is a set of parse stacks.

The equations for $X1[s_1]$ and $X2[s_1]$ are mutually recursively defined, and their solutions are computed by least-fixed-point iteration. Here are the solutions:

$$\begin{aligned} X1[s_1] &= X0[s_1] \cup X2[s_1] = (s_1 :: [s_3]) \cup (s_1 :: [s_3]) = s_1 :: [s_3] \\ X2[s_1] &= s_1 :: (X1[s_1] \oplus R) \Rightarrow s_1 :: s_1 :: R[s_3] \Rightarrow s_1 :: [s_3] \\ X2[s_0] &= s_0 :: (X1[s_1] \oplus R) \Rightarrow s_0 :: s_1 :: R[s_3] = s_0 :: s_1 :: s_3 :: [s_4] \Rightarrow [s_5] \\ X1[s_0] &= X0[s_0] \cup X2[s_0] = [s_5] \cup [s_5] = [s_5] \end{aligned}$$

$X3[s_0] = X1[s_1] = [s_5]$ validates that the strings printed at the hot spot must be S -phrases. (Note again: these answers are really sets, that is, $X3[s_0] = \{\{s_5\}\}$.) The algorithm that generates the residual equations and simultaneously solves them is a worklist algorithm like those used for demand-driven data-flow analyses [1, 8, 10]; it also resembles *minimal function-graph semantics* [12].

Figure 3 shows the worklist algorithm applied to the example. The algorithm uses three data structures: the worklist of unresolved calls, $Xi[s_j]$; a *Cache* (“seen-before list”) that maps each call to its current (partial) solution (a set of abstract parse stacks); and the graph of call dependencies, which is dynamically constructed.

The initialization step places initial call, $X0[s_0]$, into the worklist and into the dependency graph and assigns to the cache the partial solution, $Cache[X0[s_0]] = \emptyset$. The iteration step repeats the following until the worklist is empty:

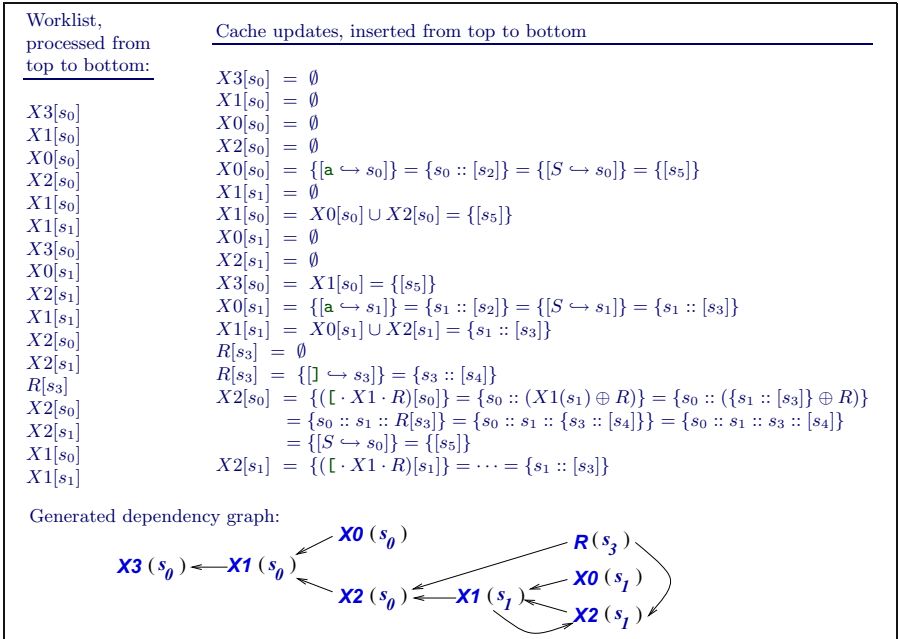


Fig. 3. Worklist-algorithm calculation of call, $X3[s_0]$, in Figure 1

- Extract the front call, $X[s]$, from the worklist, and for its corresponding flow equation, $X = E$, compute $E[s]$, a set, using the parser’s shift/reduce rules:
 1. While computing $E[s]$, if a call, $X'[s']$ is encountered, (i) add the dependency, $X'[s'] \rightarrow X[s]$, to the dependency graph (if not already present); (ii) if there is no entry for $X'[s']$ in the cache, then assign $Cache[X'[s']] = \emptyset$ to the cache and add $X'[s']$ to the end of the worklist; (iii) use $Cache[X'[s']]$ as the meaning of $X'[s']$ in the computation of $E[s]$.
 2. When $E[s]$ computes to an answer set, P , and P contains a parse stack not already listed in $Cache[X[s]]$, then set $Cache[X[s]] = Cache[X[s]] \cup P$ and add to the end of the worklist all $X''[s'']$ such that $X[s] \rightarrow X''[s'']$ appears in the dependency graph.

4 Abstract Parse Stacks

In the previous example, the result for each $X_i[s_j]$ was a single stack. In general, a set of parse stacks can result, e.g., for

$x = \text{'[}'$	$X0 = [$
$\text{while } \dots$	$X1 = X0 \sqcup X2$
$x = x \cdot \text{'[}'$	$X2 = X1 \cdot [$
$x = x \cdot \text{'a' } \cdot \text{'}]'$	$X3 = X1 \cdot a \cdot]$

at conclusion, x holds zero or more left brackets and an S -phrase, and $X3[s_0]$ is the infinite set, $\{[s_5], s_1 :: [s_3], s_1 :: s_1 :: [s_3], s_1 :: s_1 :: s_1 :: [s_3], \dots\}$.

To bound the set, we abstract it by “folding” its stacks so that no state repeats in a stack. A stack segment like $s_1 :: s_1 :: [s_3]$ is a graph, $\leftarrow s_1 \leftarrow s_1 \leftarrow [s_3] \leftarrow$;

the folded stack merges identical states: $\leftarrow s_1 \leftarrow [s_3] \leftarrow$. Since the set of parse-state names is finite, folding produces a finite set of finite-sized stacks (that contain cycles). For the previous example, the worklist algorithm calculates $X3[s_0] = \{[s_5], s_1^+ :: [s_3]\}$. As noted earlier, each parse stack is a finite path through the LR-parser’s finite-state controller automaton, and folding the parse stack generates the smallest subgraph of the automaton that covers the path.

Stack folding can be profitably delayed when straightline code is analyzed, so we fold stacks *only if* there is backwards control flow: When calculating a call, $X_i[s_i] = \dots X_j[s_j] \dots$, if $X_j \rightarrow X_i$ is a “back arc” in the program’s control flow (that is, $j \geq i$), *only then* we fold the set of stacks defined by $X_j[s_j]$ to compute $X_i[s_i]$. This way, we lose precision exactly when the source program’s control flow itself loses precision. Again, finite convergence is guaranteed.

5 LR(k) Grammars Are Accommodated the Same Way

Abstract parsing also applies to LR(*k*) grammars, for *k* > 0. Figure 4 presents an LR(1) grammar, its controller, and an example parse. The parse states have

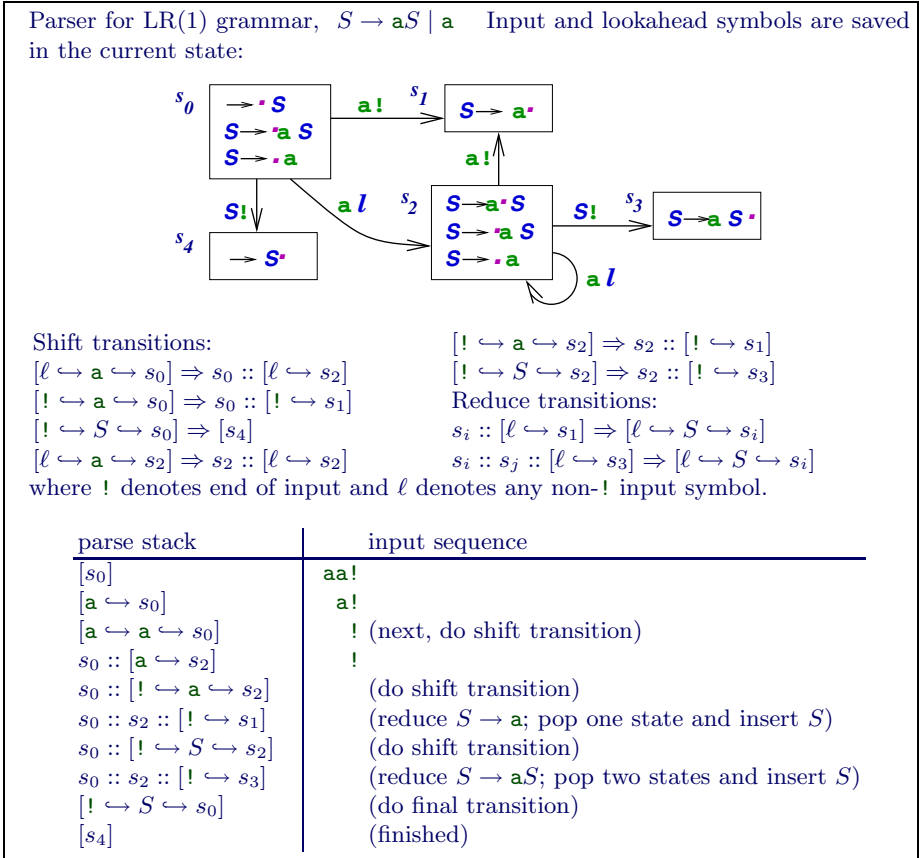


Fig. 4. An LR(*k*) grammar uses a state of form, $[\ell_k \hookrightarrow \ell_{k-1} \hookrightarrow \dots \hookrightarrow \ell_0 \hookrightarrow s]$

form, $[\ell_j \hookrightarrow \dots \hookrightarrow \ell_0 \hookrightarrow s]$, where $0 \leq j \leq k + 1$. When a program is statically parsed with an LR(*k*) grammar, *k* > 0, the first-order residual equations have form,

$$Xi[\ell_j \hookrightarrow \dots \hookrightarrow \ell_0 \hookrightarrow s] = E$$

for $0 \leq j \leq k$. (Alas, this means a residual-equation set of order $(k + 1)!$.) For this program,

```

x = 'a'
while ...
  x = 'a' . x . 'a'
print x !

```

$X0 = a$
 $X1 = X0 \sqcup X2$
 $X2 = a \cdot X1 \cdot a$
 $X3 = X1 \cdot !$

its partial evaluation proceeds as follows:

$$\begin{aligned}
X3[s_0] &= (X1 \cdot !)[s_0] = X1[s_0] \oplus ! \\
X1[s_0] &= X0[s_0] \cup X2[s_0] \\
X0[s_0] &= \mathbf{a}[s_0] = \{\mathbf{a} \hookrightarrow s_0\} \\
X2[s_0] &= (\mathbf{a} \cdot X1 \cdot \mathbf{a})[s_0] = \mathbf{a}[s_0] \oplus (X1 \cdot \mathbf{a}) \\
&= \{\{\mathbf{a} \hookrightarrow s_0\}\} \oplus (X1 \cdot \mathbf{a}) = \{X1[\mathbf{a} \hookrightarrow s_0] \oplus \mathbf{a}\} \\
X1[\mathbf{a} \hookrightarrow s_0] &= X0[\mathbf{a} \hookrightarrow s_0] \cup X2[\mathbf{a} \hookrightarrow s_0] \\
X0[\mathbf{a} \hookrightarrow s_0] &= \{\{\mathbf{a} \hookrightarrow s_0\}\} = \{\{\mathbf{a} \hookrightarrow \mathbf{a} \hookrightarrow s_0\}\} \Rightarrow \{s_0 :: [\mathbf{a} \hookrightarrow s_2]\} \\
X2[\mathbf{a} \hookrightarrow s_0] &= (\mathbf{a} \cdot X1 \cdot \mathbf{a})[\mathbf{a} \hookrightarrow s_0] = [\mathbf{a} \hookrightarrow \mathbf{a} \hookrightarrow s_0] \oplus (X1 \cdot \mathbf{a}) \\
&= \{s_0 :: [\mathbf{a} \hookrightarrow s_2]\} \oplus (X1 \cdot \mathbf{a}) = \{s_0 :: X1[\mathbf{a} \hookrightarrow s_2] \oplus \mathbf{a}\} \\
X1[\mathbf{a} \hookrightarrow s_2] &= X0[\mathbf{a} \hookrightarrow s_2] \cup X2[\mathbf{a} \hookrightarrow s_2] \\
X0[\mathbf{a} \hookrightarrow s_2] &= \mathbf{a}[\mathbf{a} \hookrightarrow s_2] = \{\{\mathbf{a} \hookrightarrow \mathbf{a} \hookrightarrow s_2\}\} = \{s_2 :: [\mathbf{a} \hookrightarrow s_2]\} \\
X2[\mathbf{a} \hookrightarrow s_2] &= (\mathbf{a} \cdot X1 \cdot \mathbf{a})[\mathbf{a} \hookrightarrow s_2] = [\mathbf{a} \hookrightarrow \mathbf{a} \hookrightarrow s_0] \oplus (X1 \cdot \mathbf{a}) \\
&= \{s_2 :: [\mathbf{a} \hookrightarrow s_2]\} \oplus (X1 \cdot \mathbf{a}) = \{s_2 :: (X1[\mathbf{a} \hookrightarrow s_2] \oplus \mathbf{a})\} \\
X1[\mathbf{a} \hookrightarrow s_2] &= \{s_2 :: [\mathbf{a} \hookrightarrow s_2]\} \cup \{s_2 :: (X1[\mathbf{a} \hookrightarrow s_2] \oplus \mathbf{a})\}
\end{aligned}$$

The residual equations are solved by least-fixed point calculation; $X1[\mathbf{a} \hookrightarrow s_2]$ computes to $\{s_2^i :: [\mathbf{a} \hookrightarrow s_2] \mid i \in 1, 3, 5, \dots\}$, which our analysis approximates by $\{s_2^+ :: [\mathbf{a} \hookrightarrow s_2]\}$. Using this result, we obtain

$$\begin{aligned}
X1[\mathbf{a} \hookrightarrow s_2] &= \{s_2^+ :: [\mathbf{a} \hookrightarrow s_2]\} \\
X2[\mathbf{a} \hookrightarrow s_0] &= \{s_0 :: s_2^+ :: [\mathbf{a} \hookrightarrow s_2]\} \\
X1[\mathbf{a} \hookrightarrow s_0] &= \{s_0 :: s_2^* :: [\mathbf{a} \hookrightarrow s_2]\} \\
X2[s_0] &= \{s_0 :: s_2^+ :: [\mathbf{a} \hookrightarrow s_2]\} \\
X1[s_0] &= \{\{\mathbf{a} \hookrightarrow s_0\}\} \cup \{s_0 :: s_2^+ :: [\mathbf{a} \hookrightarrow s_2]\} \\
X3[s_0] &= \{\{! \hookrightarrow \mathbf{a} \hookrightarrow s_0\}\} \cup \{s_0 :: s_2^+ :: \{! \hookrightarrow \mathbf{a} \hookrightarrow s_2\}\} = \{s_4\}
\end{aligned}$$

$$\begin{aligned}
&\text{since } \{\{! \hookrightarrow \mathbf{a} \hookrightarrow s_0\}\} = \{s_0 :: \{! \hookrightarrow s_1\}\} = \{\{! \hookrightarrow S \hookrightarrow s_0\}\} = \{s_4\} \\
&\text{and } \{s_0 :: s_2^+ :: \{! \hookrightarrow \mathbf{a} \hookrightarrow s_2\}\} = \{s_0 :: s_2^+ :: \{! \hookrightarrow s_1\}\} (s_2^+ :: s_2 \text{ is approximated to } s_2^+) \\
&= \{s_0 :: s_2^* :: \{! \hookrightarrow S \hookrightarrow s_2\}\} \quad (\text{reduce } S \rightarrow a) \\
&= \{s_0 :: s_2^+ :: \{! \hookrightarrow s_3\}\} \\
&= \{\{! \hookrightarrow S \hookrightarrow s_0\}, s_0 :: s_2^* :: \{! \hookrightarrow S \hookrightarrow s_2\}\} \quad (\text{reduce } S \rightarrow \mathbf{a}S) \\
&= \{s_4\} \quad (\text{second set element adds nothing to the fixed point})
\end{aligned}$$

This proves that all possible string values of \mathbf{x} at the end are well-structured S -phrases.

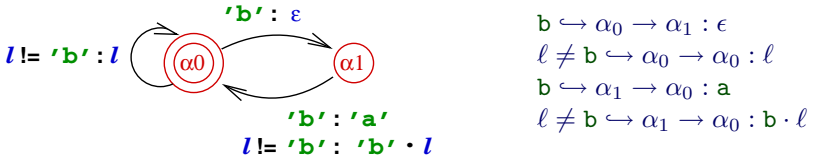
6 Abstract Parsing with String-Replacement Operations

Because of the state explosion that arises with LR(k) grammars, our implementation uses LALR(k) grammars instead. The reason we introduced the LR(1) example was to generalize the parse state to hold multiple input symbols, which we now use to process string-update operations.

Scripting languages support string updates of this form,

`y = replace 'bb' by 'a' in x`

where the pattern (here, `bb`) can be a regular expression. The update operation defines an automaton (more precisely, a *transducer*):



Both graphical and linear codings are displayed here. We use $: e$ to mean “emit e as output.” When a `replace` operation appears in a program that is analyzed, the transducer, α , defined by `replace` is composed with the parser automaton — a state configuration now holds *two* states:

$$[l_{new} \hookrightarrow \alpha_m, l_j \hookrightarrow \dots \hookrightarrow l_0 \hookrightarrow s]$$

Here, α_m is the current state of the transducer and s is the current state of the parser. A new input, l_{new} , submits first to α_m , which transits and possibly emits input for s :

$$[\alpha_n, l_{j+1} \hookrightarrow l_j \hookrightarrow \dots \hookrightarrow l_0 \hookrightarrow s]$$

In this way, strings are updated by `replace` before they are parsed. The assignment,

`x = replace S1 by S2 in E`

generates the flow equation

$$X = insert_\alpha \cdot E \cdot erase_\alpha$$

where α_0 names the start state of transducer α generated from patterns $S1$ and $S2$ and

$$\begin{aligned} insert_\alpha[\dots s] &\Rightarrow [\alpha_0, \dots s] \\ erase_\alpha[\alpha_i, \dots s] &\Rightarrow [\dots s] \end{aligned}$$

Here is a small example, worked with the above transducer and the parser in Figure 2:

<code>y = 'bb'</code>	$Y = b \cdot b \cdot]$
<code>x = '['.(replace 'bb' by 'a' in y)</code>	$X = [\cdot (insert_\alpha \cdot Y \cdot erase_\alpha)$

The abstract parse of $X[s_0]$ proceeds like this:

$$\begin{aligned}
X[s_0] &= [\llbracket \hookrightarrow s_0 \rrbracket \oplus (\text{insert}_\alpha \cdot Y \cdot \text{erase}_\alpha) = s_0 :: (\text{insert}_\alpha \cdot Y \cdot \text{erase}_\alpha)[s_1] & (i) \\
&= s_0 :: (Y \cdot \text{erase}_\alpha)[\alpha_0, s_1] = s_0 :: (Y[\alpha_0, s_1] \oplus \text{erase}_\alpha) \\
Y[\alpha_0, s_1] &= (\mathbf{b} \cdot \mathbf{b} \cdot \llbracket \rrbracket)[\alpha_0, s_1] = [\mathbf{b} \hookrightarrow \alpha_0, s_1] \oplus (\mathbf{b} \cdot \llbracket \rrbracket) & (ii) \\
&= [\mathbf{b} \hookrightarrow \alpha_1, s_1] \oplus \llbracket \rrbracket = [\alpha_0, \mathbf{a} \hookrightarrow s_1] \oplus \llbracket \rrbracket & (iii) \\
&= s_1 :: ([\alpha_0, s_2] \oplus \llbracket \rrbracket) & (iv) \\
&= [\alpha_0, S \hookrightarrow s_1] \oplus \llbracket \rrbracket = s_1 :: [\alpha_0, s_3] \oplus \llbracket \rrbracket = s_1 :: \llbracket \hookrightarrow \alpha_0, s_3 \rrbracket \\
&= s_1 :: [\alpha_0, \llbracket \hookrightarrow s_3 \rrbracket] = s_1 :: s_3 :: [\alpha_0, s_4]
\end{aligned}$$

So,

$$\begin{aligned}
X[s_0] &= s_0 :: (Y[\alpha_0, s_1] \oplus \text{erase}_\alpha) = s_0 :: (s_1 :: s_3 :: [\alpha_0, s_4] \oplus \text{erase}_\alpha) \\
&= s_0 :: (s_1 :: s_3 :: \text{erase}_\alpha[\alpha_0, s_4]) & (v) \\
&= s_0 :: (s_1 :: s_3 :: [s_4]) = [S \hookrightarrow s_0] = [s_5]
\end{aligned}$$

At point (i), input symbol $\llbracket \rrbracket$ is supplied directly to the parser. The transducer’s start state is then added to the state configuration, and the string generated by Y is supplied to the transducer *before* Y ’s string is parsed — see (ii). At point (iii), the sequence \mathbf{bb} causes the transducer state to emit \mathbf{a} , which is supplied to the parser state. Once the parser reduces \mathbf{a} to nonterminal S , the transducer state is carried along in the state configuration; see (iv). At (v), the string transducer has finished its effects and is erased.

The composition of transducer with parser in our demand-driven, backwards, precondition-style analysis means there is no backtracking and reparsing because of string updates — there is only the one parse of the appropriately altered string.

There is a last, important, technical point: a string-replacement transducer must finish its work in a final state, e.g., for

`y = replace 'bb' by 'a' in 'bbb'`

where transducer α has α_0 as its final state, the processing of 'bbb' causes α to finish in state α_1 , implicitly holding 'b' in its state. The 'b' must be “flushed”, so we add this last transition to α :

$$\text{eos} \hookrightarrow \alpha_1 \rightarrow \alpha_0 : \mathbf{b}$$

Where *eos* denotes “end of string.” This transition is enacted by the erase_α operation.

The embedding of the transducer state in the parse configuration does not affect the least-fixed point machinery for computing the solutions to the residual equations. (But a state explosion can result.) In addition, the residual-equation-least-fixed-point-calculation allows string replacements within loop bodies, avoiding difficulties encountered in related techniques [4, 5, 14].

7 Other Applications of String Transducers

Transducers have other applications in abstract parsing; here are two.

Composing a Scanner with the Parser. The basic abstract-parsing algorithm does “scannerless parsing” — the characters of a string are input one at a

time to the parser state, which must parse characters into words and words into phrases. We have found the technique acceptable in practice for HTML grammars, but for theoretical or practical reasons, one might wish to scan characters into tokens before parsing them.

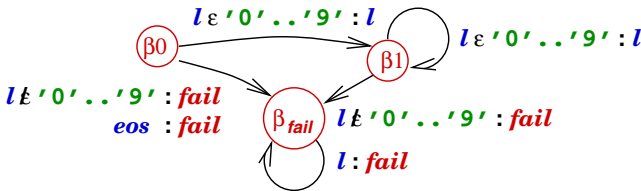
A scanner defined as a transducer, σ , can be added to the state configuration so that abstract parsing is undertaken with configurations of the form, $[\sigma_i, \ell_j \leftrightarrow \dots \leftrightarrow \ell_0 \leftrightarrow s_j]$, where σ_i is the state of the scanner, ℓ_i are the generated tokens, and s_j is the state of the parser. There is no resulting state explosion, since a scannerless parser must hold the same scanner-state information within its parse state, anyway, and there is the advantage that scanning and parsing can be defined separately.

String Filtering through Conditional Commands. A technique needed for taint analysis [19–21] is filter functions that model the tests of conditional commands. For example, a script might contain a conditional command that filters untrusted user input:

```

read x
if isAllDigits(x) :
then ...assert here that x holds all digits...
    
```

The test expression, `isAllDigits(x)`, is defined as a transducer that reads string `x` and emits *failure* (\perp) if a character is a nondigit. A failure means that `x`'s value is filtered from entering the conditional's then-arm. The filter transducer for `isAllDigits(x)` appears:



The transducer emits *fail* when its input fails the boolean test. The complement automaton, $\neg\beta$, merely swaps the outputs, `l` and *fail*.

Our approach to analyzing conditional statements goes as follows:

For the conditional,	generate these flow equations:
<code>if B(x):</code>	$X_B = \text{insert}_\beta \cdot X \cdot \text{erase}_\beta$
<code>then ...x...</code>	$\dots X_B \dots$
<code>else ...x...</code>	$X_{\neg B} = \text{insert}_{\neg\beta} \cdot X \cdot \text{erase}_{\neg\beta}$
	$\dots X_{\neg B} \dots$

where β is the transducer that implements test B and $\neg\beta$ implements $\neg B$.

The *fail* character is special — when processed as an input, it causes the parse to denote \perp (empty set in the powerset lattice): $[\dots, \text{fail}, \dots] = \perp$. For example,

<code>x = 'a'</code>	$X_0 = a$
<code>if isAllDigits(x):</code>	$X_1 = \text{insert}_\beta \cdot X_0 \cdot \text{erase}_\beta$
<code> print x</code>	$X_2 = X_1 \cdot !$

and

$$\begin{aligned} X2[s_0] &= X1[s_0] \\ X1[s_0] &= X0[\beta_0, s_0] \oplus \text{erase}_\beta \\ X0[\beta_0, s_0] &= \mathbf{a}[\beta_0, s_0] = [\mathbf{a} \hookrightarrow \beta_0, s_0] = [\beta_0, \text{fail} \hookrightarrow s_0] = \perp \end{aligned}$$

Hence,

$$X1[s_0] = \text{erase}_\beta \perp = \perp = X2[s_0]$$

The analysis correctly predicts that nothing prints within the body of the conditional.

8 Modelling Global Variables and User Input by Nonterminals

An abstract parser can process a grammar's nonterminal symbols as input just like terminal symbols: the symbol is supplied to the parse state, which shifts it. Say that a module uses a string-valued global variable that is initialized outside of the module. If we can assume the global variable's value has the structure named by a nonterminal, then the global variable can be used in an abstract parse. For example, assume global variable \mathbf{g} holds an S -structured string:

$$\begin{array}{ll} \mathbf{x} = \text{'['.g. ']} & G = S \\ \text{print } \mathbf{x} & X = [\cdot G \cdot] \end{array}$$

We readily compute the abstract parse for $X[s_0]$, using Figure 2:

$$\begin{aligned} X[s_0] &= ([\cdot G \cdot \text{'}]') [s_0] = [[\hookrightarrow s_0] \oplus (G \cdot \text{'})'] = s_0 :: (G[s_1] \oplus []) = s_0 :: ([S \hookrightarrow s_1] \oplus []) \\ &= s_0 :: (s_1 :: [] [s_3]) = s_0 :: s_1 :: [] \hookrightarrow s_3 = s_0 :: s_1 :: s_3 :: [s_4] = \dots = [s_5] \end{aligned}$$

In a similar way, user input can be assumed to have structure named by a nonterminal, and abstract parsing can be undertaken:

$$\begin{array}{ll} \mathbf{g} = \text{read}_S() & G = S \\ \mathbf{x} = \text{'['.g. ']} & X = [\cdot G \cdot] \\ \text{print } \mathbf{x} & \end{array}$$

Of course, we must supply a script that parses the input at runtime, to ensure that the input assumption is not violated.

But there is a rub — the program might contain string-replacement operations, which cannot process nonterminals. We solve this problem by unfolding the nonterminal, supplying the generated strings to the string-replacement transducer (recall that $S \rightarrow \mathbf{a} \mid [S]$):

<pre> g = read_S() y = replace '[' by '[' in g print y </pre>	<pre> S = a □ [· S ·] G = S Y = insert_γ · G · erase_γ </pre>
---	---

where transducer γ is the obvious one-state transducer. The analysis proceeds like this:

$$\begin{aligned}
Y[s_0] &= G[\gamma_0, s_0] \oplus \text{erase}_\gamma \\
G[\gamma_0, s_0] &= S[\gamma_0, s_0] \\
S[\gamma_0, s_0] &= (\mathbf{a}[\gamma_0, s_0]) \cup ([\cdot S \cdot][\gamma_0, s_0])
\end{aligned}$$

The last residual equation, for $S[\gamma_0, s_0]$, shows how nonterminal S is unfolded and its symbols fed to γ . There is a tedious but finitely computable solution:

$$\begin{aligned}
S[\gamma_0, s_0] &= (\mathbf{a}[\gamma_0, s_0] \cup ([\cdot S \cdot][\gamma_0, s_0]) \\
&= \{\gamma_0, s_5\} \cup \{s_0 :: s_1 :: S[\gamma_0, s_1] \oplus \}
\end{aligned}$$

The partial evaluation of $S[\gamma_0, s_1]$ unfolds almost identically, producing

$$\begin{aligned}
S[\gamma_0, s_1] &= (\mathbf{a}[\gamma_0, s_1] \cup ([\cdot S \cdot][\gamma_0, s_1]) \\
&= \{s_1 :: [\gamma_0, s_3]\} \cup ([\cdot S \cdot][\gamma_0, s_1]) \\
&= \{s_1 :: [\gamma_0, s_3]\} \cup \{s_1 :: s_1 :: [\gamma_0, s_1] \oplus (S \cdot)\} \\
&= \{s_1 :: [\gamma_0, s_3]\} \cup \{s_1^+ :: (S[\gamma_0, s_1] \cdot)\}
\end{aligned}$$

The least fixed-point solution of $S[\gamma_0, s_1]$ is $\{s_1^+ :: [\gamma_0, s_3]\}$, which gives $Y[s_0] = \{[s_5, s_1^+ :: [s_3]]\}$.

With the technique just illustrated, we can show the correctness of input-validation codings. For example, a script that goes

```

x = readS()
if isAllDigits(x):
then...

```

can be analyzed with respect to the automaton defined by `isAllDigits` and this reference grammar:

$$\begin{aligned}
S &::= C \mid CS & D &::= 0 \dots 9 \\
C &::= D \mid N & N &::= \text{all characters not in } D
\end{aligned}$$

9 Abstract Parsing with Semantic Processing

Since we can predict the syntax of dynamically generated strings, we should be able to predict the semantics as well by adapting attribute-grammar techniques.

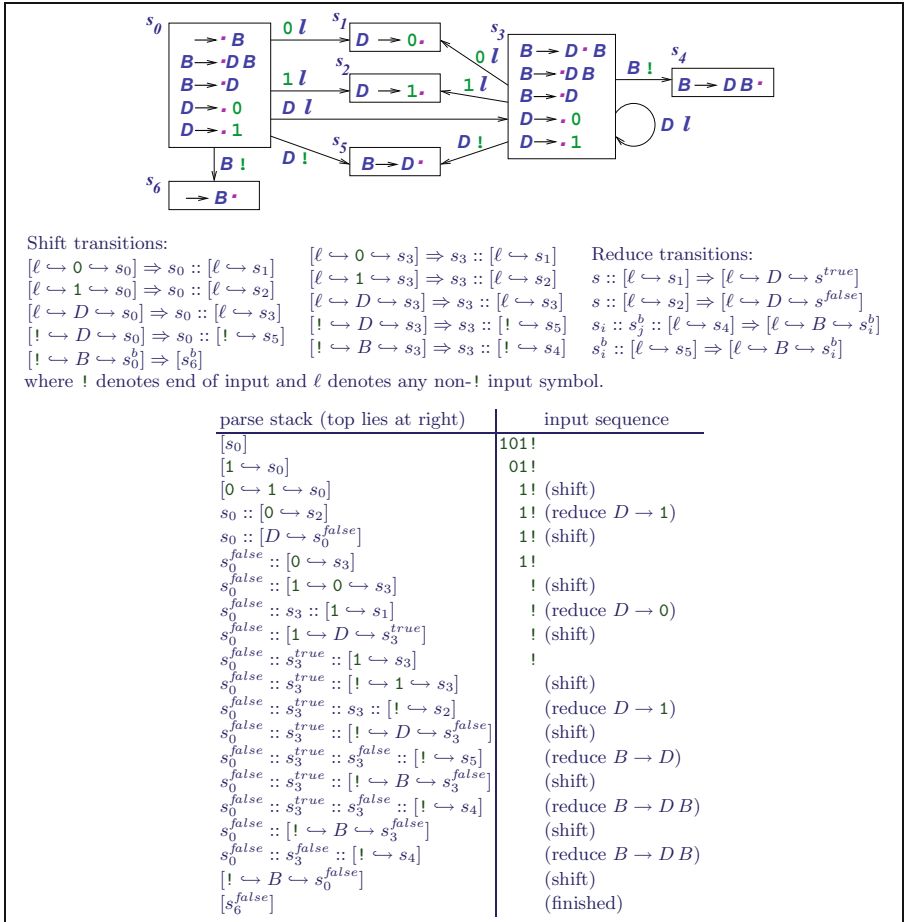


Fig. 5. Syntax-directed semantic processing for LR(1) grammar, $B \rightarrow DB \mid D$, $D \rightarrow 0 \mid 1$

Here is a simple but useful example. Binary numerals are generated by this LR(1) grammar,

$$\begin{aligned}
 B &\rightarrow DB \mid D \\
 D &\rightarrow 0 \mid 1
 \end{aligned}$$

where B stands for the set of binary numerals and D stands for the set of binary digits. The semantics of binary numerals can be specified with *attributes* associated with the grammar symbols and *semantic rules* associated with the productions. Suppose we want to know whether or not a binary numeral is even-valued. The semantic rules associated with the productions below specify how to calculate the answer:

production	semantic rule
$\rightarrow B!$	$\text{answer} = B.\text{even}$
$B \rightarrow D B_1$	$B.\text{even} = B_1.\text{even}$
$B \rightarrow D$	$B.\text{even} = D.\text{even}$
$D \rightarrow 0$	$D.\text{even} = \text{true}$
$D \rightarrow 1$	$D.\text{even} = \text{false}$

Here, each nonterminal, B and D , has a synthesized attribute, *even*, which has value *true* if the binary numeral generated by the nonterminal is an even number, *false* otherwise.

Since the grammar is LR(1) and only associated with synthesized attributes, the semantic rules can be computed during LR-parsing, as seen in Figure 5. When a reduce transition occurs, its corresponding semantic rule is computed. The computed result is annotated to its corresponding state, shown as a superscript in our notation. In the example in Figure 5, the computed attribute values are annotated only to the states, s_0 and s_3 . For the example binary numeral, 101, the computed result is *false*, as expected. For this program,

<code>x = '0'</code>	$X0 = 0$
<code>while ...</code>	$X1 = X0 \sqcup X2$
<code>x = read_D() · x</code>	$X2 = D · X1$
<code>print x · !</code>	$X3 = X1 · !$

its abstract parsing with synthesized-attribute computing proceeds as follows:

$$\begin{aligned}
 X3[s_0] &= (X1 \cdot !)[s_0] = X1[s_0] \oplus ! \\
 X1[s_0] &= X0[s_0] \cup X2[s_0] \\
 X0[s_0] &= 0[s_0] = \{[0 \hookrightarrow s_0]\} \\
 X2[s_0] &= (D \cdot X1)[s_0] = [D \hookrightarrow s_0] \oplus X1 = X1[D \hookrightarrow s_0] \\
 &= X0[D \hookrightarrow s_0] \cup X2[D \hookrightarrow s_0] \\
 X0[D \hookrightarrow s_0] &= \{[0 \hookrightarrow D \hookrightarrow s_0]\} \Rightarrow \{s_0 :: [0 \hookrightarrow s_3]\} \\
 X2[D \hookrightarrow s_0] &= (D \cdot X1)[D \hookrightarrow s_0] = [D \hookrightarrow D \hookrightarrow s_0] \oplus X1 \\
 &\Rightarrow \{s_0 :: [D \hookrightarrow s_3]\} \oplus X1 = \{s_0 :: X1[D \hookrightarrow s_3]\} \\
 X1[D \hookrightarrow s_3] &= X0[D \hookrightarrow s_3] \cup X2[D \hookrightarrow s_3] \\
 X0[D \hookrightarrow s_3] &= \{[0 \hookrightarrow D \hookrightarrow s_3]\} \Rightarrow \{s_3 :: [0 \hookrightarrow s_3]\} \\
 X2[D \hookrightarrow s_3] &= (D \cdot X1)[D \hookrightarrow s_3] = [D \hookrightarrow D \hookrightarrow s_3] \oplus X1 \\
 &\Rightarrow \{s_3 :: [D \hookrightarrow s_3]\} \oplus X1 = \{s_3 :: X1[D \hookrightarrow s_3]\}
 \end{aligned}$$

Now we have a recursive equation to solve:

$$\begin{aligned}
 X1[D \hookrightarrow s_3] &= \{s_3 :: [0 \hookrightarrow s_3]\} \cup \{s_3 :: X1[D \hookrightarrow s_3]\} \\
 &= \{s_3^+ :: [0 \hookrightarrow s_3]\}
 \end{aligned}$$

Using this result, we obtain:

$$\begin{aligned}
X2[D \hookrightarrow s_0] &= \{s_0 :: s_3^+ :: [0 \hookrightarrow s_3]\} \\
X2[s_0] &= \{s_0 :: [0 \hookrightarrow s_3], s_0 :: s_3^+ :: [0 \hookrightarrow s_3]\} \\
X1[s_0] &= \{[0 \hookrightarrow s_0], s_0 :: [0 \hookrightarrow s_3], s_0 :: s_3^+ :: [0 \hookrightarrow s_3]\} \\
X3[s_0] &= \{[! \hookrightarrow 0 \hookrightarrow s_0], s_0 :: [! \hookrightarrow 0 \hookrightarrow s_3], s_0 :: s_3^+ :: [! \hookrightarrow 0 \hookrightarrow s_3]\} \\
&= \{[s_6^{true}]\}
\end{aligned}$$

$$\text{since } \{[! \hookrightarrow 0 \hookrightarrow s_0]\} \Rightarrow \{s_0 :: [! \hookrightarrow s_1]\} \Rightarrow \{[! \hookrightarrow D \hookrightarrow s_0^{true}]\}$$

$$\Rightarrow \{s_0^{true} :: [! \hookrightarrow s_5]\} \Rightarrow \{[! \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\}$$

$$\text{and } \{s_0 :: [! \hookrightarrow 0 \hookrightarrow s_3]\} \Rightarrow \{s_0 :: s_3 :: [! \hookrightarrow s_1]\} \Rightarrow \{s_0 :: [! \hookrightarrow D \hookrightarrow s_3^{true}]\}$$

$$\Rightarrow \{s_0 :: s_3^{true} :: [! \hookrightarrow s_5]\} \Rightarrow \{s_0 :: [! \hookrightarrow B \hookrightarrow s_3^{true}]\}$$

$$\Rightarrow \{s_0 :: s_3^{true} :: [! \hookrightarrow s_4]\} \Rightarrow \{[! \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\}$$

$$\text{and } \{s_0 :: s_3^+ :: [! \hookrightarrow 0 \hookrightarrow s_3]\} \Rightarrow \{s_0 :: s_3^+ :: s_3 :: [! \hookrightarrow s_1]\}$$

$$\Rightarrow \{s_0 :: s_3^+ :: [! \hookrightarrow D \hookrightarrow s_3^{true}]\} \Rightarrow \{s_0 :: s_3^+ :: s_3^{true} :: [! \hookrightarrow s_5]\}$$

$$\Rightarrow \{s_0 :: s_3^+ :: [! \hookrightarrow B \hookrightarrow s_3^{true}]\} \Rightarrow \{s_0 :: s_3^+ :: s_3^{true} :: [! \hookrightarrow s_4]\}$$

$$\Rightarrow \{s_0 :: [! \hookrightarrow B \hookrightarrow s_3^{true}], s_0 :: s_3^+ :: [! \hookrightarrow B \hookrightarrow s_3^{true}]\}$$

(second set element adds nothing to fixed point)

$$\Rightarrow \{s_0 :: s_3^{true} :: [! \hookrightarrow s_4]\} \Rightarrow \{[! \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\}$$

This proves that all possible string values of \mathbf{x} at the end are well-structured B -phrases and even-valued. The approach is well suited to “type checking” XML-like documents; this application is currently under investigation.

10 Conclusion

The worklist algorithm for abstract parsing discussed in this paper has been implemented for PHP applications that dynamically generate HTML documents. A scannerless LALR(1) parsing table for an HTML grammar written up to character level is automatically generated by a parser generator, and a set of flow equations are generated from the PHP application to be analyzed. Our abstract parser then takes the flow equations, the parsing table, and a hot spot and parses the set of all documents dynamically generated at the given hot spot. In addition, our abstract parser builds a set of abstract syntax trees of the documents for the use of further analyses. The current implementation has been applied to a suite of PHP applications publicly available and has successfully identified multiple parse errors in a reasonable execution time with a few predictable false positives [7].

The extensions proposed in this paper, such as dealing with destructive string operators, composing scanner with the parser, modular abstract parsing with the existence of unknown string variables, string filtering through conditionals, and semantic processing such as type checking and taint analysis, are currently being implemented or are planned for implementation in the near future. The extensions are expected to remove false positives observed from our initial implementation and to make abstract parsing more practical and useful.

Acknowledgements. We thank Carolyn Talcott for her decades of leadership in programming-languages research and dedicate this paper to her on the occasion of her 60th birthday.

Conversations with Anders Møller, Se-won Kim, and Kwangkuen Yi and his group at Seoul National University have been helpful. We thank GTOne's Soo-Yong Lee for his inspiration and support.

References

1. Agrawal, G.: Simultaneous demand-driven data-flow and call graph analysis. In: Proc. Int'l. Conf. Software Maintenance, Oxford (1999)
2. Brabrand, C., Møller, A., Schwartzbach, M.I.: The <bigwig> project. ACM Trans. Internet Technology 2 (2002)
3. Choi, T.-H., Lee, O., Kim, H., Doh, K.-G.: A practical string analyzer by the widening approach. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 374–388. Springer, Heidelberg (2006)
4. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Static analysis for dynamic XML. In: Proc. PLAN-X 2002 (2002)
5. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Extending Java for high-level web service construction. ACM TOPLAS 25 (2003)
6. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
7. Doh, K.-G., Kim, H., Schmidt, D.A.: Abstract parsing: static analysis of dynamically generated string output using lr-parsing technology. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 256–272. Springer, Heidelberg (2009)
8. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. ACM TOPLAS 19, 992–1030 (1997)
9. Goldfarb, C.F.: The SGML Handbook. Oxford Univ. Press (1991)
10. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engg. (1995)
11. Jones, N., Nielson, F.: Abstract interpretation: a semantics-based tool for program analysis. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, vol. 4, pp. 527–636. Oxford Univ. Press (1995)
12. Jones, N.D., Mycroft, A.: Data flow analysis of applicative programs using minimal function graphs. In: Proc. 13th Symp. POPL, pp. 296–306. ACM Press (1986)
13. Kirkegaard, C., Møller, A.: Static analysis for Java Servlets and JSP. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 336–352. Springer, Heidelberg (2006)
14. Minamide, Y.: Static approximation of dynamically generated web pages. In: Proc. 14th ACM Int'l Conf. on the World Wide Web, pp. 432–441 (2005)
15. Minamide, Y., Tozawa, A.: XML validation for context-free grammars. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 357–373. Springer, Heidelberg (2006)
16. Møller, A., Schwarz, M.: HTML validation of context-free languages. Technical report, Computer Science Dept., Aarhus University (2010)
17. Nishiyama, T., Minamide, Y.: A translation from the HTML DTD into a regular hedge grammar. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 122–131. Springer, Heidelberg (2008)
18. Thiemann, P.: Grammar-based analysis of string expressions. In: Proc. ACM Workshop Types in Languages Design and Implementation, pp. 59–70 (2005)
19. Wassermann, G., Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. ACM Trans. Software Engineering and Methodology 16(4), 14:1–14:27 (2007)

20. Wassermann, G., Su, Z.: The essence of command injection attacks in web applications. In: Proc. 33d ACM Symp. POPL, pp. 372–382 (2006)
21. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proc. ACM PLDI, pp. 32–41 (2007)

Appendix: Concrete, Collecting, and Abstract Semantics

A source program computes an output store that maps variables to strings. The *concrete collecting semantics* [11] defines a set of stores for each program point (command line); the collecting semantics is then abstracted in the usual fashion so that it computes, for each program point, a single store that maps each variable to a set of strings. For the example in Figure 1, we have

$$\begin{aligned}
 p_0 &== \mathbf{x} \mapsto \{\text{'a'}\} \\
 p_r &== \mathbf{x} \mapsto \{\text{'a'}\}, \mathbf{r} \mapsto \{\text{'}'\} \\
 p_1 &== \mathbf{x} \mapsto \{\text{'[}'^i\text{'a'}\text{'}'^i \mid i \geq 0\}, \mathbf{r} \mapsto \{\text{'}'\} \\
 p_2 &== p_1 == p_3
 \end{aligned}$$

The collecting semantics is overapproximated by the *data-flow semantics*, which uses flow equations to define the set of strings denoted by each variable at each program point. In Figure 1, the listed data-flow equations are a shorthand for this fuller form:

$$\begin{aligned}
 X_0 &= \mathbf{a} & X_r &= X_0 \\
 R_r &= \text{]} & R_1 &= R_r \\
 X_1 &= X_r \sqcup X_2 & R_1 &= R_r \\
 X_2 &= [\cdot X_1 \cdot] & R_2 &= R_1 \\
 X_3 &= X_1 & R_3 &= R_1
 \end{aligned}$$

The least-fixed-point solution is computed in the domain of sets of strings. Since the example ignores the loop test, the data-flow semantics computes the same sets as the collecting semantics.

Let Σ name the states in the LR(k)-parser’s controller. A parse stack has form, $s_1 :: s_2 :: \dots :: s_j :: [c]$, $j \geq 0$, where each $s_i \in \Sigma$, and the top, current parse state, $[c] \in \text{Configuration}$, has the form, $[\ell_j \hookrightarrow \dots \hookrightarrow \ell_0 \hookrightarrow s]$, $0 < j < k$, where the ℓ_i s are input symbols and $s \in \Sigma$.

Function $\gamma : \text{ParseStack} \rightarrow (\Sigma \times \mathcal{P}(\text{String}))$ concretizes a parse stack into the start state and the string(s) that generate the stack:

$$\gamma(st) = (s_0, T) \text{ such that } LR\text{parse}(t, s_0) = st \text{ and } t \in T$$

The function, $\gamma^* : \mathcal{P}(\text{ParseStack}) \rightarrow \mathcal{P}(\Sigma \times \text{String})$, is the induced lift.

The *abstract-parse interpretation*, \mathcal{X} , computes the set of parse stacks denoted by each variable at each program point: For flow equation, $X_i = E_i$, the function, $\mathcal{X}_i : \text{Configuration} \rightarrow \mathcal{P}(\text{ParseStack})$, is defined $\mathcal{X}_i[s] = \llbracket E_i \rrbracket [c]$, where

$\llbracket \mathbf{a} \rrbracket [c] = \{\text{rewrite}[\mathbf{a} \leftrightarrow c]\}$, where \mathbf{a} is a terminal symbol

$\llbracket E_1 \sqcup E_2 \rrbracket [c] = \llbracket E_1 \rrbracket [c] \cup \llbracket E_2 \rrbracket [c]$

$\llbracket X_j \rrbracket [c] = \llbracket E_j \rrbracket [c]$, where $X_j = E_j$ is the flow equation for X_j

$\llbracket E_1 \cdot E_2 \rrbracket [c] = \{\text{rewrite}(p') \mid p' \in (\llbracket E_1 \rrbracket [c] \oplus \llbracket E_2 \rrbracket [c])\}$,

where $S \oplus g = \{\text{tail}(p) :: g(\text{head}(p)) \mid p \in S\}$, $S \in \mathcal{P}(\text{ParseStack})$

where $\text{rewrite}(st)$ repeatedly applies the shift/reduce rules, \Rightarrow , to st until a normal form is achieved.

Using γ , we can prove the abstract-parse interpretation sound with respect to the concrete collecting semantics.

The abstract-parse interpretation is made finitely convergent by abstracting the domain, $\mathcal{P}(\text{ParseStack})$, into the domain of sets of subgraphs of the LR(k)-parser automaton: Represent a stack, $st = s_1 :: s_2 :: \dots :: s_j :: [c]$, as the linked path, $\text{pathst} = \leftarrow s_1 \leftarrow s_2 \leftarrow \dots \leftarrow s_j \leftarrow [c] \leftarrow$. Define $\text{fold} : \text{ParseStack} \rightarrow \text{ParserSubgraph}$ as $\text{fold}(\text{pathst}) = \leftarrow^1 G \leftarrow^j [c] \leftarrow$, where G is the smallest subgraph in the parser automaton that covers the path from s_1 to s_j ; \leftarrow^1 is an out-arc from node s_1 ; and \leftarrow^j is an in-arc into node s_j . The answer graph is computed by folding repeating states in the argument path and preserving all arcs.

fold 's definition is easily generalized from folding paths to folding graphs — merge repeating nodes and retain the arcs.

The finitely convergent abstract-parse interpretation is defined in terms of fold ; we modify the semantics of this one clause of the abstract-parse interpretation:

$\llbracket X_j \rrbracket [c] = \text{fold}^*(\llbracket E_j \rrbracket [c])$, where $X_j = E_j$ is the flow equation for X_j
 and $\text{fold}^*(T) = \{\text{fold}(t) \mid t \in T\}$

A set of subgraphs might be further abstracted into a single graph by unioning the graphs in the set into one graph — merge the graphs' like-named nodes and preserve the edges.

Fractionated Software for Networked Cyber-Physical Systems: Research Directions and Long-Term Vision

Mark-Oliver Stehr, Carolyn Talcott, John Rushby, Pat Lincoln,
Minyoung Kim, Steven Cheung, and Andy Poggio

SRI International

{stehr,clt,rushby,lincoln,mkim,cheung,poggio}@cs1.sri.com

Abstract. An emerging generation of mission-critical systems employs distributed, dynamically reconfigurable open architectures. These systems may include a variety of devices that sense and affect their environment and the configuration of the system itself. We call such systems *Networked Cyber-Physical Systems* (NCPS). NCPS can provide complex, situation-aware, and often critical services in applications such as distributed sensing and surveillance, crisis response, self-assembling structures or systems, networked satellite and unmanned vehicle missions, or distributed critical infrastructure monitoring and control.

In this paper we lay out research directions centered around a new paradigm for the design of NCPS based on a notion of software fractionation that we are currently exploring which can serve as the basis for a new generation of runtime assurance techniques. The idea of software fractionation is inspired by and complementary to hardware fractionation — the basis for the fractionated satellites of DARPA’s F6 program. *Fractionated software* has the potential of leading to software that is more robust, leveraging both diversity and redundancy. It raises the level of abstraction at which assurance techniques are applied. We specifically propose research in just-in-time verification and validation techniques, which are agile — adapting to changing situations and requirements, and efficient — focusing on properties of immediate concern in the context of locally reachable states, thus largely avoiding the state space explosion problem. We propose an underlying reflective architecture that maintains models of itself, the environment, and the mission that is key for adaptation, verification, and validation.

1 Introduction and Motivation

The increasing availability of systems and devices that can sense and affect their environment in different ways and with different levels of sophistication is the starting point for development of a new generation of *Networked Cyber-Physical Systems* (NCPS). Such systems provide complex, situation-aware, and often safety- or mission-critical services. Examples include traffic control (air and ground), medical systems, smart power grids, flexible manufacturing systems, automated laboratories, microclimate control in buildings, structural monitoring

and control, self-assembling structures or systems, unmanned vehicles (including autonomous robots and UAVs), networked satellite missions (including future fractionated designs), deep space exploration vehicles, instrumented spaces for surveillance and emergency response, and ad hoc combat teams (on the ground and airborne). Especially interesting and challenging examples are complex heterogeneous networked systems with humans and (autonomous) agents in the loop, such as vehicular networks, mobile social networks, or the global network of financial markets.

A number of special-purpose solutions exist for different aspects of NCPS. However, general principles and tools for building robust, effective NCPS applications/services using individual cyber-physical devices as building blocks are missing. Furthermore, the verification and validation of NCPS is notoriously difficult and conventional techniques are too expensive, which is a serious problem because the capabilities and the flexibility of NCPS are urgently needed for today's complex mission-critical applications. Factoring out the minimal functionality common to NCPS is a first step toward making verification feasible, because the cost of verification can be amortized over many instantiations of the common framework. This is far from enough, however, because mission-specific properties and performance metrics will require verification, too, and the mission-specific software will typically be much more complex than the minimal framework. Furthermore, conventional verification cannot enable rapid deployment at acceptable cost.

We propose to tackle this problem by considering the notion of software fractionation, which is directly inspired by hardware fractionation, specifically the idea of fractionated satellites [9] that is the basis for DARPA's F6¹ program. We believe that software fractionation has the potential of leading to software that is more robust and can be designed to be verified at reasonable cost by raising the level of abstraction at which verification is applied. We will argue, however, that verification in the conventional sense is not a sensible solution for the flexible, dynamically reconfigurable, mission-critical NCPS, which will lead us to propose new research opportunities in a hardly explored direction of runtime assurance.

Challenges and Opportunities in Networked Cyber-Physical Systems. Many challenges exist in the context of NCPS. They have a wide range of assurance requirements, operate in a distributed environment, and unlike pure sensor networks they can perform physical actions and are usually characterized by (distributed) control loops through which the environment provides essential feedback. There is a large overlap between NCPS and wireless sensor networks augmented with actuators, also known as sensor/actor networks [4,19], but it should be noted that, in NCPS, node and communication capabilities can vary significantly. For instance, in addition to resource-constrained embedded sensor/actuator nodes, devices carried by humans (e.g., PDAs), energy-rich nodes attached to vehicles (e.g., laptops), resource-constrained UAVs, solar-powered satellites of different

¹ Future, Fast, Flexible, Formation-Flying, Fractionated Spacecraft united by Information eXchange.

sizes (including pico-satellites such as Cubesats), as well as nodes with continuous Internet connectivity (e.g., ground stations and computationally powerful grid nodes) can all be part of the same NCPS.

In addition to the real-time, resource-limited, reactive aspects of traditional embedded systems, an NCPS must embody a situation awareness that reflects the overall distributed system and its environment. Local situation awareness of a network node is not sufficient. Each node must maintain a model of its local, directly observable situation together with models about the rest of the network. Models must also account for uncertainty, partial knowledge, and bad or stale information. Furthermore, different nodes may have different degrees of awareness according to their capabilities. Asynchronous actions must achieve a desired overall coherent effect. An NCPS needs to be open in the sense that nodes may come and go. In fact, a system may assemble ‘on the fly’ for a given purpose. Mission-critical systems may be scaled up or down depending on mission requirements.

An advantage of multiple distributed nodes is that resources can be pooled and limitations can be partially overcome by cooperation. To realize the potential benefits of pooling resources (energy, CPU cycles, memory, bandwidth, sensors/actuators) it is necessary for the different processes/layers on each node to adapt resource usage (setting parameters, choosing policies) to achieve system-wide objectives, not just local goals.

From Networked to Fractionated Cyber-Physical Systems. The networked structure of NCPS normally arises as a by-product of their required capabilities (e.g., the need to perform distributed sensing) and is usually seen as an inconvenience for engineering, a challenge for verification, and even a hazard to the operation of the system. In this paper, we propose to view distribution as an opportunity (and in some sense as a necessity) rather than an obstacle for building high-assurance systems. In fact, we propose what seems to be counterintuitive — namely, to even further increase the degree of distribution and nondeterminism by moving toward systems that are fractionated by design not only in terms of their hardware but also at the software level. Hardware and software *fractional elements* or *fragments*, as we call them, are very different from traditional components, in that they do not have to correctly perform a well-defined function. Instead, reliable functionality is achieved by a group of such fragments interacting in an opportunistic fashion.

The idea of achieving robustness through diversity and redundancy seems to be a fundamental underlying principle of biological systems. The natural exposure to faults has not only enabled evolution as a mechanism for progress in many dimensions, but has been turned into an advantage by favoring more robust designs. For instance, the human immune system is an example of an effective NCPS. Characteristics of the immune system include robustness, generic and adaptive responses to events, distributed knowledge, diversity, authentication and integrity checking mechanisms, adaptive control, autonomous operation, and heterogeneous actuators. It is a system with continual deployment of novel entities, intermittent connectivity, exchange of information among

heterogeneous entities, such as the nervous and metabolic system components, and uninterrupted operation. There is dynamic optimization, for example, in the crucial balance between quick generic action and deliberate, aggressive specialized actions. The global behavior of the system emerges from predominantly local actions and asynchronous propagation of information.

Diversity and redundancy have also been successfully employed for risk reduction in finance, although the recent financial crisis shows that alone they are not sufficient to prevent systemic failures. Hardware and software fault tolerance is another area where these concepts have been exploited, but their use is mostly coarse grained, with limited degrees of diversity and redundancy, and applied to specific components or subsystems rather than used as an overall design principle. In fault-tolerant or disruption-tolerant networking, the loss of nodes (or connectivity) can be overcome, but a natural question is whether software can be designed so that this tolerance emerges as a special case of more fine-grained general design principles.

Why is this related to verification and validation? The simple answer is that an inherently fault-tolerant architecture raises the level of abstraction to a point where verification and validation becomes interesting and worthwhile. We propose to steer away from low-level code verification and to focus the verification effort mainly on system properties. In our view, code verification is too expensive for what it provides — namely, local correctness properties against detailed and possibly incorrect/incomplete specifications that are based on many assumptions about the environment and the underlying hardware and software. For instance, in challenging environments where failures in processors, memory, networking, sensors, firmware, and drivers are common, the benefit of maximum assurance for just one aspect — namely, the code — is economically questionable. To obtain a precise understanding of the benefits and trade-offs, an economic theory of high assurance design (and possibly beyond) would be needed, for instance along the lines of Rushby's suggested *science of certification* [69] taking into account possible trade-offs between confidence and degree of correctness [8]. A key idea elaborated in [70] is that at some level of abstraction formal methods are able to provide a notion of *possible perfection* enabling compositional arguments about system reliability.

Fractionated software represents a potential paradigm shift, but the high level of abstraction enabled by fractionated design is where the real challenges start. Conventional verification techniques will not be suitable for the mission-driven dynamically reconfigurable cyber-physical systems that we envision in the future. System requirements and configuration are usually not known at design time, which requires us to shift most of the verification activities to the time when sufficient information is available. Typically, this will be after the deployment, that is, at system runtime.

From Design-Time to Runtime Assurance. To our knowledge the provocative possibility of just-in-time certification of cyber-physical systems was first raised in [69]. In fact, just-in-time certification is one step beyond just-in-time verification in the sense that an explicit certificate is generated at runtime as evidence

for system correctness. In general, it may not be necessary to generate an explicit certificate, but the core idea of just-in-time certification — namely, the application of design-time formal methods at runtime — is an opportunity that we suggest exploring systematically. Hence, a few key arguments from the above-mentioned paper are worthwhile to summarize. Standards-based certification as it is mostly practiced today in the United States (using a standard such as DO-178B for airborne software) does not provide a clear link between the required artifact and the system requirements. The choice of methods has to rely on extensive expert knowledge and experience, which means that the application to novel circumstances is nearly impossible, making it a barrier to innovation. Usually, the conservative design practices that are required (e.g. limitations on scheduling and memory management) are at odds with innovative architectures such as those needed for today’s flexible mission-critical systems. Future systems exist in many configurations, are reconfigurable, and undergo evolution during their lifetime. The number of possible configurations can be enormous (e.g., 50000 lines of XML for an airplane). Since the final configuration is determined after the design and most configurations are never used, just-in-time certification would be a perfectly adequate solution.

We suggest going one step further by looking at systems, like fractionated spacecrafts, that are dynamically reconfigurable and extensible so that the consequent generalization of this idea is to view verification as an ongoing process during the entire lifetime of the system that can be carried out by the system itself. The need for rapid instantiation and deployment of a system for a new possibly unanticipated mission (e.g., within hours) dictates that the verification process must be automated and needs to be executable under critical time and resource constraints. Yet another argument for runtime methods is simply the expectation that future systems will be highly flexible and possibly universal to capture the diversity of possible missions, and the requirements can rarely be stated at design time. A related issue that is often neglected is the validation of specifications, to answer the question if the specification, which will be typically derived from the mission objective, is sensible and captures the intentions. In line with the previous arguments, the most essential validation tasks should also be performed just-in-time — namely, whenever the system interacts with the operator and is tasked with a new mission. The result of a failed validation might mean that the system must be scaled up (e.g., extended) or the objectives need to be scaled down. Clearly, modifications of a mission and changes of the system need to be revalidated, which is why just-in-time validation needs to be, like verification, an ongoing process, which in a similar way takes advantage of (partial) knowledge about the current system configuration.

Overview of this Paper. To build systems that satisfy requirements (verify) and perform their intended mission (validate) under a wide range of possible system configurations and with potentially degraded resources, we propose the reflective system architecture depicted in Fig. 1 and outline key research directions. The architecture has three main components: (1) A fractionated software kernel with a reflective simulation capability that is a crucial building block for

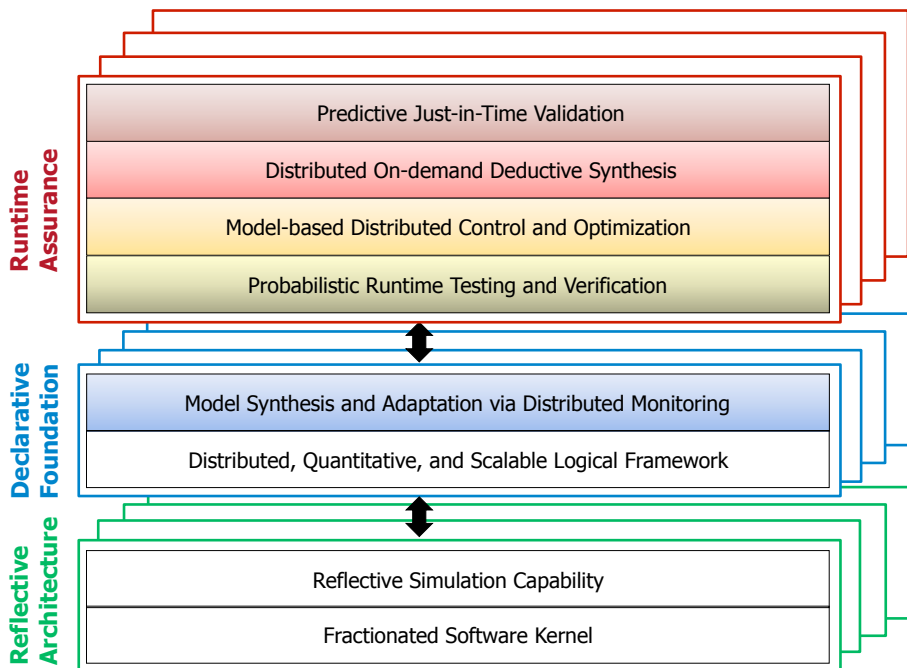


Fig. 1. Stylized System architecture

runtime assurance. (2) A declarative foundation for NCPS in the form of a distributed logical framework that is quantitative and scalable. The logic supports reasoning in the context of system goals and models maintained via distributed monitoring. (3) A new generation of runtime assurance techniques, including novel probabilistic runtime testing and verification methods such as predictive analysis, integration of symbolic and simulation-based techniques, adaptive runtime abstraction, resource- and situation-aware runtime assurance, and learning from the system dynamics. System adaptation through model-based distributed control and optimization is at the core of this set of techniques. We illustrate these ideas using examples centered around fractionated satellite networks, which originally served as an inspiration for the overall approach.

The stylized system architecture illustrates how the different techniques discussed in this paper can work together. For clarity we used a one-dimensional presentation, showing how different levels of runtime assurance can be built on top of each other and ultimately on top of our proposed reflective fractionated software architecture. This is by no means the only way to integrate the different techniques, and not all layers will be equally important or even needed for all NCPS. The architecture clearly distinguishes between the logical framework that provides a declarative view of the NCPS and the runtime assurance layers. Declarative and executable models of the physical world and of the system fragments are maintained and continuously adapted while the system evolves. The logical framework enables a rich set of possible behaviors of the NCPS, whereas

the control and optimization strategy restricts the evolution of the NCPS by exploiting models, invoking runtime assurance techniques, and taking into account the overall system goal, which is represented in the language of the logical framework. On-demand synthesis produces solutions (plans) for complex tasks that require transitioning the system through a series of intermediate goals. Just-in-time validation, finally, is concerned with the validation of mission goals in the context of other applicable policies.

Guided by this architecture, we will address each for the following research directions in a subsequent section, followed by an illustration of these ideas in the domain of fractionated satellites and by a discussion of some related work.

- Software Fractionation
 - provides high level of abstraction from low-level failures
 - leverages diversity, redundancy, distribution, and nondeterminism
 - covers wide spectrum of autonomy and cooperation
- Distributed Logical Foundation for NCPS
 - expresses degrees of satisfaction and uncertainty
 - supports distributed robust dynamic proofs
 - enables adaptive models with multilevel abstraction
- Runtime Assurance with Distributed Declarative Control
 - covers runtime validation, synthesis, verification, testing
 - integrates proof and optimization strategies
 - balances system and assurance goals through agile adaptation

2 A Reflective Architecture for Fractionated Software

To provide a suitable level of abstraction for the new set of runtime verification and related techniques that we propose to explore, we assume that software consists of fragments that are running on top of a fractionated software kernel, a minimalistic framework that enables the fragments to interact. Since a common aspect of all our proposed verification techniques is the capability of the system to analyze its own behavior, we factor out a reflective simulation capability as an intermediate layer directly on top of the kernel.

2.1 Fractionated Software Kernel

Like a biological cell, a software fragment does not have to make sense in isolation, but its interesting properties may emerge only at a higher level of abstraction where multiple fragments interact with each other. The emerging properties are the ones that we are interested in verifying. How they are established is less important as long as we can quantify their probabilities.

By enabling verifiable software, the objectives of the fractionated software kernel are similar to those of a separation kernel [67] and robust partitioning [68] in integrated modular avionics. The fractionated software kernel, however,

is conceptually distributed over many nodes and exploits distribution to achieve a new dimension of decoupling, diversity, and redundancy. Each node could be running a separation kernel, but this may not be necessary in highly fractionated systems where sufficient probabilistic separation guarantees between fragments are provided by their random distribution over the nodes and hence by the distributed nature of the system alone. Like a separation kernel, the fractionated software kernel should be minimalistic so that trust in it can be established once and for all with acceptable cost. The fragments will be required to be self-coordinating so that global coordination, a bottleneck, and a potential point of failure in component-based approaches to fault tolerance, would not be needed.

To exploit distribution, the kernel will also provide minimal but robust networking capabilities so that, disregarding possible networking delays, local fragments can interact with nonlocal fragments just as if they were local. The interaction will be delay tolerant and no upper bounds on delays will be assumed because network disruptions (including intermittent or episodic connectivity) are assumed to be part of the normal operation. To ensure maximal decoupling between the fragments, the interactions will not be direct (unlike remote procedure calls and the message-passing paradigm used for instance in today's service-oriented architectures) but rather enabled by partially ordered knowledge sharing. The idea, which has been successfully used in sensor networks and in disruption-tolerant networking (DTN), specifically in our work [75], is that each node has a local knowledge base, which local fragments can access and which is shared across the network using a peer-to-peer knowledge dissemination protocol. A general framework, which serves as a prototype to experiment with the partially ordered knowledge-sharing model, has been presented in [42] and used as the basis of a distributed logical framework in [74]. This loosely coupled paradigm resembles that of a distributed blackboard (generalizing the well-known blackboard paradigm for multiagent systems) and distributed tuple spaces (e.g., [61]) with the important difference that no global coordination and no consistency guarantees are required. Instead, fragments are engineered to be delay insensitive and tolerant to inconsistent and incomplete knowledge. Similar to the paradigm of content-centric networking [79], the fragments operate at a level of abstraction in what might be called a knowledge-centric approach, where they are not concerned with protocols and message flow (and resulting synchronization problems) but only with the question of how to use the knowledge once it becomes available.

In a fractionated networked system, software fragments solving the same or similar problems will be distributed over the networked nodes with a suitable degree of diversity and redundancy. Various approaches developed for software fault tolerance [78,54] can be utilized and combined to achieve diversity of fragments, in particular distributed n -way redundancy and n -version design [12]. We additionally propose to use randomization (exploiting both non determinism and concurrency) as an important source of diversity. Individual fragments should be self-checking [54] (in a rigorous sense) but thanks to fragment diversity and redundancy do not have to be self-correcting. Nevertheless, checkpointing,

restart, and recovery block techniques [78,54] can be used locally at the fragment level to improve robustness, but not at the distributed system level, where inconsistent knowledge is accepted as a normal operating condition. Furthermore, diversity does not have to be confined to the implementation level. Fragments (e.g., parameterized by complexity levels) that can accomplish similar functions with possibly different resource requirements are also desirable and have the advantage of providing not only diversity but also a potentially continuous high-dimensional trade space for system optimization. In spite of each individual fragment being bound to a fixed location, this approach with transparent knowledge sharing leads to location independence of the function provided by the fragments as a group. Note that this approach does not rely on any form of code or agent mobility (such as [61], which comes with its own set of problems, especially in networks that are dynamic and unreliable).

We envision fractionated software to be continuously maintained and evolving at runtime. In fact, complex and expensive future systems (such as fractionated satellites) may not only be deployed incrementally but design and deployment will likely become concurrent incremental activities to enable risk and cost reduction through partial deployment and early testing. Hence, beyond the addition of new nodes, the fractionated software kernel needs to support removal and installation of software fragments on existing nodes without system interruption. Since a new fragment will typically be installed remotely (e.g., from a ground station in the case of a satellite network) and on many nodes, the dissemination of fragments can utilize the same mechanism that is used to disseminate knowledge. Clearly, an asynchronous system cannot be upgraded globally in one step, but with a sufficient amount of diversity and redundancy an incremental distributed upgrade should be possible even without risking the interruption of an ongoing mission.

2.2 Reflective Simulation Capability

To support runtime assurance and related techniques efficiently we envision that a reflective simulation capability will be directly built as a layer on top of the fractionated software kernel. In our context, reflection means that software fragments and their encapsulated hardware are reflected as models that can support reasoning and optimization activities. Models are not constrained to a single level of abstraction. Furthermore, models can be executable and can themselves be viewed as fragments that can be composed to larger models. Executability of models is essential to enable the predictive runtime verification techniques discussed in the next section. Computational reflection is a well-known concept in computer science that has many applications [55]. It has been successfully implemented as part of the Maude system [16]. The importance of runtime reflection as an enabler of traditional (monitoring-based) runtime verification for safety-critical systems has already been recognized [50]. Here, we propose to generalize runtime reflection to open distributed systems, to multiple levels of abstraction, and to use it systematically as a basis for the implementation of a wide range of runtime assurance techniques.

3 A Declarative Foundation for Cyber-Physical Systems

A logical framework should serve as a uniform declarative interface to all capabilities of the NCPS. At the same time it should provide a semantically well-founded way to represent, manipulate, and share knowledge across the network. The logical framework should also serve as a basis for abstract models that take the form of logical theories and are continuously adapting to new incoming knowledge resulting from local or nonlocal observations.

3.1 Distributed, Quantitative, and Scalable Logical Framework

Various kinds of knowledge need to be expressed including models, facts, goals, and proofs — i.e., derivations of goals from facts. In NCPS, facts can represent sensor readings at specific locations, and goals can represent queries for information or requests to actors or actuators to perform certain actions. Although there are cases where a goal can be directly satisfied by a single local action, it is typically the case that distributed actions are needed and the more relevant feedback will be conveyed via a feedback loop through the environment. Such indirect feedback can consist of facts (representing observations) from multiple sensors that together can measure the progress toward reaching the original high-level goal. As a consequence, rigid top-down or bottom-up approaches are not sufficient for NCPS. Furthermore, models can have many different flavors ranging from precise physical models to qualitative commonsense models, and can include approximate and partial models of the real world based on observations. Combinations of different flavors are usually needed. For instance, a satellite as part of a network could utilize an approximate model for network connectivity combined with a precise orbit model based on Kepler’s laws and knowledge gathered by active exploration (e.g., beaconing for neighbor discovery) and passive observations (e.g., attitude determination).

A Logical View of Cyber-Physical Systems. Apart from a few notable exceptions such as cyberlogic [66], it is interesting to note that the distributed nature of today’s problems is rarely considered in the design of logical frameworks. For cyber-physical systems it is essential, since carrying out proofs may require cooperation across multiple nodes. In many cases, goals and facts cannot be matched locally. Consider an example of gathering certain information from a particular area under observation (e.g., from a sensor network on the ground that is part of a global network of UAVs and satellites). In an interest-driven routing protocol, such as directed diffusion [38], a node expresses interest for specific data by sending requests into the network. Data matching the interest is then drawn toward the node from which the interest originates. From a logical point of view, a goal, representing an information request, is injected and disseminated through the network. The goal is a logical formula expressing that the information needs to be of the required kind (content subgoal) and be delivered at the requesting node (delivery subgoal). A fact representing the presence of information at the source will match or satisfy part of the goal — namely, the content subgoal. Now

there is an incentive to route the partially satisfied goal with the requested content toward the interested application, since this will incrementally increase the *degree of satisfaction* of the overall goal and eventually complete the distributed proof. In other words, an interest-driven routing and many similar processes can be seen as distributed proofs and optimization strategies that try to bring facts and goals together.

To serve as a formal framework for NCPS, the logic must have the capability to express degrees of satisfaction so that both search and optimization become instances of a generalized notion of deduction. As a starting point, we propose to use a version of first-order logic with equality, real arithmetic, and degrees of confidence. The specific application domain will be reflected in the background theory relative to which the reasoning takes place and can also influence the search, reasoning, and optimization strategies employed at the higher layers (see Section 4). Due to the resource-constrained nature of many cyber-physical systems, trade-offs between expressiveness and efficiency need to be considered and a scalable logic — i.e., a logic with sublanguages and inference systems of adjustable complexity — would be the ideal solution. The language needs to go beyond propositional and Horn clause logic, since a functional sublanguage representing cost and utility functions with discrete and continuous parameters and functional parts of the models will be essential. Furthermore, predicates with discrete and continuous parameters are important to support predicate abstraction [31]. To support functional computation as part of reasoning and optimization strategies, the logic should be equipped with operational semantics — e.g., based on conditional term rewriting similar to that of equational specification languages such as Maude [16], which is key to combining abstract logical models with an efficient notion of execution.

The logical view of NCPS allows us to recast information collection, control, and decision problems as logical problems that are primarily centered around the duality of two kinds of knowledge: facts and goals. Various classes of distributed algorithms can be declaratively expressed using this duality. Proactive, data-driven, or optimistic algorithms are mostly concerned with the establishment of new facts from existing facts, hoping to satisfy the goal but considering it as a secondary aspect. Reactive, demand-driven, or pessimistic algorithms are primarily goal oriented, meaning that during their execution new subgoals are established based on existing goals, which eventually can be directly established using the facts. It is noteworthy, however, that many interesting practical algorithms (e.g., hybrid routing for sensor nets) are a mixture of different paradigms. Hence, in our logical framework, both facts and goals need to be treated on an equal footing together with corresponding forward and backward inference rules.

Toward a Robust Logic of Degree and Uncertainty. A logical model is an instance from a fixed model class represented by a common background theory. In most applications, we are concerned with incomplete information, and the model of the real world is not entirely characterized. Hence, we are almost always concerned with an entire class of models that are consistent with the facts to various degrees. Apart from the natural incompleteness of knowledge due to

partial observability, many sources of uncertainty exist in cyber-physical systems, including environmental noise, measurement errors, system perturbations, sensor and actuator delays, and clock drift. Networked systems exhibit further sources of uncertainty caused by delayed, outdated, incomplete, or inconsistent knowledge. Furthermore, uncertainties play a natural role in information fusion and probabilistic algorithms. The consideration of a class of models also allows standard logics to represent certain aspects of uncertainty, but the degree of uncertainty is not explicitly represented. A natural solution would be to use an instance of many-valued logic [30] that is sufficiently constrained to be consistent with common probabilistic [22,24], stochastic [17], and quantitative interpretations [81]. To enable expression of priorities between goals or their relative importance (e.g. to differentiate between hard and soft constraints), we furthermore need weighted formulas.

In cyber-physical systems, models, facts, and goals are continuously changing. Therefore, it will be essential to design a robust logical framework that can gracefully, incrementally, and efficiently deal with such changes. One possible approach is to maintain proofs explicitly at a suitable level of abstraction — e.g., as partial orders (as opposed to sequential proofs) capturing all dependencies between facts and goals. Proof maintenance will take advantage of the locality of changes and hence can improve the efficiency of automated deduction and constraint solving/optimization. For instance, an explicit partial-order representation of dependencies enables more sophisticated search and optimization strategies, such as conflict-driven backtracking and logical state composition strategies that do not assume centralized control (see Section 4.2).

Depending on the nature of changes, proofs can either remain valid, require local adjustments, or become entirely invalid. Clearly, the former case is preferred, which is why we suggest complementing proof maintenance with a notion of proof robustness that, when used as an optimization criterion, allows us to avoid fragile proofs whenever possible. Proofs can be fragile because they are based on rapidly changing or unstable facts or because they lack redundancy. Consider, for instance, the goals of maintaining network connectivity or sensor coverage. Clearly, proofs representing solutions that rely on stable facts about the neighborhood of a node are preferred. Furthermore, in dynamic environments, proofs can be carried out in a robust way that instead of relying on an individual fact, which could become a single point of failure, relies on an abstraction — e.g., a disjunction of independent facts representing coverage or connectivity via several neighbors that remains invariant under a larger set of network perturbations.

3.2 Model Synthesis and Adaptation via Distributed Monitoring

Models in our approach come in two flavors — namely simulation and logical. Simulation models are executable and represented by a set of software fragments. Such fragments can be direct reflections of implementation fragments, but they can also represent a more abstract version of the implementation. Additional simulation fragments can capture executable models of the environment of the NCPS, which for instance includes node mobility and networking capability.

Logical models are represented by a background theory together with facts that further narrow the relevant class of models, — e.g., by fixing or constraining model parameters. Depending on its level of abstraction, a logical model can have an underlying executable model.

In most cases, the models cannot be fully characterized in advance and can change while the system is in operation, which is why model adaptation is an essential ingredient of our architecture. Models (or their characteristic parameters) are shared just like other forms of knowledge, and this process is subject to the limitations of network connectivity and bandwidth, which leads to additional possibilities for delays, incompleteness, and inconsistencies.

Model synthesis and adaptation go hand in hand. Lacking other knowledge, the system can start with a default model (e.g., a single node cluster/constellation in our satellite application), which is incrementally refined during operation of the system. Knowledge can be passively accumulated by observations — e.g., from cyber-physical sensors — while the system is executing its primary function or mission, or it can be actively pursued by exploration, which may require physical actions. Often, combinations of the passive and active modes of model adaptation will be needed for acceptable performance with low exploration overhead. The specific exploration strategy is part of the system strategy so that trade-offs between exploration and exploitation of knowledge can be expressed as part of the overall system goal. The trade-offs between exploration and exploitation (of accumulated knowledge) are well known in the context of reinforcement learning [76], but are more challenging in our context due to constraints imposed by the model and goals, by resource limitations, and by the distributed nature of NCPS. In the case of satellites, even a relatively minor form of physical exploration by means of orbit adjustments can require significant resources (e.g., energy, time) and can be in conflict with the primary goal (e.g., maintenance of network connectivity). Network beaconing or probing (for node discovery or performance estimation) can be seen as another less expensive form of exploration of the environment.

In addition to these trade-offs at the strategy level, there are trade-offs that need to be considered when developing the models themselves. Even logical models can be executable in a sense. In fact, suitable abstract models can take advantage of the fragment of logic that supports symbolic execution, efficient deduction, search, and optimization. In the context of model adaptation, there is another reason why abstract models are often preferable as a basis for system control. In principle, models can try to precisely capture the reality such as mechanical models of motion or the path loss models used for wireless communication, but the parameter estimation needed to adapt such models to reality can be expensive or infeasible given the amount of data and sensing capabilities of cyber-physical systems. Indeed, in machine learning the notion that precise system identification is necessary for best performance has mostly been rejected [13]. Adaptation of simpler models has the advantage of requiring fewer data points, but predictions will be necessarily less precise. Still simpler models often lead to superior performance, because their lack of precision is compensated

for by their robustness under noise and their capability to generalize to new situations.

4 A New Generation of Runtime Assurance Techniques

A major hurdle in traditional system verification is the explosion of possible cases to consider due to lack of knowledge at design time about the particular system state or configuration at runtime. The flexibility offered by dynamic reconfiguration and retasking further exacerbates this problem. Furthermore, the typical NCPS we are interested in should support unanticipated missions which means that even the specification is not known at design time. To tackle these problems, we propose validation, synthesis, and verification techniques that can take place at system runtime when the best possible knowledge about the system state and the mission goals is available. Such runtime assurance techniques can exploit the knowledge about the current system state to focus the verification on what is currently relevant or relevant in the near future. Due to their very focused nature, the potential for state space explosion is significantly reduced and the savings in terms of resources can be substantial.

To avoid confusion, we should point out that the techniques we propose as research opportunities are significantly more dynamic than ongoing research in the field of (monitoring-based) runtime verification [51,1,11]. This area is not so much concerned with shifting design-time methods to system runtime, but mostly with a much more specific problem — namely, the construction of efficient (possibly distributed) monitors for a given fixed property. The property is known before the system is instrumented for monitoring, and it remains fixed during system runtime. Instead of considering self-verifying systems, the verifier is usually external to the system that is monitored, or more precisely is assumed not to have an impact on its behavior, an assumption that we cannot make for the resource-constrained systems in which we are interested. Furthermore, the properties of interest are expressed in a relatively weak temporal logic and intended to capture only specific aspects (e.g., null-pointer dereferencing or race conditions) of a software system. These state-of-the-art runtime verification techniques can be used offline (at design time) and online (during normal system operation). They have been implemented in various frameworks, such as Pathexplorer [65], Eagle [5], and MaC [47], and have been very successful in finding subtle software bugs. Most current techniques appear to be focusing on the code level. Interesting and notable exceptions are the component-based architecture [6] and AMOEBA-RT [27], which can verify adaptation properties of systems that transition between different regimes. Unfortunately, these techniques are not sufficiently expressive and dynamic for our purposes, where system properties and goals are very complex and constantly evolving. The approach [29] of using runtime monitoring to correct the global dynamics of systems (UAV swarms in this case) evolving according to the local rules of an artificial physics model is quite close to the spirit of fractionated software, except that the assumption of a global view and global control needs to be relaxed.

4.1 Probabilistic Runtime Testing and Verification

We sketch several new runtime testing and verification techniques that can provide probabilistic assurance that current and future states of the system satisfy the system goals. These can include specific mission goals, intermediate goals, and general or specific policy constraints and performance requirements. We envision runtime testing and verification techniques to be invoked by higher layers. Specifically, the distributed control and optimization strategy can be used to verify the violation or near-violation (e.g., by means of a slightly stronger property) of critical properties at present or future states. The set of future states will typically be bounded by a state- and property-dependent look-ahead horizon so that state space explosion can be kept under control. Given that runtime verification can be invoked repeatedly on the trajectory of the system, a tightly bounded look-ahead horizon is acceptable as long as it is sufficient to take corrective actions if the need arises. Runtime verification needs to be complemented by just-in-time validation (see Section 4.4), which will be applied at longer time scale.

Randomized Symbolic Verification. The most basic form of runtime verification without prediction can focus all resources on the present state. It uses the formal model together with the current goal to detect violations or near violations of critical invariants. The verification will typically involve global system properties, but it will be based on local knowledge (about local and remote states and about the state of the environment). As an extension of the basic technique, we propose to take into account the imprecision or lack of knowledge about the global state of a distributed system. To this end, one might consider performing runtime verification on a (symbolically represented) envelope (i.e., set of states) around the system state derived from the best knowledge available. Efficient symbolic SMT (satisfiability modulo theories) solving techniques such as those used in Yices [20] and efficient symbolic computation and deduction by conditional rewriting (modulo theories) as in Maude [59,16] are mature technologies on which to build on. If the symbolic representation reaches a certain complexity threshold, probabilistic assurance can be obtained by lifting the basic verification techniques to the set of states using sampling techniques. Each sample can be an entire symbolic region (a possibly infinite set of states) so that high coverage can be achieved with a relatively small number of samples. This probabilistic use of symbolic solvers opens a rich set of possibilities that to our knowledge have not been investigated in the past. Other very promising approaches to the integration of logic and sampling-based analysis techniques are Markov Logic Networks [64] as, for instance, implemented in the Probabilistic Consistency Engine [3] (PCE), which can be used to quantify the probability that a property holds or that the system is in a certain state (e.g., based on partial observations).

Dynamic Runtime Model Checking. A complementary direction is to exploit the time dimension by using available computational resources to predict the future evolution of the system up to a certain time horizon, which needs to be short enough to avoid state space explosion. One category of prediction-based assurance methods would use model checking at runtime. Model checking

can be directly applied to executable logical models as the Maude LTL model checker [23] demonstrates. Some early work exists on runtime model checking of safety properties for multithreaded programs [86], which incorporates interesting ideas on dynamic partial-order reduction (to further reduce the state space to be explored). There is also work on guiding a model checker based on a runtime analysis of programs [35], and a next consequent step would be to perform both analysis and model checking at runtime. Lifting these ideas from program code to higher levels of abstraction is an opportunity worth exploring. Model checking would be performed locally but based on the continuously adapting models of local and nonlocal behavior. Since the time horizon is limited, bounded-model checking techniques, which can be implemented using efficient SAT solvers, would be of interest as well. Furthermore, statistical model checking techniques [48] that can deal with probabilistic models (e.g., represented as Markov chains) and probabilistic properties could be applied at runtime. A major challenge with all these approaches is to develop efficient runtime algorithms that can scale with the available resources and to explore how model checking can be distributed (some ideas from [72] about distributing formulas can be useful in this context) and take advantage of the fractionated nature of our systems.

Dynamic Directed Monte Carlo Analysis. Simulation-based methods for distributed systems are well suited for quantitative analysis, but since they do not provide full coverage, they are inherently probabilistic in nature. Monte Carlo techniques can be used to account for imprecision of models and of the global state. The advantage of using such techniques at runtime rather than design time is the potential for a much more directed application exploiting the additional knowledge available, which directly translates into resource savings and/or precision improvements. Furthermore, by generalizing statistical verification via hypothesis testing, the number of samples can be dynamically adjusted based on the actually required confidence that may be known only at runtime. Black box statistical model checking generalizes hypothesis testing to temporal properties [73,88], has been extended to quantitative properties [80], and has been applied to verifiable cross-layer adaptation in our own work [44,43]. In a fractionated architecture, Monte Carlo techniques such as these will naturally scale, and the precision improves (or alternatively the local load will be reduced) with the number of nodes. Several unexplored extensions of this scalable simulation-based approach will also be of interest. First, the use of the current state as a starting point can be relaxed, by focusing the verification on interesting, critical, or recurring states. Machine learning techniques could acquire such states (and their distribution) during the lifetime of the system. Monte Carlo simulations on such states can then be continuously executed in the background, possibly controlled by resource availability. Monte Carlo simulation can furthermore be biased to explore performance extremes (runtime stress testing), rare (e.g., black swan) events, or high-risk situations (based on runtime risk assessment). Markov-Chain Monte Carlo (MCMC) techniques, which are at the core of PCE [3], are of interest as

well due to their capability to efficiently sample from complex joint distributions. By making the temporal dimension explicit in the model, a tool such as PCE can also be used to perform temporal analysis (with a reasonably short look-ahead horizon).

Integrating Formal Methods and Simulation. Given that deduction, model checking, and simulation-based techniques each have their own advantages, a natural question is if these approaches could be integrated into a single hybrid runtime verification technique. One possibility is through executable formal methods such as Maude [16], an idea that we explored under the name *formal prototyping* in the context of fault-tolerant middleware [33] and security [28]. Another unexplored possibility is based on a notion of abstraction. An abstraction essentially replaces subsets of system states by representatives so that the complexity of verification is further reduced. Given such an abstraction we can use simulation-based techniques to evaluate the performance of a system in each abstract state (or a relevant subset) by using detailed simulation models. Model checking and similar formal methods can then take place at the abstract level, and verify properties such as if a certain level of performance can always be maintained in certain situations that can be expressed logically. A more intelligent integration might trigger the underlying simulation on the fly only for states that the model checker explores. Furthermore, the number of samples could be determined by the required confidence level, which essentially means that resources are directed to the properties that matter. It should also be noted that model checking can naturally be used as part of a deductive system [71] and hence fits naturally with the higher-layer runtime assurance techniques that we will discuss subsequently.

Adaptive and Probabilistic Runtime Abstraction. We have seen that runtime assurance can greatly benefit from knowledge that is not available at design time. Learning the reachable or relevant states of the system is just one dimension to exploit. A second dimension is concerned with the problem that the properties that need to be verified may become available or sufficiently concrete only at runtime. A third orthogonal dimension is to use runtime information to determine and adjust at runtime the level of abstraction where other techniques are applied. For instance, the combination of model checking with abstraction [15] has attracted a lot of attention for the purpose of design-time verification. Finding the right abstraction, however, is not easy, and currently done by repeated model checking with counterexamples-guided refinement. Without being confined to model checking, a yet unexplored but related idea of violation-driven refinement could be used at runtime to choose a suitable level of abstraction for monitoring properties of interest. Runtime techniques would furthermore enable an alternative and more efficient approach to learning or synthesizing abstractions from observations of real system dynamics. The key idea is that states with similar observable properties at runtime do not have to be distinguished even if there is a theoretical possibility that they behave differently under some condition that the system will never reach. More generally, verification techniques can benefit from a notion of probabilistic runtime abstraction, where the correctness

of the abstraction (and corresponding abstract models) is empirically established with quantifiable probability and confidence.

Resource- and Situation-aware Runtime Assurance. Based on the environment, available resources, or timing, one may employ different runtime verification strategies. A flexible, fractionated architecture together with model-based distributed control and optimization strategies can facilitate switching among them. The general idea is to focus on features that matter in a particular situation but also partition the resources between the primary function of the system and the different runtime assurance techniques in a way that takes into account the various trade-offs in this space. For example, when a system (e.g., a spacecraft) is launched for the first time, one may allocate more resources for performing runtime monitoring, testing, and verification to ensure that the system functions properly. At a later stage, when the system has run for some time without issues, one may reduce the frequency or depth of runtime assurance to conserve resources or to reallocate them for other purposes. Another example of situation-aware runtime assurance is that the amount of runtime monitoring, testing, and verification (e.g., of security policies) performed may depend on perceived threats (e.g., attacks from adversaries). Based on the threat level, one may reconfigure the system to deploy a more comprehensive monitoring posture and, possibly additional, what may be called *defensive fragments*.

Learning from Failures and Near Failures. Failures of software fragments and violations of properties at runtime, even if corrected, can tell us a lot about future risks. Near failures and near violations (whether they are determined with or without prediction) are another possible source of critical states that should not be ignored. Reachable and critical states (and possibly their distribution) can be learned at runtime and analyzed in the background using directed runtime simulation and verification techniques. However, there is another unexplored opportunity here — namely to learn how to recognize similar critical situations and use runtime state avoidance techniques to circumvent them in the future (at least during a critical mission). The distributed learning of such states can happen as a generalization of distributed monitoring, which is needed in a mission-critical system for many reasons, but in particular to guide the continuous maintenance, improvement, and evolution of the system over its lifetime and over future generations. More generally, machine learning (especially statistical learning theory) is an area with a rich set of techniques that can contribute to new runtime assurance techniques in many ways (e.g., learning of specifications and model-based prediction), but we have touched upon only a few examples due to space limitations.

4.2 Model-Based Distributed Control and Optimization

System control and optimization in NCPS is challenging. The control of runtime assurance techniques and the consideration of the trade-offs of potential adaptations and countermeasures must be performed in context of the overall system

goal, in which quantitative aspects will typically play an important role. Traditional optimization techniques that strive for optimal solutions based on precise models are not suitable for most NCPS, where models have many dimensions of uncertainty, and optimality in the strict sense is neither desirable nor achievable. What is needed in practice are strategies to find acceptable and robust solutions, sufficient to achieve the goal while taking into account the limitations of the models and available resources.

On the other hand, the fractionated nature of our system offers many advantages including fault tolerance, distributed sensing, coordinated actions, and inherent parallelism for computational processes that should be exploited not only for the primary function of the system but also by the runtime assurance techniques, and in particular the control and optimization strategies. Clearly, a top-down decomposition of the overall goal in a divide-and-conquer fashion is not a viable approach, because solutions may require ad hoc cooperation across layers and across nodes. This and related problems of strictly layered approaches have led to the recent trend of cross-layer design and optimization in networking (and especially sensor networks). Among other sources (see Section 6), our xTune architecture for cross-layer control and optimization based on the runtime application of formal methods has served as an inspiration for the following ideas.

Closely related to the idea of combining runtime verification and model-based control is the area of model predictive control also known as *receding horizon control*. In a discrete setting, it has been successfully applied, for instance, in NASA's Livingstone [83], a kernel for a self-reconfiguring, reactive, and autonomous system. It combines model-based diagnosis and a propositional controller, an idea that has been further generalized to model-based programming in [82], based on a language that can be compiled into hierarchical constraint automata. Another model-based architecture that was the basis for our framework [18] for goal-oriented operation of remote agents [62] is JPL's Mission Data System [21], a unified flight-ground control and data system. Protection against faults and dealing state uncertainty are noteworthy features. None of these architectures, however, were aiming at loosely coupled, highly distributed and fractionated NCPS that lead to many new challenges as we explain below.

Control and Optimization as Logical Strategies. Mathematically, the logical framework and its underlying fractionated computing paradigm allow a rich set of conceivable behaviors that need to be constrained to a subset that satisfies the system objectives. We suggest developing strategies that control and optimize the operation of NCPS based on its declarative representation in the logical framework with the idea that the generated control actions are correct by construction. These strategies will be resource-aware and adaptive. For example, in homogeneous scenarios, our strategies can exploit the parallelism of many nodes so that resource consumption at each node can be low. In heterogeneous cases, they can exploit powerful or energy-rich nodes that perform heavy computations so that low-power nodes can save their resources. Nodes will try to share knowledge and cooperate while communication conditions are good, but if communication is impaired or disrupted, nodes will tend to operate more autonomously. If knowledge

including facts, goals, and solutions can be shared using a framework based on partially ordered knowledge sharing such as [42], the location of computations is flexible to a large degree and limited only by the communication and node capabilities. A distributed logical framework that could serve as a basis for this approach is currently being explored in [74] and [46].

Ideally, the strategy exploits parallelism inherent in search and optimization problems, by allowing nodes to sample the search space independently. Unlike numerical approaches, sampling can be done symbolically, by randomly generating new subgoals that represent entire regions of potential solutions in a finite way. The sampling heuristics can be biased by a nonuniform distribution to express locality and preference for solutions that can be reached more easily or with lower cost. In addition, the cost of reaching a solution can be explicitly quantified and constrained by the system goal. The best stable solution will be shared opportunistically across the nodes and is ultimately used to drive the local actions of NCPS. The symbolic sampling strategy explores the search space of potential solutions, but conflicts can arise, possibly after several subsequent reasoning or constraint-refinement (i.e., narrowing down the solution region) steps performed using the logical framework. Conflicts can manifest themselves either as logical inconsistencies or nonacceptable solutions. One possibility to deal with conflicts is by local randomized backtracking driven by the conflict itself, exploiting the dependencies maintained by the underlying logical framework. A randomized approach to search and optimization tends to avoid redundant computations (i.e., the same computation at several nodes) under cooperative conditions, but would not rule out redundant computations that are essential for progress if nodes need to operate autonomously.

In traditional approaches to planning and optimization, the process terminates when an acceptable solution is found and leaves it to lower layers to take the actions to implement and fine-tune the solution. NCPS, however, need to be continuously controlled and optimized. The continuous optimization will consider the most recent known state of the distributed system and hence can quickly adapt to changing facts and goals. Even if actions have been already taken toward the transition into an acceptable solution region, a significantly better solution might emerge either because the solution was not explored until now due to computational resource limitations or because it arises due to new unexpected conditions, including failures preventing the system from reaching the solution it was aiming at in the first place.

Abstraction as the Key to Robustness and Composability. A logical approach to optimization would also enable the composition of (partial) solutions. Rather than aiming at a numerical point solution each node narrows down the goal to one or multiple solution regions represented by logical formulas. If two nodes establish connectivity, the goals will be composed by a logical conjunction resulting in a goal that semantically corresponds to the intersection of solutions acceptable for both nodes. The approach can be generalized to entire groups of nodes that merge due to a network topology modification. There is a natural connection between abstraction, robustness, and composability. Composability

is enabled by a suitable level of abstraction that avoids over-constrained point solutions. In other words, solutions are robust enough to accommodate, at least to some degree, the needs of other nodes. The use of an abstract solution region reduces the likelihood of conflicts in the case of composition, but clearly cannot exclude this possibility entirely. Conflicts caused by composition can be treated just like any other conflicts arising during search and optimization.

4.3 Distributed On-demand Deductive Synthesis

Techniques to synthesize software based on a declarative specification have a long tradition in what is sometimes called automated software engineering. NASA's Amphion [53] is one of the well-known projects where automated synthesis has had a large impact in the reduction of labor-intensive software engineering activities. The Amphion system, which is still in use at NASA today, generates scientific programs as a composition of subroutine libraries. Since synthesis is based on deduction in a sound logic, in this case the first-order logic of the SNARK [2] automated theorem prover, the solutions are correct by construction.

As with the other proposed techniques, we propose to shift the synthesis process to system runtime. More specifically, we propose on-demand synthesis whenever a new mission or policy goal requires a solution that cannot be implemented by a single (coordinated) action but requires a certain degree of planning with intermediate goals. The solution would consist of a set of activities or components suitably instantiated, parameterized, and composed to achieve the overall goal. The bigger challenge is, however, to perform the synthesis, like control and optimization, in a process that exploits the loosely coupled fractionated computing paradigm, and furthermore the solution generated by the synthesis should be distributed in the same sense.

To illustrate the logical inferences in a distributed deductive synthesis process, consider a greatly simplified example of intelligent surveillance. Assume that each satellite in a fractionated system is equipped with only one kind of capability, either a high-resolution camera or a motion sensing capability that is implemented on the basis of measurements received from a sensor network on the ground. Assume that predicates $Motion(a, t)$ and $Pattern(a, t)$ are true if a movement or a particular pattern has been detected in an area a at time t (approximately). Assume furthermore that $Image(I, a, t, t')$ means I is an image of area a taken in the interval t, \dots, t' , and $Delivered(I, r)$ means that the information I has been delivered at r . Now the following goal is injected at a ground node r :

$$Motion(a, t) \vee Pattern(a, t) \Rightarrow \\ \exists I : Image(I, a, t, t + \Delta t) \wedge Delivered(Extract(Abstract(I)), r)$$

It expresses that an image needs to be taken of a specific area a with maximum delay Δt after a motion has been sensed or a visual pattern has been recognized. The image then should be delivered to r after abstraction and feature extraction. After the goal is disseminated in the network, each node tries to solve the goal.

Let us now assume that a node above area a generates a fact $Motion(a, t)$ that can be used by another node that is monitoring that area and is equipped with a high-resolution camera to simplify the goal to

$$Image(I, a, t, t + \Delta t) \wedge Delivered(Extract(Abstract(I)), r)$$

so that the only way to make progress is to take an image i to satisfy $Image(i, a, t, t + \Delta t)$ leading to the remaining goal

$$Delivered(Extract(Abstract(i))), r)$$

Let us assume that the abstraction $i' = Abstract(i)$ can be performed immediately after taking the image but feature extraction will be performed at a more powerful node, say at the ground station, because it is computationally expensive. This node will then simplify

$$Delivered(Extract(i'), r)$$

after performing the computation $i'' = Extract(i')$ to $Delivered(i'', r)$, which can be incrementally solved by moving $Delivered(i'', r)$ closer to r , the requesting ground node, where it is finally realized by a delivery action.

A similar but more detailed example of a logical theory for distributed surveillance using a team of mobile robots can found in [74]. In spite of its simplicity, this example exploits three dimensions (computation, abstraction, and communication) of distributed computing, and yields a solution that is synthesized on the fly and correct by construction based on the soundness of the underlying logical framework. It furthermore illustrates the combination of logical inference and partial evaluation and their generalization to the distributed setting in which goals and facts can be bound to actions at different locations in the cyber-physical world. In a more complex example, we might easily imagine that $Motion(a, t)$ and $Pattern(a, t)$ cannot be satisfied using the current distribution of nodes so that some nodes will have to adjust orbits to achieve sufficient coverage of area a . Clearly, this opens a rich trade space of possible solutions, which can be tackled by the combined capabilities of distributed deductive synthesis and the distributed control and optimization strategies discussed previously.

4.4 Predictive Just-in-Time Validation

It is well-known (but often forgotten) that the correctness of a system w.r.t. its specification is not sufficient to guarantee that the system operates as expected and is suitable for a given mission. The problem is that a high-level specification of the system goals, even if it is declarative and far less complex than the implementation, is complex enough that it is difficult for humans to judge whether it captures their intent. Inconsistencies (e.g., logical contradictions in the extreme case) or incompleteness (e.g., missing key properties) are very common. Hence, verification needs to be complemented by validation techniques that can increase the users' confidence in the specification. Clearly, another level of

verification relative to even higher-level specifications cannot be the answer, because the fundamental problem would be just postponed. Instead, we propose a simulation-based approach, which includes the runtime assurance techniques of all layers and as motivated earlier would be executed just in time, whenever the system goals are modified as a consequence of user interactions. Just-in-time validation has the advantage that the specification can be very specific to a particular mission, eliminating many possible use cases of the system that are simply not relevant. Thanks to its simulation-based nature, the results of the validation will be quantitative rather than simple yes/no answers. Quantities are not limited to probabilities of properties being satisfied but can include expected performance metrics and bounds. Furthermore, counterexamples in terms of property-violating (or just risky) executions can be fed back to the user who then has many options to respond, ranging from adjusting or replanning the mission to reallocating resources or scaling up the system capabilities (e.g. by additional launches in the case of a fractionated satellite mission). Since the simulation is performed by the highest layer, it may include the execution of the embedded runtime assurance techniques, and as a consequence its coverage and capability to detect problems is higher than that of conventional simulation techniques without embedded verification.

Predictive just-in-time validation has to cover the entire distributed system as well as all layers of the architecture with a time horizon that covers or is at least representative for the entire mission. Hence, predictive just-in-time validation can be computationally resource intensive and probabilistic simulation-based techniques are preferable. Given that mission validation can be time critical, the parallel nature of probabilistic simulation techniques will be an important advantage. As with all runtime assurance techniques, a fractionated software architecture leaves a lot of freedom regarding where the actual simulation is carried out. In case of a networked satellite mission, it would make sense to utilize a computing grid on the ground to perform a large number of such simulations around an approximation of the current state of the system, which is always available by means of adaptive models. In other words, we continuously maintain a virtual approximation of the real system that is used for just-in-time validation whenever the system needs to be configured for a new mission. Clearly, multiple concurrent overlapping missions by multiple users of the cyber-physical infrastructure are particularly interesting, because the effects of sharing limited resources will be predicted by the validation process, and the injection of a new mission into the system may be rejected because of resource limitations.

Since predictive just-in-time validation can also be applied at design time (although at a higher computational cost due to the more limited knowledge about the future system state and configuration), it should be general enough to subsume existing validation and performance evaluation techniques, namely discrete-event (network) simulation and hardware-in-the-loop simulation techniques. The current practice is still centered around the use of a variety of simulation tools (such as Matlab, Qualnet, or the STK satellite modeling toolkit) to capture different aspects of the system under evaluation. However, the

diversity of tools and their different levels of abstraction often leave a significant gap between the real system and the model that is evaluated. Keeping the simulation models in sync with the actual code is a labor-intensive and failure-prone task and the confidence that the simulation captures all important aspects is usually based on experience and subjective judgment. In our proposed reflective architecture, simulation models are first-class concepts, so that runtime and design-time validation and evaluation can use the same set of models, which at the lowest level of abstraction can be identical to the actual implementation, thereby reducing the modeling gap.

5 Illustrating Example: Fractionated Satellite Networks

Consider a network of fractionated satellites that has already been deployed in space and needs to be retasked rapidly, i.e., within hours, for surveillance of a particular geographic region during a crisis. To accomplish this, the satellites need to perform coordinated orbit adjustment maneuvers to provide sufficiently good coverage of the areas of interest with a frequency that satisfies the mission requirements. Specifically, we chose a primary goal, such as the collection of information (e.g., images) from a particular area, that can be achieved only by actively morphing and expanding the network topology — e.g., by tethering (stretching the network in a particular direction) possibly with some redundancy to reduce the likelihood and duration of disconnections. Various essential policy and system goals concerning sensor coverage, network connectivity, or energy consumption can be active at the same time in addition to the primary user objective.

Now suppose there exists a (previously) unknown bug in the image processing software that manifests itself when it processes data pertaining to a very small number of (geographical) coordinates. Using runtime verification, the satellite may be able to discover the bug and take corrective actions to avoid the problem. Specifically, based on the current coordinate of the satellite and its trajectory, the runtime verification system discovers that the image processing software fails when it reaches a certain coordinate. The satellite finds several possible solutions to mitigate the problem. First, the satellite may change its trajectory to avoid the problematic coordinate. Second, the satellite may stop functioning temporarily when it reaches the problematic coordinate, while having other satellites to handle the area it is supposed to cover. Third, the satellite has another implementation of the image processing software module that does not have the bug, and the satellite replaces the faulty software with it. After evaluating the costs and benefits of the options, the satellite chooses the most cost-effective one.

Several variations of this sample mission would lead to more challenging test cases pushing runtime assurance techniques to their limits. Hardware and software fragments could be instrumented to fail continuously with unusually high rates during the mission (simulating a combination of software and hardware faults), and the high-level system objective and performance still needs to be maintained without interruption by agile system adaptation. Also, the dynamic

improvement of the capabilities by launching new nodes, as well as simulated network partitioning, merging, perturbations, and the loss of nodes, is a rich source of test cases for system robustness. Other possibilities include considering more complex system goals with partially conflicting multiuser objectives, policies, and corresponding trade-offs. In addition to energy, system goals can involve timing constraints, quantification of QoS and robustness (e.g., of network connectivity), and consideration of risks (e.g., of losing nodes) and options (e.g., flexibility to react to new mission goals). Finally, the resource-adaptive distributed operation in a nonhomogeneous global network — e.g., a combination of small satellites, UAVs, a ground sensor net, a ground station network (with fixed and mobile nodes), and powerful grid nodes in the Internet — would be an ultimate test case in system-of-systems interoperation.

6 Background and Related Work

For an up-to-date overview of our ongoing work on Networked Cyber-Physical Systems and a large body of background literature that is beyond the scope of this paper we refer to [63]. In the following we limit ourselves to a few selected research directions and projects that had a significant influence on our suggested approach.

Delay- and Disruption-Tolerant Networking (DTN) and Sensor Networks. DTN [25] enables communication in challenging environments where many NCPS are deployed. Underwater sensor networks [32], wildlife tracking [58], vehicular networks [52], satellite networking [39], and interplanetary deep space networking [10] are just a few examples demonstrating the wide range of applications. By combining network caching and routing on an equal footing, DTN can overcome intermittent connectivity, such as in highly dynamic networks of mobile nodes or in sensor networks that are scheduled for energy efficiency. *Space-Time Adaptive Networking Architecture (STAN)*, which we have recently proposed as a small-footprint solution for small satellite networks (such as those based on the Cubesat [77] platform), further improves upon existing DTN architectures. STAN is a true *cross-layer architecture* that leverages adaptive and predictive models for intelligent power-management, caching, and routing. The example used in this paper captures several interesting aspects of DTN and STAN if applied to fractionated satellite networks. In a limited form, some of the ideas proposed as research opportunities in this paper are present in our earlier work in the context of DTN. For instance, our *reflective routing algorithm* [75] increases the probability of delivery based on a reflective and predictive logical model of the distributed system. Furthermore, a special kind of runtime abstraction, coined *self-organizing abstraction*, of dynamic networks has been used in our recent work to increase performance of disruption-tolerant routing.

Constraint Solving, Optimization, and Distributed Approaches. The borderline between constraint satisfaction and optimization has mostly disappeared due to

the need to judge the quality of solutions for efficient search. Recent advances in SAT solving also show that logical approaches to SAT solving can be naturally extended to optimization problems such as MaxSAT [49] and MiniMaxSAT [36], which supports weighted clauses. Much progress has been made on moving from propositional logic to more expressive fragments of first-order logic as witnessed by recent SMT solvers such as Yices [20]. Unlike this line of work, which aims at completeness and optimality, our approach aims at sufficiently good results for more expressive fragments of first-order logic and our quantitative extension. In spite of their limited expressiveness, modern SAT/SMT solvers became powerful enough to realize the idea of viewing planning as a satisfiability and optimization problem [41,34]. Some evidence that higher expressiveness can be very practical with acceptable trade-offs is provided by our work on software-defined radios, in which we developed a policy logic and a constraint-based reasoner for dynamic spectrum access [84].

Some recent research has been conducted on parallelizing SAT solving [7], GridSAT [14] being one implementation. In earlier work, randomized backtracking has been proposed as a mechanism for a parallel Prolog implementation [40]. These parallel approaches are mainly concerned with performance gain, possibly fault tolerance, but do not cope with the inherently distributed nature of the problem, which is crucial in many NCPS.

It seems, however, that distributed algorithms offer this promise. *Distributed constraint satisfaction* (DisCSP) and *optimization* (DisCOP) problems have been investigated in the context of multiagent systems [87]. Some common algorithms are distributed versions of their centralized counterparts, like local annealing [26], distributed hill climbing [56], distributed stochastic search and distributed breakout [89], or ADOPT [60], which performs distributed depth-first backtracking based on a fixed variable ordering. One of the most interesting algorithms is OptAPO [57], which is not simply the adaptation of a centralized algorithm, but is based on a dynamically selected mediator, which internalizes a larger part of the problem and helps to solve conflicts. In spite of their asynchronous nature, all algorithms use classical multimessage protocols. A bigger problem is that the DisCSP/DisCOP assumptions (finite domains, reliable communication) are not satisfied for many NCPS. Nevertheless, there are interesting aspects of DisCSP/DisCOP solutions with potential to generalize. For instance, the idea of *mediation-based cooperation* has served as another source of inspiration for our loosely coupled approach, where every agent internalizes (part of) the problem and can therefore act as a mediator and disseminate the new solution state. The difference is that in our approach this happens opportunistically (and using an expressive logical framework) rather than as part of a multimessage mediation protocol.

Compositional Cross-layer Optimization. It is widely accepted that cross-layer optimization, e.g., involving physical, medium access, and routing layers, is a key technology for resource-efficient networking. The idea of using formal methods at system runtime has recently been applied to *compositional* cross-layer optimization [45] in the context of the xTune framework [85]. In xTune, we have the

classical optimization objective of finding suitable parameter settings at each component based on a *utility* function capturing the *effectiveness* of the settings relative to the user and system objectives. For example, utility can be a function of energy consumption, timeliness of operation, quality of service, bandwidth demand, and buffer capacity requirements. In xTune, we achieved cross-layer optimization by constraining the behavior of local optimizers working at all abstraction layers (application, middleware, operating system, hardware architecture) that are connected by a *vertical* composition. Each local optimizer uses the other optimizer’s refinement results as its constraints. Thus, the constraint language serves as a common interface among different local optimizers, leading to improvements of solution quality, robustness, and speed of convergence. Compositional optimization through constraint refinement enables a controller to coordinate existing local optimizers, which can accommodate different objectives, by treating them as black boxes. The control and optimization strategies that we discussed in this paper can be seen as a generalization of the compositional constraint-refinement approach to include *horizontal* composition capturing the distributed nature of NCPS.

Constraint-based optimization can be entirely generic or guided by a model of the system to optimize. Aiming at the latter case, we can build on our experience with probabilistic runtime analysis [44] and tuning of abstract cross-layer models [37,45,43] specified in the formal modeling framework Maude [59,16] that is based on the notion of executable specifications. Statistical analysis techniques have also recently been integrated into our cyber-application framework [42]. In this paper, we propose to move from purely local reasoning, statistical analysis, and model checking techniques toward distributed compositional techniques that integrate randomization and symbolic reasoning. Compared with our earlier work [45] the constraint language would become part of an expressive logical framework that can support strategies for distributed cooperative constraint refinement.

7 Conclusion

It is our belief that traditional techniques for the verification and validation of complex distributed software systems are trapped in an unsatisfactory local optimum, and significant progress is possible only by fundamentally rethinking the way distributed software is designed. Today’s distributed software, in the best case, is based on a rigid composition of relatively tightly interacting coarse-grained components. This makes the entire system prone to low-level faults of many different kinds, and the sheer number of possibilities to consider (not only due to faults) makes verification and validation prohibitively expensive. The discrete and non-scalable nature of conventional software makes it furthermore difficult to build trustable distributed systems that are adaptive and dynamically reconfigurable in a flexible manner. The intuition behind fractionated software is to transform software into a more fine-grained, more continuous form (figuratively speaking, more like a flexible fluid than a rigid composition of bricks) that

like biological systems leverage diversity and redundancy to achieve a high level of robustness against low-level faults. As a by-product, fractionated software can also be better distributed, scaled, controlled, and optimized especially as part of NCPS that need to interact with the continuous physical world.

The biggest challenge in moving from the traditional coarse-grained to extremely fine-grained concurrency with self-coordination is the overhead associated with the mapping of a large number of small concurrent computation threads and their interaction on today's computer and network architectures. First experimental results with a prototype implementation of our partially ordered knowledge-sharing model have been reported in [42]. This prototype makes use of thread pools, shared memory and multicast capabilities of the network to support a large number of distributed fragments. Using a case study of evolutionary optimization algorithms we evaluated the scalability of our model using a small number of PlanetLab multi-core hosts, but the granularity of concurrency needs to be further decreased to approach the vision of truly fractionated software. At the same time the number of fragments will increase, and technical solutions (ideally at the OS level) need to be developed to more efficiently map a large number of threads to a large number of computing cores connected at various levels (ranging from shared memory to potentially unreliable networking technologies). Considerations of efficient use of caching (in the presence of a large number of threads) as well as low-overhead networking protocols that implement the knowledge-sharing paradigm in a more direct way would be important to explore in the future.

Once a foundation for fractionated software is available, trustable systems can be built by applying suitable verification and validation techniques at the right level of abstraction and at the right time. We have argued that the right level of abstraction for such systems is the macroscopic level of system properties rather than the microscopic code level that is encapsulated in software fragments, which becomes nearly invisible if the degree of diversification and redundancy is sufficiently high. The right time is the system runtime for the flexible mission-critical systems of interest, when the best possible knowledge is available. Hence, we have suggested numerous research opportunities for new runtime assurance techniques that cover the entire spectrum from validation to synthesis, verification, and testing. Different from today's practice, which mostly relies on subjective judgment, confidence in critical properties should be probabilistically quantified, whether empirically or through models, should be explicitly maintained, and needs to flow through the system along with the invocation of assurance techniques at runtime.

An explicit declarative representation not only of the mission objective, policies, intermediate goals, and performance requirements, but also of the NCPS and its models, is a key feature of our suggested approach, because it allows us to use runtime techniques to generate solutions or actions that are correct by construction. We view correctness as just one dimension in a high-dimensional trade space among many other performance metrics, and we accept that it can be achieved at reasonable cost only by a dynamically balanced set of techniques.

Hence, it is essential that distributed control and optimization strategies steer the application of runtime assurance techniques as part of the primary system function in a rational way, enabling the system to operate and respond based on available resources, performance goals, and trade-offs. As an illustrating example we have used the mission-driven operation of a fractionated satellite network because it comes with many facets and challenges, especially in terms of fault tolerance and dynamic reconfigurability, that are far beyond the scope of today's verification and validation techniques.

With the idea of fractionated software we are prepared to exploit the growing trend of distributed and parallel hardware, e.g., in the form of large-scale networks of powerful many-core (rather than multicore) processors. With fractionated software we would also be prepared for a possible future where hardware becomes much less reliable — e.g., due to further miniaturization down to the nanoscale or, more speculatively, where reliability is given up completely as a hardware design goal in favor of extreme parallel performance and/or energy efficiency. On the other hand, a much more concrete opportunity can be found in the domain of our proposed case study. For reliability reasons, spacecraft are usually based on previous generation low-performance processors (often radiation hardened), but the combination of fractionated hardware and fractionated software, which does not rely on the reliability of its fragments, would open an entirely new space of exciting possibilities in terms of cost and performance.

Acknowledgments. Support from National Science Foundation Grant 0932397 (A Logical Framework for Self-Optimizing Networked Cyber-Physical Systems) and Office of Naval Research Grant N00014-10-1-0365 (Principles and Foundations for Fractionated Networked Cyber-Physical Systems) is gratefully acknowledged. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or ONR.

References

1. <http://runtime-verification.org/>
2. <http://www.ai.sri.com/~stickel/snark.html/>
3. PCE User Guide, Version 1.0. Technical manual, Computer Science Laboratory, SRI International (July 2009)
4. Akyildiz, I.F., Kasimoglu, I.H.: Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks* 2(4), 351–367 (2004)
5. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) *VMCAI 2004*. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
6. Belhaouari, H., Peschanski, F.: A lightweight container architecture for runtime verification. In: Leucker, M. (ed.) *RV 2008*. LNCS, vol. 5289, pp. 173–187. Springer, Heidelberg (2008)

7. Blochinger, W.: Towards robustness in parallel SAT solving. In: *Parallel Computing: Current & Future Issues of High-End Computing*, Proc. Int. Conf. ParCo 2005, pp. 301–308 (2005)
8. Bloomfield, R.E., Littlewood, B., Wright, D.: Confidence: Its role in dependability cases for risk assessment. In: *37th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks*, DSN 2007, pp. 338–346 (2007)
9. Brown, O., Eremenko, P.: Fractionated space architectures: A vision for responsive space. In: *4th Responsive Space Conf.* (2006)
10. Burleigh, S.: Interplanetary overlay network: An implementation of the DTN bundle protocol. In: *Consumer Communications and Networking Conf.* (2007)
11. Watterson, C., Heffernan, D.: Runtime verification and monitoring of embedded systems. *IET Software* 1(5), 172–179 (2007)
12. Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995, ‘Highlights from Twenty-Five Years’ (1995)
13. Cherkassky, V., Mulier, F.M.: *Learning from Data: Concepts, Theory, and Methods*, 2nd edn. Wiley-IEEE Press (2007)
14. Chrabakh, W., Wolski, R.: GridSAT: A Chaff-based distributed SAT solver for the Grid. In: *SC 2003: Proc. 2003 ACM/IEEE Conf. Supercomputing*, p. 37. IEEE Computer Society, Washington (2003)
15. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50(5), 752–794 (2003)
16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: *All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic*. LNCS, vol. 4350. Springer, Heidelberg (2007)
17. James, C.: Stochastic logic programs: Sampling, inference and applications. In: *UAI 2000: Proc. 16th Conf. Uncertainty in Artificial Intelligence*, pp. 115–122. Morgan Kaufmann Publishers Inc., San Francisco (2000)
18. Denker, G., Talcott, C.L.: A formal framework for goal net analysis. In: *Workshop on Verification and Validation of Planning Systems*. AAAI (2005)
19. Dressler, F.: *Self-Organization in Sensor and Actor Networks*. Wiley (2008)
20. Dutertre, B., de Moura, L.: The YICES SMT solver (August 2006), tool paper <http://yices.csl.sri.com/tool-paper.pdf>
21. Dvorak, D., Rasmussen, R., Reeves, G., Sacks, A.: Software architecture themes in JPL’s Mission Data System. In: *IEEE Aerospace Conf. USA* (2000)
22. Adams, E.W.: *A primer of probability logic*. CSLI Publications (1998)
23. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker and its implementation. In: Ball, T., Rajamani, S.K. (eds.) *SPIN 2003*. LNCS, vol. 2648, pp. 230–234. Springer, Heidelberg (2003)
24. Fagin, R., Halpern, J.Y., Megiddo, N.: A logic for reasoning about probabilities. *Information and Computation* 87, 78–128 (1990)
25. Farrell, S., Cahill, V.: *Delay- and Disruption-Tolerant Networking*. Artech House, Inc., Norwood (2006)
26. Gerkey, B.P., Mailler, R., Morisset, B.: Commbots: Distributed control of mobile communication relays. In: *Proc. AAAI Workshop on Auction Mechanisms for Robot Coordination (AuctionBots)*, Boston, MA, pp. 51–57 (July 2006)
27. Goldsby, H.J., Cheng, B.H., Zhang, J.: AMOEBA-RT: run-time verification of adaptive software. In: *Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Reports and Revised Selected Papers*, pp. 212–224. Springer, Heidelberg (2008)

28. Goodloe, A., Gunter, C.A., Stehr, M.-O.: Formal prototyping in early stages of protocol design. In: Meadows, C. (ed.) Proc. POPL 2005 Workshop on Issues in the Theory of Security, WITS 2005, pp. 67–80 (2005)
29. Gordon, D., Spears, W., Sokolsky, O., Lee, I.: Distributed spatial control, global monitoring and steering of mobile physical agents. In: Proc. IEEE Int. Conf. Information, Intelligence, and Systems, pp. 681–688 (1999)
30. Gottwald, S.: A Treatise on Many-Valued Logics. Research Studies Press (2001)
31. Susanne, G., Hassen, S.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
32. Guo, Z., Colombi, G., Wang, B., Cui, J.-H., Maggiorini, D., Rossi, G.P.: Adaptive Routing in Underwater Delay/Disruption Tolerant Sensor Networks. In: Fifth IEEE/IFIP Annual Conf. on Wireless On Demand Network Systems and Services, WONS 2008 (2008)
33. Gutierrez-Nolasco, S., Venkatasubramanian, N., Stehr, M.-O., Talcott, C.L.: Towards adaptive secure group communication: Bridging the gap between formal specification and network simulation. In: 12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), December 18–20, pp. 113–120. University of California, Riverside (2006)
34. Kautz, H.: Satplan04: Planning as satisfiability. In: IPC4, ICAPS (2004)
35. Havelund, K.: Using runtime analysis to guide model checking of Java programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 245–264. Springer, Heidelberg (2000)
36. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSat: A new weighted Max-SAT solver. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 41–55. Springer, Heidelberg (2007)
37. <http://xtune.ics.uci.edu>
38. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.* 11(1), 2–16 (2003)
39. Ivancic, W., Eddy, W., Wood, L., Stewart, D., Jackson, C., Northam, J., da Silva Curiel, A.: Delay/disruption-tolerant network testing using a LEO satellite. In: Eighth Annual NASA Earth Science Technology Conf. (2008)
40. Janakiram, V.K., Agrawal, D.P., Mehrotra, R.: A randomized parallel backtracking algorithm. *IEEE Trans. Comput.* 37(12), 1665–1676 (1988)
41. Kautz, H., Selman, B.: Pushing the envelope: Planning, propositional logic, and stochastic search. In: Shrobe, H., Senator, T. (eds.) Proc. Thirteenth National Conf. Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conf., pp. 1194–1201. AAAI Press, Menlo Park (1996)
42. Kim, M., Stehr, M.-O., Kim, J., Ha, S.: An application framework for loosely coupled networked cyber-physical systems. In: Proc. 8th IEEE Intl. Conf. on Embedded and Ubiquitous Computing, EUC 2010 (2010)
43. Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: Combining formal verification with observed system execution behavior to tune system parameters. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 257–273. Springer, Heidelberg (2007)
44. Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: A probabilistic formal analysis approach to cross layer optimization in distributed embedded systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 285–300. Springer, Heidelberg (2007)

45. Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: Constraint refinement for online verifiable cross-layer system adaptation. In: DATE 2008: Proc. Design, Automation and Test in Europe Conference and Exposition (2008)
46. Kim, M., Talcott, C.L., Stehr, M.-O.: A distributed logic for networked cyber-physical systems. To appear in Proc. Intl. Conf. on Fundamentals of Software Engineering (FSEN 2011). LNCS (2011)
47. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for Java programs. *Form. Methods Syst. Des.* 24(2), 129–155 (2004)
48. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. *Int. J. Softw. Tools Technol. Transf.* 6(2), 128–142 (2004)
49. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient max-sat solving. *Artif. Intell.* 172(2-3), 204–233 (2008)
50. Leucker, M.: Checking and enforcing safety: Runtime verification and runtime reflection. *ERCIM News* (75), 35–36 (2008)
51. Leucker, M., Schallhart, C.: A brief account of runtime verification. *Logic and Algebraic Programming* 78(5), 293–303 (2009)
52. Li, X., Shu, W., Li, M., Huang, H., Wu, M.-Y.: DTN routing in vehicular sensor networks. In: Global Telecommunications Conf., IEEE GLOBECOM 2008i, pp. 1–5 (2008)
53. Lowry, M.R., Philpot, A., Pressburger, T., Underwood, I.: A formal approach to domain-oriented software design environments. In: KBSE, pp. 48–57 (1994)
54. Lyu, M.R. (ed.): *Software Fault Tolerance*. John Wiley and Sons, Inc. (1995)
55. Maes, P.: Concepts and experiments in computational reflection. *SIGPLAN Not.* 22(12), 147–155 (1987)
56. Mailler, R.: Using prior knowledge to improve distributed hill climbing. In: IAT 2006: Proc. IEEE/WIC/ACM Int. Conf. Intelligent Agent Technology, pp. 514–521. IEEE Computer Society, Washington, DC (2006)
57. Mailler, R., Lesser, V.: Solving distributed constraint optimization problems using cooperative mediation. In: AAMAS 2004: Proc. Third Int. Joint Conf. Autonomous Agents and Multiagent Systems, pp. 438–445. IEEE Computer Society, Washington, DC (2004)
58. Martonosi, M.: ZebraNet and beyond: Applications and systems support for mobile, dynamic networks. In: CASES 2008: Proc. 2008 Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems, p. 21. ACM, New York (2008)
59. Maude System, <http://maude.cs1.sri.com>.
60. Modi, P.J., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence* 161, 149–180 (2005)
61. Murphy, A.L., Picco, G.P., Roman, G.-C.: Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.* 15(3), 279–328 (2006)
62. Muscetolla, N., Pandurang, P., Pell, B., Williams, B.: Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2), 5–48 (1998)
63. Networked Cyber-Physical Systems at SRI, <http://ncps.cs1.sri.com>
64. Richardson, M., Domingos, P.: Markov logic networks. *Machine Learning* 62, 107–136 (2006)
65. Rosu, G., Havelund, K.: Monitoring Java programs with Java PathExplorer. In: Proc. Runtime Verification (RV), pp. 97–114. Elsevier (2001)

66. Rueß, H., Shankar, N.: *Introducing Cyberlogic* (2003)
67. Rushby, J.: The design and verification of secure systems. In: *Eighth ACM Symposium on Operating System Principles (SOSP)*, Asilomar, CA, pp. 12–21 (December 1981); *ACM Operating Systems Review* 15(5)
68. Rushby, J.: *Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Contractor Report CR-1999-209347, NASA Langley Research Center (June 1999), also to be issued by the FAA as DOT/FAA/AR-99/58 <http://www.tc.faa.gov/its/worldpac/techrpt/ar99-58.pdf>
69. Rushby, J.: Just-in-time certification. In: *12th IEEE Int. Conf. Engineering of Complex Computer Systems (ICECCS)*, Auckland, New Zealand, pp. 15–24. IEEE Computer Society (2007), <http://www.csl.sri.com/~rushby/abstracts/iceccs07>
70. Rushby, J.: Software verification and system assurance (invited paper). *SEFM* (2009)
71. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D. (eds.) *CAV 1999*. LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999)
72. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: *26th Int. Conf. Software Engineering (ICSE 2004)*, pp. 418–427 (2004)
73. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) *CAV 2004*. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
74. Stehr, M.-O., Kim, M., Talcott, C.: Toward distributed declarative control of networked cyber-physical systems. In: Yu, Z., Liscano, R., Chen, G., Zhang, D., Zhou, X. (eds.) *UIC 2010*. LNCS, vol. 6406, pp. 397–413. Springer, Heidelberg (2010)
75. Stehr, M.-O., Talcott, C.: Planning and learning algorithms for routing in disruption-tolerant networks. In: *Proc. IEEE Military Communications Conference, MILCOM 2008* (2008)
76. Sutton, R.S., Barto, A.G.: *Reinforcement Learning: An introduction*. MIT Press (1998)
77. Toorian, S., Diaz, K., Lee, S.: The CubeSet approach to space access. In: *IEEE Aerospace Conf.* (2008)
78. Torres-Pomales, W.: *Software Fault Tolerance: A Tutorial*. Technical report, NASA (October 2000)
79. Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N., Braynard, R.: Networking named content. In: *Fifth ACM Int. Conf. Emerging Networking EXperiments and Technologies, CoNEXT 2009* (2009)
80. VeStA Tool, <http://osl.cs.uiuc.edu/~ksen/vesta2>
81. Wang, G., Zhou, H.: Quantitative logic. *Inf. Sci.* 179(3), 226–247 (2009)
82. Williams, B.C., Ingham, M., Chung, S.H., Elliott, P.H.: Model-based programming of intelligent embedded systems and robotic space explorers. *Proc. IEEE* 91(3), 212–237 (2003)
83. Williams, B.C., Pandurang Nayak, P.: A model-based approach to reactive self-configuring systems. In: *Proc. AAAI 1996*, pp. 971–978 (1996)
84. XG Reasoner, <http://www.springerlink.com/content/25021851k303tlu0>
85. xTune Framework, <http://xtune.ics.uci.edu>

86. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Runtime Model Checking of Multithreaded C/C++ Programs. Technical report, University of Utah (March 2007)
87. Yokoo, M.: Distributed constraint satisfaction: Foundations of cooperation in multi-agent systems. Springer, London (2001)
88. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.* 204(9), 1368–1409 (2006)
89. Zhang, W., Wang, G., Xing, Z., Wittenburg, L.: Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artif. Intell.* 161(1-2), 55–87 (2005)

Model Feasible Interactions in Distributed Real-Time Systems*

Shangping Ren, Yue Yu**, and Miao Song

Computer Science Department
Illinois Institute of Technology
Chicago, IL 60616
{ren,yyu8,msong8}@iit.edu

Abstract. When a distributed system contains only causal relations from input events to output events, an interaction diagram (*id*) provides a convenient mechanism to study observable behaviors of the system as all events can be mapped to a set of global times that preserve the initial causal relations. However, the interaction diagram focuses only on causal orders among distributed events, which is not sufficient for most real-time applications. Furthermore, in real-time context, a feasible interaction is the one that satisfies not only causal constraints and precedence constraints, but also real-time constraints. However, feasibility checking for a given set of real-time constraints is asymptotically harder than for causal or precedence constraints. In this paper, we first extend the interaction diagram with precedence constraints and develop a mechanism that allows order preserving composition of the extended interaction diagram (*eid*) with timing constraint graph (*tcg*). The composition of the extended interaction diagram and timing constraining graph is called timed interaction diagram (*tid*). To reduce the time complexity differences between the two different feasibility checkings, event bundling is introduced to partition timed interaction diagrams. We show that a lattice of bundled interaction diagrams (*bid*) can be derived from a given timed interaction diagram to improve the efficiency of feasibility checking for arbitrary real-time constraints.

1 Introduction

In distributed real-time applications, such as multidimensional battlefield control systems and airport tracking systems, distributed entities usually expose a high degree of concurrency and autonomy. Basing the semantics of distributed real-time systems on conventional notions of states or state transitions introduces unnecessary nondeterminism, impedes the understanding of the systems. Such models in general are problematic and fail to be compositional [10].

* This research is supported in part by NSF under grants CNS 0746643, CNS 1018731, and CNS 1035894

** The work is done when he was a ph.d student at IIT. Dr. Yue Yu is currently working at the China Bond Insurance Co., Ltd

As an example, consider the actor model [1]. In the actor model, actors are concurrent and autonomous objects. They interact with each other through asynchronous messages. Individual actors are sequential, since each actor has a single thread of execution. Unprocessed messages are buffered in the receiver actor's mailbox. The actor model provides a simple abstraction for open distributed systems and fosters the construction of highly concurrent applications.

Conventionally, the behavior of an actor system is defined by system state transitions [1, 3, 2], where the state is expressed via a high ordered structure called a *configuration*, which is an instantaneous global snapshot of individual actor states (α) and pending message sets (μ), denoted as $\langle\langle\alpha \mid \mu\rangle\rangle$. However, this notion of system states is not well defined in a real-time environment for the following reasons.

- A global snapshot of the internal states of asynchronous entities is difficult to achieve in real-time settings. It highly depends upon the precisions of individual clocks.
- Messages sent from actors may be in transit for some time before they are ready to be chosen as the next transition for execution. However, the μ definition does not differentiate the subtle but crucial state difference among the yet to be processed messages.

In contrast, interface-based and interaction-based semantic models, such as event diagrams [6], abstract behavior types [5], and interaction semantics [15, 11, 4], avoid unnecessary assumptions about the internal states of distributed entities and the states of unprocessed messages. The semantic foundations of these models are based on events, inputs/outputs, interactions, and other externally observable sorts.

For instance, Talcott et. al shows in [15, 16, 4] how interaction paths in actor-based systems can be defined as an observable projection of individual computation paths. While a computation path is based on labeled transitions of actor system configurations, the internal state-based transitions are removed. As a result, the observable behaviors of the systems are used as the only criterion for defining the meaning of the systems.

In concurrent systems, the observables, such as the interactions, may occur concurrently. The interaction path is then in fact a linearization of the observables. The same set of observables may have different sets of linearizations. Hence the notion of equivalence is introduced to study the relations of different sets of linearizations.

In [4], three different types of equivalences are defined based on the observables of an actor system, namely, *must*, *may* and *convex* equivalences. Note that for purposes of observation a closed system is assumed. To define equivalences, observations are made in the computation paths of all possible closing contexts. In particular, two actor systems are *must* equivalent if the nonoccurrence of an observable event in one system's computation paths implies the nonoccurrence of the observable event in the other system's computation path; and vice versa. The *may* equivalence states that if an observable event occurs in some of one

system's computation paths, it must also occur in some of the other system's computation paths; and vice versa. Two actor systems are *convex* equivalent if they are both *must* and *may* equivalent. The *must* and *convex* equivalences are proven to be identical when fairness is assumed, i.e, messages sent to an actor will eventually be processed by the receiving actor. These equivalence relations are rather weak and may not be sufficient under real-time settings, since they require only that the observables occur either in both or in neither computation paths. The occurrence order and timing relations of the observables are irrelevant in the equivalence definitions.

J.Jiang and J.Wu pointed out in [8] that causally independent (neither causally related nor causally conflicted) observables form an equivalence class. Observables in an equivalence class can be bundled as a single event. Furthermore, all causally independent relation sets form a partition of the observables set. When causally independent observables are bundled as a single event, the set of observables becomes totally ordered in the logical time domain.

However, in real-time applications, coordination constraints such as real-time constraints may further restrict the allowed interaction paths. Two equivalent interactions may become inequivalent when we take into account the real-time aspects of the two interactions. Therefore, for real-time systems, we need stronger equivalence classes to categorize equivalent behaviors.

A constraint graph is commonly used as a tool to study real-time constraints. For a given set of timing constraints, we can construct a constraint graph by denoting each constraint as a directed weighted edge connecting the two constrained events (nodes). Furthermore, any implicit constraints that are implied by the given constraint set are derivable from the graph. When a constraint graph is augmented with all implicit constraint edges, we obtain a precedence-preserving graph [13]. Two real-time computations are equivalent if they have equivalent interaction paths that also preserve the precedence order defined in the precedence-preserving graph.

It is well known that the validity of a computation graph can be decided by testing the existence of a causality loop in the graph, while the feasibility of a set of timing constraints can be determined by checking if a negative cycle exists in the deadline/delay graph [13, 9, 12, 17]. However, since testing for negative weight cycles in a directed weighted graph is asymptotically harder than testing for cycles in a directed unweighted graph, feasibility checkings for real-time constraints are harder than for causal constraints. However, if we utilize the fact that timing constraints are imposed on its corresponding computation graph, the efficiency of feasibility checking can be improved.

The rest of the paper is organized as follows. Section 2 gives basic definitions of the interaction semantics that will be used throughout the paper. Section 3 introduces *extended interaction diagrams* and discusses the feasibility issues of given extended interaction diagram. In section 4, we introduce *timed interaction diagram* and discuss interaction semantics under real-time constraints. Section 5 introduces the lattice of *bundled interaction diagrams* derived from a timed interaction diagram. We show that interaction equivalence is preserved under bundled

interaction diagram before and after timing constraints are imposed. More importantly, we show that for a real-time application, timing constraint feasibility checking can be done locally within each bundle of a bundled interaction diagram, eliminating the global checking of a timed interaction diagram, and hence improving the efficiency of feasibility checking. Finally, Section 6 concludes the paper and discusses our future work.

2 Interaction Diagram

To facilitate our discussion, we assume the actor model [1] is used to model distributed real-time systems. For completeness, we first re-state the key concepts about interaction diagrams defined in [16] (Definition 1 through Definition 6). We then discuss partial orders in individual interaction diagrams and their compositions.

2.1 Definitions

Definition 1. *Interaction diagrams* (id) are structures of the form

$$id = (\rho, \chi)(I, O, \prec) \quad (1)$$

where (ρ, χ) ¹ is an interface consisting of a set of receptionists, ρ , which specifies the actors within the system that are visible to the environment, and a set of externals, χ , which specifies the names of actors in the environment known to actors of the system. I is a set of input events and O is a set of output events. The sets I and O are disjoint and $\prec \subseteq I \times O$ is the visible combined ordering restricted to input and output events. \square

To make the definition more concrete, consider a matrix multiplication example.

Example 1. A matrix multiplier (with address m) is an actor that multiplies two matrices $M_{A \times B}$ and $M_{B \times C}$ sent from a customer (with reply address c) in two separate messages. It then replies the customer with the result $M_{A \times C}$. The interaction diagram representation of the matrix multiplier is given below:

$$id = (\{m\}, \{c\})(I, O, \prec)$$

where

$$I = \{e_1, e_2\}$$

$$O = \{e_3\}$$

$$\prec = \{(e_1, e_3), (e_2, e_3)\}$$

¹ The actor interface does not play an important role in our discussions. However, to be consistent with the original interaction diagram theories and to facility our future work, we keep the interface part in all the following discussions.

where the packet function for the multiplier events is:

$$\begin{aligned} pkt(e_1) &= m \triangleleft M_{A \times B} \\ pkt(e_2) &= m \triangleleft M_{B \times C} \\ pkt(e_3) &= c \triangleleft M_{A \times C} \end{aligned}$$

The graphic representation of interaction diagram for the matrix multiplier is illustrated in Fig. 1. □

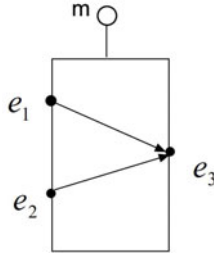


Fig. 1. Interaction Diagram for Matrix Multiplier

Given that the set of input and output event sets I and O , a global time map, $g : I \cup O \xrightarrow{inj} N$, is an order preserving injective function from the event sets to the set of natural numbers.

Definition 2. Global Time for Interaction Diagram: given an interaction diagram $id = (\rho, \chi)(I, O, \prec)$, the set of global times for the id is defined as following:

$$GT(id) = \{g : I \cup O \xrightarrow{inj} N \mid \forall e_1, e_2 \in I \cup O, e_1 \prec e_2 \Rightarrow g(e_1) < g(e_2)\} \quad (2)$$

□

Interaction paths are event paths in which the domain of the global time map contains no internal events. Interaction paths model possible global times and abstract completed computations.

Definition 3. Interaction Path: given an interaction diagram $(\rho, \chi)(I, O, \prec)$, the interaction paths are structures of the form

$$(\rho, \chi)g \quad (3)$$

where g is a global time mapping for the interaction diagram, and $Dom(g) \subseteq (I \cup O)$. □

Definition 4. Interaction Equivalence Systems: two actor systems are said to be interaction equivalent if they have the same set of interaction paths. \square

In [16], two interaction diagrams can be composed by linking and hiding synchronization points between two diagrams. Intuitively, the synchronization points of $((\rho_0, \chi_0)g_0, (\rho_1, \chi_1)g_1)$ are the time points when one path outputs a message whose target is a receptionist of the other path. Synchronization points are defined as

$$\begin{aligned} \text{sync}((\rho_0, \chi_0)g_0, (\rho_1, \chi_1)g_1) = \\ \{i \mid \exists j < 2, e \in I \cup O, \text{s.t. } g_j^{-1}(i) = \text{out}(e) \wedge g_{1-j}^{-1}(i) = \text{in}(e)\} \end{aligned} \quad (4)$$

The composition of interaction paths and interaction diagrams is defined as follows.

Definition 5. Composition of Interaction Paths: the composition of interaction paths $(\rho_0, \chi_0)g_0$ and $(\rho_1, \chi_1)g_1$ is a structure of the form $(\rho, \chi)g$ of which:

$$(\rho, \chi)g = (\rho_0, \chi_0)g_0 \circ (\rho_1, \chi_1)g_1 \quad (5)$$

where

$$\rho = \rho_0 \cup \rho_1,$$

$$\chi = \chi_0 \cup \chi_1 - \rho,$$

$$\text{Dom}(g) = \text{Dom}(g_0) \cup \text{Dom}(g_1) - \bigcup_{i \in \text{sync}((\rho_0, \chi_0)g_0, (\rho_1, \chi_1)g_1)} (g_0^{-1}(i) \cup g_1^{-1}(i)),$$

$$g(e) = g_j(e) \text{ if } e \in \text{Dom}(g_j) \quad j = 0, 1 \quad \square$$

Definition 6. Composition of Interaction Diagrams: the parallel composition of interaction diagrams $((\rho_0, \chi_0)(I_0, O_0, \prec_0), g_0)$ and $((\rho_1, \chi_1)(I_1, O_1, \prec_1), g_1)$ is a structure of the form $((\rho, \chi)(I, O, \prec), g)$ of which

$$\begin{aligned} ((\rho, \chi)(I, O, \prec), g) = \\ ((\rho_0, \chi_0)(I_0, O_0, \prec_0), g_0) \circ ((\rho_1, \chi_1)(I_1, O_1, \prec_1), g_1) \end{aligned} \quad (6)$$

where

$$(\rho, \chi)g = (\rho_0, \chi_0)g_0 \circ (\rho_1, \chi_1)g_1$$

$$I = \text{Dom}(g) \cap (I_0 \cup I_1)$$

$$O = \text{Dom}(g) \cap (O_0 \cup O_1)$$

$$\prec = (\prec_0 \cup \prec_1) \downarrow (I \times O) \cup \{(in(e), out(e')) \mid \exists e'', j < 2,$$

$$in(e) \prec_j out(e'') \wedge in(e'') \prec_{1-j} out(e') \wedge g_j(out(e'')) = g_{1-j}(in(e''))\}$$

and $(\prec_0 \cup \prec_1) \downarrow (I \times O)$ restricts $(\prec_0 \cup \prec_1)$ to $I \times O$. \square

2.2 Validity of Interaction Diagram

An interaction path of an interaction diagram is defined as a global time mapping g together with the interface (ρ, χ) of the interaction diagram. Note that the sufficient condition for an interaction diagram to have a global time mapping g is that the interaction diagram is a DAG since the existence of a causality loop C indicates that $\forall e \in C, e \prec e$. In other words, events on a cycle cannot be mapped to natural numbers that preserves the \prec relation.

Definition 7. A *valid interaction diagram* is the one that contains no causality loop.

The following theorems state that interaction diagrams given by Definition 1 or Definition 6 are valid.

Theorem 1. Let id be an interaction diagram that satisfies Definition 1. The id is valid and thus a non-empty set of global time mappings exist for the diagram. \square

Proof. Suppose to the contrary that a causality loop C exists, where $C = e_1 \prec e_2 \prec \dots \prec e_n \prec e_1$. Since $\prec \subseteq I \times O$, it must be that

$$\begin{aligned} e_1 &\in I, e_2 \in O \\ e_2 &\in I, e_3 \in O \\ &\vdots \\ e_n &\in I, e_1 \in O \end{aligned}$$

which contradict that I and O are disjoint. Therefore, id is a DAG, and a topological sort algorithm can give a global time mapping for events in the id .

Theorem 2. The composition of two interaction diagrams is valid if and only if the two composing interaction diagrams are valid. \square

Proof. The composition of two interaction diagrams is accomplished by hiding events at synchronization points (so that these events become internal) and keeping the remaining input and output events (which remain external). In the composed interaction diagram, causal relations are defined as $\prec \subseteq I \times O$. As shown in Theorem 1, no causal loop exists among the external events. Since the causal relation defined between events at synchronization points can be explained as $\prec' \subseteq O \times I$, it may be possible to introduce a causality loop that contains internal events. Therefore, we only need to prove that such possibility does not exist.

Suppose to the contrary that, the composition of two causality loop free interaction diagrams $(\rho_0, \chi_0)(I_0, O_0, \prec_0)$ and $(\rho_1, \chi_1)(I_1, O_1, \prec_1)$ produce a causality loop $C = out(e) \diamond in(e) \prec_1 \dots \prec_1 out(e') \diamond in(e') \prec_0 \dots \prec_0 out(e)$, where $out(e) \in O_0, in(e) \in I_1, out(e') \in O_1, in(e') \in I_1$, and “ \diamond ” denotes synchronization points as illustrated in Fig. 2.

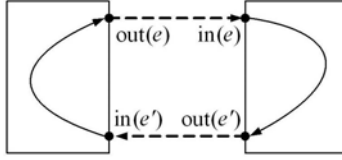


Fig. 2. Composition of Interaction Diagram

By the definition of the causal relations “ \prec_1 ” and “ \prec_2 ” in the two individual components, we have

$$\left\{ \begin{array}{l} \underbrace{in(e) \prec_1 \cdots \prec_1 out(e')}_{0 \text{ or more}} \Rightarrow g_1(in(e)) \leq g_1(out(e')) \\ \underbrace{in(e') \prec_0 \cdots \prec_0 out(e)}_{0 \text{ or more}} \Rightarrow g_0(in(e')) \leq g_0(out(e)) \end{array} \right. \quad (7)$$

By the definition of synchronization points between the two components, we have

$$\left\{ \begin{array}{l} g_0(out(e)), g_1(in(e)) \in \text{synch}(\rho_0, \chi_0)g_0, (\rho_1, \chi_1)g_1 \\ \Rightarrow g_0(out(e)) = g_1(in(e)) \\ g_1(out(e')), g_0(in(e')) \in \text{synch}(\rho_0, \chi_0)g_0, (\rho_1, \chi_1)g_1 \\ \Rightarrow g_1(out(e')) = g_0(in(e')) \end{array} \right. \quad (8)$$

From (7) and (8), we have

$$g_0(out(e)) = g_1(in(e)) = g_1(out(e')) = g_0(in(e')) \quad (9)$$

Note the following property of \prec

$$(\forall e_1, e_2, e_1 \underbrace{\prec \cdots \prec}_{0 \text{ or more}} e_2) \Rightarrow g(e_1) = g(e_2) \text{ iff } e_1 = e_2 \quad (10)$$

From (9) and (10), we have

$$in(e) = out(e') \wedge in(e') = out(e)$$

which contradicts that I and O are disjoint.

From Theorem 1 and Theorem 2 and the induction proof, it is clear that interaction diagrams composed of individual components complying with Definition 1 are causality loop free and thus global time mappings exist for incrementally defined interaction diagrams.

3 Extended Interaction Diagram

Interaction diagrams abstract the actor model of computation by hiding internal events of event diagrams, and keeping the induced causal orders among events that occur at the interfaces of the actor system. Our observation is that the current interaction diagram theory may not be sufficient to model distributed real-time systems for the following reasons:

1. According to the Definition 1, $\prec \subseteq I \times O$ is the set of causal orders from input events to output events. However, there are cases where precedence constraints $\prec' \subseteq (I \cup O) \times (I \cup O)$ are necessary. For instance, a computational unit could require its input events to come in a certain order and thus $\prec' \subseteq I \times I$ are inevitable.
2. Since the basic interaction diagrams consider only logical times (causal orders), the timing properties that are important for distributed real-time systems are not directly supported. For instance, in a distributed environment, in order for a decision unit to take critical actions, the data from two different sources must arrive at the requester within a limited time frame. In this case, we have to timely constraint the two input events.

To overcome the issues discussed above, we extend the interaction diagram by broadening $\prec \subseteq I \times O$ to $\prec \subseteq (I \cup O) \times (I \cup O)$.

3.1 Extended Interaction Diagram

In order to model individual components with precedence relations among their events, we extend the interaction diagram(*id*) to allow precedence constraints among any pairs of events. The formal definition of extended interaction diagram is given below:

Definition 8. *Extended interaction diagrams (eid) are structures of the form*

$$eid = (\rho, \chi)(I, O, \prec) \tag{11}$$

where ρ, χ, I , and O are the same as given in Definition 1. $\prec \subseteq (I \cup O) \times (I \cup O)$ is the set of partial orders between event pairs. \square

As given in the definition, the extended interaction diagram only concerns the orderings between event pairs and boundaries of input events and output events are diminished from the diagram. The extended interaction diagram for Example 1 given in Section 2.1 shown in Fig. 3.

In Section 2.2, we showed that an interaction diagram defined by Definition 1 does not have causality loops and thus a non-empty set of global time mappings always exists for the diagram. However, when precedence constraints are added, the existence of a global time mapping is not guaranteed and thus validity checking must be conducted.

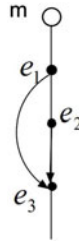


Fig. 3. Extended Interaction Diagram for Matrix Multiplier

3.2 Validity of Extended Interaction Diagram

Although the extended interaction diagram, we are able to model individual components with precedence relations among their events, this flexibility comes at the cost of additional complexity. With our extension, it is not hard to see that Theorem 1 becomes invalid in the context of *eid*. By allowing $\prec \subseteq (I \cup O) \times (I \cup O)$, we do not distinguish input and output events. Therefore, it is possible that a component modeled by an extended interaction diagram contains precedence loop and thus does not have a global time mapping.

Besides, Theorem 2 no longer holds for extended interaction diagrams since precedence orders between arbitrary pairs of events are allowed in the extended interaction diagrams. Therefore, even if two extended interaction diagrams are valid (precedence loop free), it is possible that their compositions may contain precedence loops and thus does not have a global time mapping. Fig. 4 shows an example.

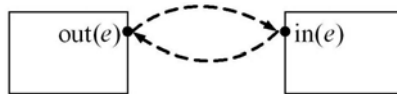


Fig. 4. Composition of Extended Interaction Diagrams

Therefore, for a given extended interaction diagram, we must conduct validity checking. A valid or feasible interaction is the one without precedence loop. A depth first search algorithm can be applied on the extended interaction diagram to check for loops, and it runs in optimized time of $O(|I \cup O| + |\prec|)$.

4 Timed Interaction Diagram

In this section, we study the real-time extension of the extended interaction diagram, i.e., timed interaction diagram, and its interaction semantics.

4.1 Composing Extended Interaction Diagram with Timing Constraint Graph

Definition 9. A *timing constraint* among events is a function $\delta : E \times E \rightarrow I_{\mathbb{R}}$ where E is a set of events and $I_{\mathbb{R}}$ denotes the set of continuous subsets (i.e., intervals) of the set of real numbers. A timing constraint is of the form

$$\delta(e_i, e_j) = [t_{min}, t_{max}] \Leftrightarrow e_i + t_{min} \leq e_j \leq e_i + t_{max} \quad (12)$$

Two events are unconstrained if and only if $\delta(e_i, e_j) = (-\infty, +\infty)$. □

Definition 10. A *timing constraint graph* (tcg) is a directed weighted graph (E, δ) which satisfies the following conditions:

- The vertex set of tcg is E .
- For each pair of events e_i, e_j ($i \neq j$), the edge between e_i and e_j and its weight is defined as

$$\text{if } \neg(t_{min} = -\infty \wedge t_{max} = +\infty) \delta(e_i, e_j) = [t_{min}, t_{max}] \Leftrightarrow e_i \xrightarrow{[t_{min}, t_{max}]} e_j \quad (13)$$

and there is no edge between e_i, e_j if $\delta(e_i, e_j) = (-\infty, +\infty)$, i.e., e_i and e_j are unconstrained.

- The co-domain of δ satisfies (14) is:

$$\forall \delta(e_i, e_j) = [t_{min}, t_{max}] : t_{min} \in \mathbb{R}^+ \cup \{-\infty\} \wedge t_{max} \in \mathbb{R}^+ \cup \{+\infty\} \quad (14)$$

□

Note that the restriction of δ in the definition does not restrict the expressiveness power of linear timing constraints. Given a timing constraint $\delta'(e_i, e_j)$ with arbitrary intervals, it can be transformed to comply with the restriction in the following way:

$$\delta'(e_i, e_j) = [t_{min}, t_{max}] \Leftrightarrow \begin{cases} \delta(e_j, e_i) = [-t_{max}, -t_{min}] \Leftrightarrow \\ e_i \xleftarrow{[-t_{max}, -t_{min}]} e_j \text{ if } t_{min} \leq 0 \wedge t_{max} \leq 0 \\ \delta(e_i, e_j) = [0, t_{max}] \wedge \delta(e_j, e_i) = [0, -t_{min}] \Leftrightarrow \\ e_i \xrightarrow{[0, t_{max}]} e_j \wedge e_i \xleftarrow{[0, -t_{min}]} e_j \text{ if } t_{min} \leq 0 \wedge t_{max} > 0 \end{cases} \quad (15)$$

Therefore, in the following, we restrict our discussions to timing constraints of the form

$$\delta \in E \times E \rightarrow \{[t_{min}, t_{max}] \mid t_{min} \in \mathbb{R}^+ \cup \{-\infty\} \wedge t_{max} \in \mathbb{R}^+ \cup \{+\infty\}\}$$

It is easy to see that precedence relations in extended interaction diagram $\prec \subseteq (I \cup O) \times (I \cup O)$ can be generalized as a special case of timing constraint $\delta' : (I \cup O) \times (I \cup O) \rightarrow \{[\epsilon, +\infty] | \epsilon \rightarrow 0^+\} \cup \{(-\infty, +\infty)\}^2$:

$$e_i \prec e_j \Leftrightarrow e_i + \epsilon \leq e_j (\epsilon \rightarrow 0^+) \Leftrightarrow e_i \xrightarrow{[\epsilon, +\infty]} e_j$$

This provides a basis for defining the composition of extended interaction diagram and timing constraint graph. The composition of the two event structures, the timed interaction diagram, has the set of all events as its vertices and the edges from both event structures. In the case where multiple edges exist from one vertex to another in the composition, we merge the edges and take the intersection of the ranges defined on these edges. The formal definition is given below:

Definition 11. A *timed interaction diagram* (*tid*) is a composition of an extended interaction graph $(\rho, \chi)(I, O, \prec)$ and a feasible timing constraint graph $(I \cup O, \delta)$. It has the structure of the form

$$tid = (\rho, \chi)(I, O, \Delta) \quad (16)$$

where ρ, χ, I , and O are defined the same as in Definition 1. $\prec : (I \cup O) \times (I \cup O) \rightarrow \{[\epsilon, +\infty] | \epsilon \rightarrow 0^+\} \cup \{(-\infty, +\infty)\}$ is the partial orders between visible event pairs. $\delta : (I \cup O) \times (I \cup O) \rightarrow \{[t_{min}, t_{max}] | t_{min} \in \mathbb{R}^+ \cup \{-\infty\} \wedge t_{max} \in \mathbb{R}^+ \cup \{+\infty\}\}$, is the timing constraint function restricted to input and output events. The composed order function $\Delta : (I \cup O) \times (I \cup O) \rightarrow \{[t_{min}, t_{max}] | t_{min} \in \mathbb{R}^+ \cup \{-\infty\} \wedge t_{max} \in \mathbb{R}^+ \cup \{+\infty\}\}$, is defined as:

$$\begin{aligned} Dom(\Delta) &= (I \cup O) \times (I \cup O) \\ \Delta(e_i, e_j) &= \prec(e_i, e_j) \cap \delta(e_i, e_j) \end{aligned} \quad (17)$$

□

For graphical representation purposes, we use different arrows to differentiate the relationships in extended integration diagram (*eid*), time constraint graph (*tcg*), and timed integration diagram (*tid*). In particular, we use \rightarrow to represent \prec in *eid*, \rightarrow for δ in *tcg*, and \rightarrow for Δ in *tid*, respectively.

The following theorem states that the timed interaction diagram neither relax nor tighten the original timing constraints on the set of input and output events.

Theorem 3. Given an extended interaction diagram $(\rho, \chi)(I, O, \prec)$ and a timing constraint graph $(I \cup O, \delta)$, the composed diagram, i.e., timed interaction diagram, $(\rho, \chi)(I, O, \Delta)$, preserves \prec and δ . □

Proof. Based on Definition 11, there are four cases as shown in Fig. 5 between any two pair of events e_i and e_j when composing an extended interaction diagram and a timing constraint graph:

² ϵ is a small real number larger than 0 as the global time mapping must map two causally related events to different time points.

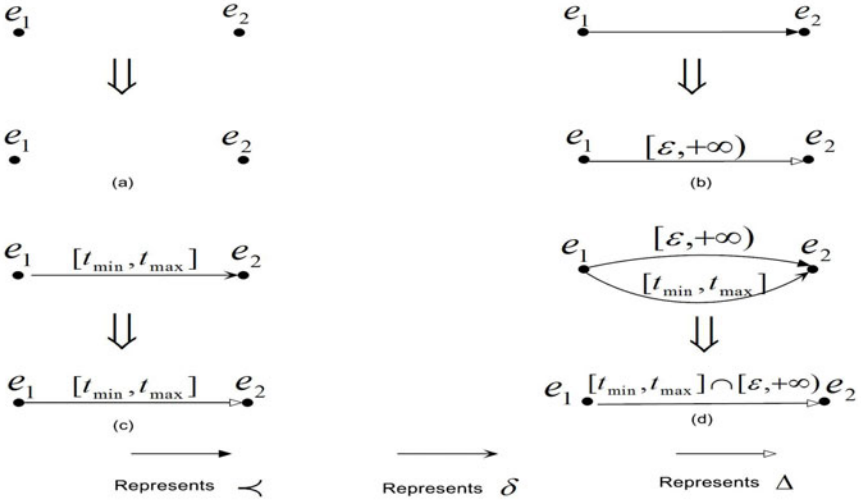


Fig. 5. Constructing Timed Interaction Diagram

Case 1: $\prec(e_i, e_j) = (-\infty, +\infty) \wedge \delta(e_i, e_j) = (-\infty, +\infty)$ i.e., when there is no precedence, nor timing constraints between e_1 and e_2 , then in the timed interaction graph, $\Delta(e_i, e_j) = (-\infty, +\infty)$, i.e., the two events are unconstrained, and thus there will be no edge between them (Fig. 5(a)).

Case 2: $\prec(e_i, e_j) = [\epsilon, +\infty) \wedge \delta(e_i, e_j) = (-\infty, +\infty)$ i.e., when there is a precedence constraint between e_1 and e_2 , but no timing constraints, then in the timed interaction graph, $\Delta(e_i, e_j) = [\epsilon, +\infty)$ which maintains the same precedence constraint direction (Fig. 5(b)).

Case 3: $\prec(e_i, e_j) = (-\infty, +\infty) \wedge \delta(e_i, e_j) = [t_{min}, t_{max}]$ where $\neg(t_{min} = -\infty \wedge t_{max} = +\infty)$, i.e., when there is a timing constraint, but no precedence constraints between e_1 and e_2 , then in the timed interaction graph, $\Delta(e_i, e_j) = [t_{min}, t_{max}]$ and there will be an edge which maintains both the constraint direction and the constraint weight (Fig. 5(c)).

Case 4: $\prec(e_i, e_j) = [\epsilon, +\infty) \wedge \delta(e_i, e_j) = [t_{min}, t_{max}]$ where $\neg(t_{min} = -\infty \wedge t_{max} = +\infty)$, i.e., when there is both precedence and timing constraints between e_1 and e_2 , there will be an edge with weight equals to $[\epsilon, +\infty) \cap [t_{min}, t_{max}]$ (Fig. 5(d)). Note that:

$$\begin{aligned}
 \Delta(e_i, e_j) &= \prec(e_i, e_j) \cap \delta(e_i, e_j) \\
 &\Leftrightarrow e_j - e_i \in [\max\{\epsilon, t_{min}\}, \min\{t_{max}, +\infty\}] \\
 &\Leftrightarrow e_j - e_i \in [\epsilon, +\infty) \wedge e_j - e_i \in [t_{min}, t_{max}] \\
 &\Leftrightarrow \prec(e_i, e_j) = [\epsilon, +\infty) \wedge \delta(e_i, e_j) = [t_{min}, t_{max}]
 \end{aligned} \tag{18}$$

Therefore, the composition preserves constraint information.

4.2 Validity of Timed Interaction Diagram

As shown in various literatures [13, 9, 12, 17], timing constraints in distributed real-time systems are either deadline or delay constraints, where

- A deadline constraint is of the form

$$e_j - e_i \leq d_k (d_k \geq 0)$$

- A delay constraint is of the form

$$e_j - e_i \leq -d_k (d_k \geq 0)$$

A directed weighted constraint graph (we call it deadline/delay graph to distinguish it from *tcg* defined in the previous subsection) can be defined from the two types of constraints in a similar way as the previous subsection. The theory about constraint satisfaction feasibility is well-established for deadline/delay graph thus defined. It states that a negative cycle in the graph means that the set of timing constraints is not feasible.

In fact, the timing constraint graph (and timed interaction diagram) and the deadline/delay graph have the same expressiveness and are mutually transformable. This is shown in the Fig. 6.

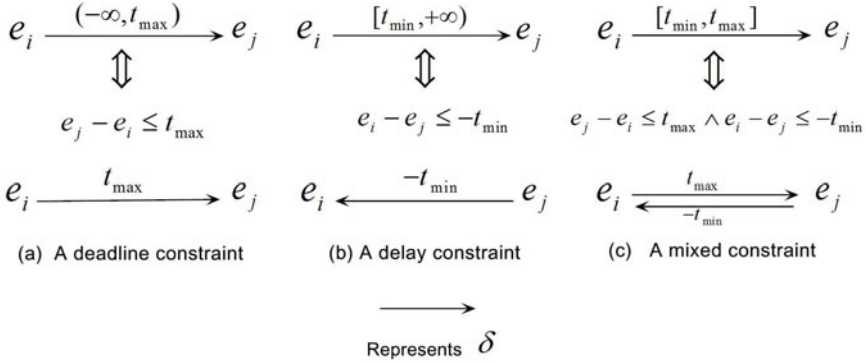


Fig. 6. Deadline/Delay Constraints

Therefore, feasibility checking of the timed interaction diagram can be done by transforming the *tid* to a deadline/delay graph and checking for negative weight cycles in the graph. Bellman-Ford algorithm can be used to implement the feasibility checking which runs in time $\mathcal{O}(|I \cup O| \cdot |\Delta|)$. Also note that in the Section 3, we pointed out that an extended interaction diagram needs to be a DAG in order to have a non-empty set of global time mappings. This can be viewed as a special case in the context of deadline/delay graph where a

precedence constraint is a special case of delay constraint which is represented as a negative weight edge. Thus a precedence loop in an extended interaction diagram is a negative cycle in the deadline/delay graph which means that the set of timing constraints is not feasible [13].

5 Bundled Interaction Diagrams and Their Properties

In the previous section, we show that an interaction diagram $(\rho, \chi)(I, O, \prec)$ can be composed with a timing constraint graph $(I \times O, \delta)$ to yield a timed interaction diagram $(\rho, \chi)(I, O, \Delta)$. We have also shown that the timed interaction graph preserves both the causal order (\prec) and the timing constraints (δ) defined in both graphs. Let $IPSet(eid)$, $IPSet(tcg)$, and $IPSet(tid)$ denote the sets of interaction paths of an extended interaction diagram eid , a feasible timing constraint graph tcg , and the timed interaction diagram tid composed from id and tcg , respectively. It is easy to see that

$$IPSet(eid) \cap IPSet(tcg) = IPSet(tid) \quad (19)$$

Therefore, some interaction paths that are valid in the interaction diagrams may be invalid in timed interaction diagrams. Moreover, since the interaction diagram contains only partial orders, theories developed for the interaction diagram are based on causality. Therefore, it is important for us to see the invariant of an interaction diagram when timing constraints are imposed, that is, the event structures that are preserved after timing constraints are imposed. As shown in this section, if we bundle events in an interaction diagram into “big events” when forming the timed interaction diagram such that the relations between bundles are only of acyclic causal orders, interaction paths valid for such event structures under the interaction diagram remains valid when timing constraint are imposed.

On the other hand, from an algorithmic perspective, determining the feasibility of an extended interaction diagram can be reduced to the problem of checking for cycles in a directed unweighted graph, and determining the feasibility of a timed interaction diagram can be reduced to the problem of checking for negative weight cycles in a directed weighted graph. However, checking for negative cycle in a directed weighted graph $G(V, E)$ is asymptotically harder than checking for cycles in a directed unweighted graph $G'(V', E')$. To check for negative cycle in a directed weighted graph, Bellman-Ford algorithm could be run on the graph which runs in time $\mathcal{O}(|V| \cdot |E|)$; while to check for cycle in a directed unweighted graph, a depth first search suffices which only runs in time $\mathcal{O}(|V'| + |E'|)$. A careful observation of the event structure of the timed interaction diagram unveils that if we bundle its events into “big events” such that the relations between bundles are only of acyclic partial orders, the time consuming negative-weight-cycle checking can be restricted within bundles so that the complexity of feasibility checking can be dramatically reduced.

5.1 Bundled Interaction Diagram

Before we present the bundled interaction diagram, consider the following examples:

Example 2. Given an extended interaction diagram and a timing constraint graph shown in Fig. 7(a) and 7(b), respectively. In Fig. 7, m, c donates different actors, the solid circles donate input or output events within actors. Composing these two diagrams, we obtain the timed interaction diagram shown in Fig. 7(c). Note that the weight $[\epsilon, +\infty)$ on the edges of the timed interaction diagram which inherit from the causal relations in the original interaction diagram are ignored in Fig. 7(c).

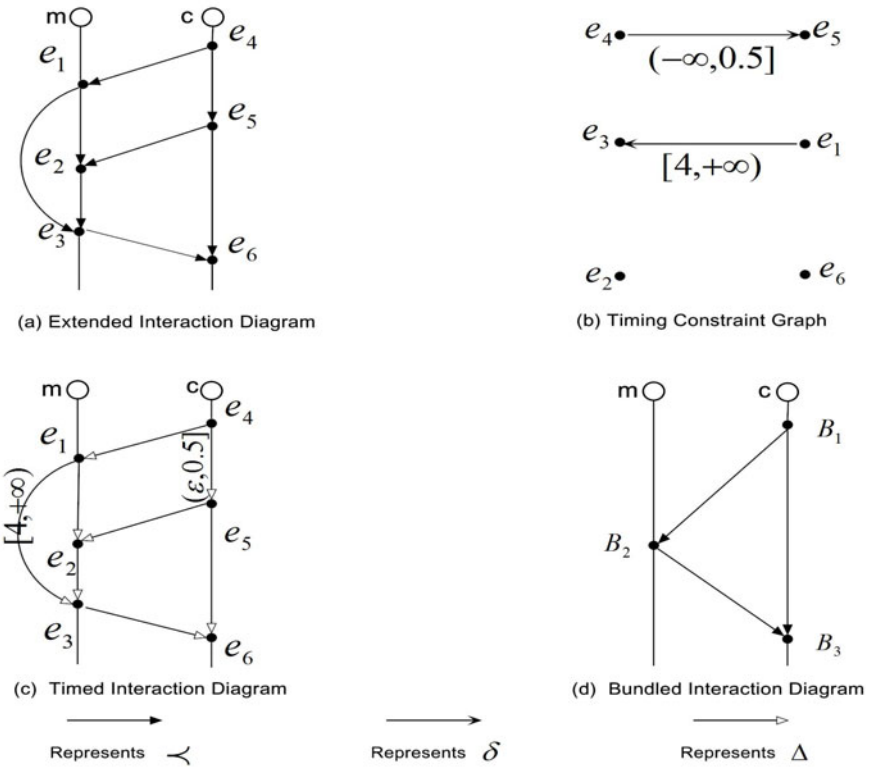


Fig. 7. Bundling Timely Constrained Events

As can be seen, valid interaction paths of (a) may not be valid for (c). For example, a valid global time mapping for the actor system before timing constraints are imposed could be $g(e_4) = 1, g(e_1) = 2, g(e_5) = 3, g(e_2) = 4, g(e_3) = 5, g(e_6) = 6$. However, when timing constraints are added, $g(e_3) = 5$ and $g(e_1) = 2$ violates the delay constraint $e_1 + 4 \leq e_6$ because delay constraints are more

strict than causal order constraints (note that precedence constraint $e_1 \prec e_6$ in Fig. 7(a) is overridden by the delay constraint $e_1 + 4 \leq e_6$ when the timed interaction diagram is constructed). Moreover, $g(e_4) = 1$ and $g(e_5) = 3$ violates the deadline constraint $e_5 - e_1 \leq 0.5$.

Intuitively, bundling timely constrained events in the underlying timing constraint graph of a timed interaction diagram will result in an event structure consisting of only acyclic precedence orders. In this example, the events in the timed interaction diagram in Fig. 7(c) can be partitioned into $\{\{e_4, e_5\}, \{e_1, e_2, e_3\}, \{e_6\}\}$, where $B_1 = \{e_4, e_5\}$, $B_2 = \{e_1, e_2, e_3\}$, $B_3 = \{e_6\}$ as shown in Fig. 7(d). The allowed interaction path for Fig. 7(a) before timing constraints are added is $(\{m, c\}, \phi)[B_1, B_2, B_3]$ and it remains the same when timing constraints are imposed. \square

Fig. 7(d) shows the event structure when events are bundled. We call it bundled interaction diagram which is formally defined below.

Definition 12. Bundled Interaction Diagram: for a timed interaction diagram $tid = (\rho, \chi)(I, O, \Delta)$ composed from an extended interaction diagram $(\rho, \chi)(I, O, \prec)$ and a feasible timing constraint graph $(I \cup O, \delta)$, the bundled interaction diagram of bid is defined as

$$bid = (\mathbb{B}, \prec\prec) \quad (20)$$

where $\mathbb{B} = \{B_i | i = 1, \dots, n\}$ is a set of n bundles. A bundle B_i of \mathbb{B} is a pair (E_i, Δ_i) where

- $E_i \subseteq I \cup O \wedge \{E_i | i = 1, \dots, n\}$ is a partition of $I \cup O$ such that $\forall e \in E_i, e' \in E_j, i \neq j: \delta(e, e') = \delta(e', e) = (-\infty, +\infty)$
- $\Delta_i \subseteq \Delta$, Δ is a set of constraint functions restricted to event pairs in E_i : $\forall (e, e') \in \text{Dom}(\Delta_i) : e \in E_i \wedge e' \in E_i$
- $\prec\prec$ is the set of precedence orders between bundles such that $(E_i, \Delta_i) \prec\prec (E_j, \Delta_j)$, iff $\exists e \in E_i, e' \in E_j : \prec(e, e') \neq (-\infty, +\infty)$

$(\mathbb{B}, \prec\prec)$ is a directed acyclic graph. \square

Note that at the end of Definition 12, we require $(\mathbb{B}, \prec\prec)$ to be a DAG. The reason will become clear in the following subsection.

5.2 A Lattice of Bundled Interaction Diagrams and Its Properties

In the previous subsection, we show that all events in a timed interaction diagram can be partitioned such that the resultant event structure contains only acyclic precedence orders between bundles. In fact, given a timed interaction diagram, all bundled interaction diagrams satisfying Definition 12 forms a lattice with infimum given by Algorithm 3 shown later in this subsection and supremum given by bundling all events. For instance, with Example 2, the minimum bundling is $\{\{e_4, e_5\}, \{e_1, e_2, e_3\}, \{e_6\}\}$, the maximum bundling is $\{\{e_1, e_2, e_3, e_4, e_5, e_6\}\}$, and all valid partitions forms a lattice as shown in Fig. 8.

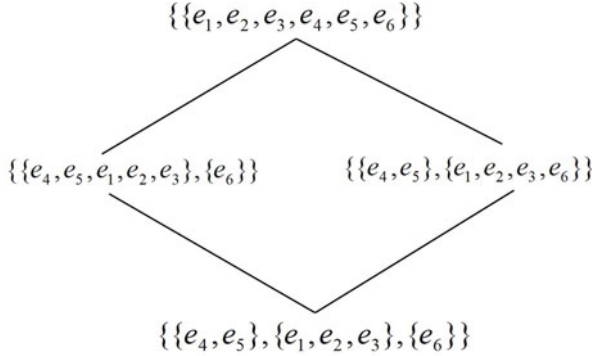


Fig. 8. A Lattice of Bundled Interaction Diagrams

As mentioned at the beginning of this section, checking for negative cycle in a directed weighted graph is asymptotically harder than checking for cycles in a directed unweighted graph. However, through bundling, we are able to reduce the time complexity by running the more time consuming algorithm on smaller problem sets. Algorithm 1 gives the details:

Algorithm 1. FEASIBILITY-CHECKING ($tid = (\rho, \chi)(I, O, \Delta)$)

- 1 **compute** the bundled interaction diagram $(B, \prec\prec)$ of tid such that the relations between bundles are only of acyclic causal orders;
 - 2 **if** $\forall B_i \in B$ is feasible **then**
 - 3 | tid is feasible;
 - 4 **else**
 - 5 | tid is not feasible;
 - 6 **end**
-

The correctness of the algorithm is supported by Theorem 4 below.

Theorem 4. *A timed interaction diagram is feasible if and only if every bundle in its corresponding bundled interaction diagram is feasible.* □

Proof. 1. Every bundle in bid is feasible implies that tid is feasible:

Prove by construction. Since every bundle in a bid is feasible, for every bundle $B_i = (E_i, \Delta_i)$ of \mathbb{B} there is a “local time mapping” (in contrast to global time mapping) $g_i : E \xrightarrow{inj} R$ which preserves Δ_i :

$$\begin{aligned}
 g_i(e_{i_1}) &= t_{i_1} \\
 &\vdots \\
 g_i(e_{i_k}) &= t_{i_k}
 \end{aligned}$$

Without loss of generality, suppose $t_{i_1} > 0, \dots, t_{i_k} > 0$ (this can be guaranteed by shifting every local time by $|\min_{j=1 \dots k} (t_{i_j})| + \epsilon$ which does not affect the preservation of Δ_i).

Since $(\mathbb{B}, \prec\prec)$ is a directed acyclic graph, there exist a topologically sorted order of all bundles in \mathbb{B} . The global time mapping $g_i : E \xrightarrow{inj} \mathbb{R}$ for the original timed interaction diagram can be constructed by the following algorithm (Algorithm 2):

Algorithm 2. CONSTRUCT-GLOBAL-TIME-MAPPING $(B, \prec\prec)$

```

1 foreach bundle  $B_i = (E_i, \Delta_i)$  in topologically sorted order from low to high do
2   for  $\forall B_{m_1} \prec\prec B_i, \dots, B_{m_n} \prec\prec B_i$  do
3     foreach  $e_{i_j} \in E_i$  do
4        $g(e_{i_j}) = g_i(e_{i_j}) + \max_{E_m \in \{E_{m_1} \dots E_{m_n}\}} \left( \max_{e_k \in E_m} g(e_k) \right)$ 
5     end
6   end
7 end

```

As the local times for every event in a bundle is consistently shifted by the same amount, hence the time change will not affect the feasibility within the bundle. Moreover, since the last line of the algorithm guarantees that events in a bundle of higher topologically sorted order have larger global times of events than events in a bundle of smaller topologically sorted order, causal relations between bundles are satisfied. Therefore, all constraints in the original timed interaction diagram are satisfied.

- 2. *tid* is feasible implies that every bundle in *bid* is feasible

Prove the contrapositive. It suffices to prove that if some bundle in *bid* is not feasible, then *tid* is not feasible. This follows from the fact that for a bundle $B_i = (E_i, \Delta_i)$, we have that $E_i \subseteq I \cup O$ and $\Delta_i \subseteq \Delta$ where I, O and Δ are the input events, output events and constraints defined in *tid*.

From the theorem, we can see that the feasibility checking process yields both positive and negative answers. Note that Algorithm 2 not only gives a proof of the correctness of checking with positive answers, but also gives a way to obtain a global time mapping from a set of local time mappings. For instance, in Example 1, one set of local time mappings for the three bundles could be:

$$\begin{aligned}
 B_1 &: g_1(e_4) = 1, g_1(e_5) = 1.3 \\
 B_2 &: g_2(e_1) = 1, g_2(e_2) = 2, g_2(e_3) = 6 \\
 B_3 &: g_3(e_6) = 1
 \end{aligned}$$

And the algorithm will run for each bundle in global topologically sorted order B_1, B_2, B_3 and find a global time mapping:

$$g(e_4) = 1, g(e_5) = 1, 3, g(e_1) = 2.3, \\ g(e_2) = 3.3, g(e_3) = 7.3, g(e_6) = 8.3$$

which satisfies all constraints.

By reducing the problem of feasibility checking of a timed interaction diagram to the problem of feasibility checking of individual bundles of the corresponding bundled interaction diagram, the complexity of the former can be dramatically reduced since the time-consuming Bellman-Ford algorithm need only be run within bundles. More specifically, assume a bundled interaction diagram has n bundles: $(E_1, \Delta_1), (E_2, \Delta_2), \dots, (E_n, \Delta_n)$. The time complexity for feasibility checking in the bundled interaction diagram is:

$$T_{bid} = \sum_{i=1}^n \mathcal{O}(|E_i| \cdot |\Delta_i|) \tag{21}$$

while the time complexity for feasibility checking in the original timed interaction diagram is:

$$T_{tid} = \mathcal{O}(|E| \cdot |\Delta|) \\ = \mathcal{O}\left(\sum_{i=1}^n |E_i| \cdot \left(\sum_{i=1}^n |\Delta_i|\right)\right) \tag{22} \\ = \sum_{i=1}^n \mathcal{O}(|E_i| \cdot |\Delta_i|) + \sum_{i=1}^n \mathcal{O}(|E_i| \cdot \sum_{j \neq i} |\Delta_j|)$$

which is $\sum_{i=1}^n \mathcal{O}(|E_i| \cdot \sum_{j \neq i} |\Delta_j|)$ larger. For example, in average case, a balanced partition would have

$$|E_1| = |E_2| = \dots = |E_n| = \frac{1}{n}|E| \\ |\Delta_1| = |\Delta_2| = \dots = |\Delta_n| = \frac{1}{n}|\Delta|$$

Thus, the complexity for feasibility checking in the bundled interaction diagram is $\mathcal{O}(|E| \cdot |\Delta|/n)$ while the complexity for feasibility checking in the timed interaction diagram is $\mathcal{O}(|E| \cdot |\Delta|)$. Therefore, the more bundles formed (larger n), the faster the feasibility checking can be done.

The worst case is when the bundle is the supremum of the lattice. In other words, all events are in the same bundle. In this case, $n = 1$ and the time complexity of Algorithm 1 becomes the same as the feasibility checking on the original timed interaction diagram. Another extreme would be $n = |E|$, this will imply that each bundle contains a single event. This can happen only when all constraints between events (Δ) are causal constraints. And the problem reduces

to feasibility checking of extended interaction diagram. However, given any timed interaction diagram, it is rarely possible that all constraints are causal, and thus the number of bundles can rarely be $|E|$.

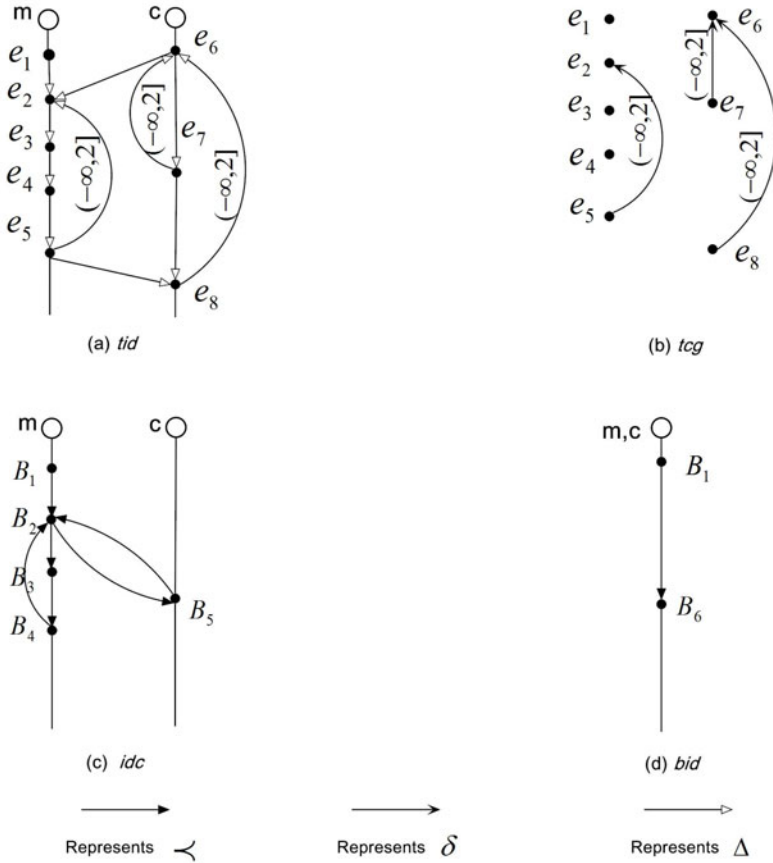


Fig. 9. Computing the Minimum Bundled Interaction Diagram

The following algorithm finds the infimum of the lattice of bundled interaction diagrams for a given timed interaction diagram. As we can see, the infimum has the maximum number of bundles so that the complexity of feasibility checking is minimized.

The following example illustrates the execution of the algorithm:

Example 3. In Fig. 9, *idc* is obtained by bundling $\{e_1\}$ to B_1 , $\{e_2, e_5\}$ to B_2 , $\{e_3\}$ to B_3 , $\{e_4\}$ to B_4 , $\{e_6, e_7, e_8\}$ to B_5 respectively. For simplicity of the presentation, constraints within bundles are not shown. The *bid* is obtained by bundling the strongly connected component $\{B_2, B_3, B_4, B_5\}$ in *idc* to B_6 .

Algorithm 3. MINIMUM-BUNDLE ($tid = (\rho, \chi)(I, O, \Delta)$, $tcg = (I, O, \delta)$)

```

1 foreach WEAKLY CONNECTED COMPONENT IN  $tcg$  do
2   | shrink events in the component into a "big event";
3   | //We call the intermediate diagram  $idc$ , which contains only
4   | causal constraints between "big events";
5   | merge causal constraints between "big events";
6 end
7 foreach STRONGLY CONNECTED COMPONENT IN  $idc$  do
8   | bundle "big events" in the component;
9   | merge causal constraints between bundles to form a  $bid$ ;
10 end

```

B_1 does not change. It is noticeable in Fig. 9 that there exist constraint loops which indicate the feasibility checking process will yield negative answers. However, at this point, we only concern about the procedure of minimum bundling, the feasibility checking problems will be left within the bundles after the bundles have been constructed. \square

Since the complexity of computing both weakly and strongly connected component of a graph $G(V, E)$ is $\mathcal{O}(|V| + |E|)$, the minimum bundled interaction diagram can be obtained is $\mathcal{O}(|I \cup O| + |\Delta|)$ which can be ignored compared with the feasibility checking processes.

The following theorem shows that given a timed interaction diagram and its underlying timing constraint graph, the bundled interaction diagram constructed by Algorithm 3 is unique.

Theorem 5. *The bundled interaction diagram constructed by Algorithm 3 is unique for a given timed interaction diagram tid and its underlying timing constraint graph tcg .* \square

Proof. To prove the theorem, it suffices to prove that

1. The intermediate diagram (idc) that only contains causal constraints between "big events" in Algorithm 3 can be uniquely constructed from tid and tcg .
This is true since the set of weakly connected components in tcg is unique, thus partition of events in tid according to weakly connected components in tcg is unique.
2. bid in Algorithm 3 can be uniquely constructed from idc
This is true since the set of strongly connected component in idc is unique, thus partition of "big events" in idc according to its strongly connected components is unique.

An alternative way of constructing bid from idc can be done by contracting cycles in idc cumulatively, i.e., up till idc is left with no more cycles, iterate over by contracting a cycle and look for another. This leads to a DAG. The resultant DAG is unique independent of the order in which we contract the cycles. This can be proven using cut-and-paste argument [7].

In Example 3, e_2 through e_8 are bundled. It seems that Algorithm 3 may produce large bundles of events such that the resultant bundled event structure is too coarse. However, as shown in the following theorem, such partition is necessary and minimal.

Theorem 6. *The bundled interaction diagram constructed by Algorithm 3 forms a partition of events in an interaction diagram id . The bundling is minimal to ensure the correctness of the feasibility checking process in Algorithm 3. \square*

Proof. Prove by contradiction.

1. Suppose that two events in a weakly connected component in tcg are in two bundles B_1 and B_2 in bid . Then there must be at least one timing constraint between B_1 and B_2 , thus the two bundles are timing constrained. Therefore, local feasibility checking within bundles is not sufficient to guarantee the feasibility of the original timed interaction diagram.
2. Suppose that two events in a strongly connected component in idc are in two bundles B_1 and B_2 in bid . Then there must be at least one precedence constraint path from B_1 to B_2 and at least one precedence constraint path from B_2 to B_1 . This forms a precedence constraint loop so that topological sorted order in Algorithm 1 is not obtainable.

Therefore, the bundling is minimal in the sense that any further partitions of any bundle in the bundled interaction diagram given by Algorithm 3 will either break the preservation of valid bundled interaction path under timing constraint or produce a non-feasible precedence constrained event structure.

Theorem 5 and 6 together imply that the bundled interaction diagram constructed by Algorithm 3 is a unique infimum of the lattice of bundled interaction diagrams. In fact, all the other bundled interaction diagrams of a given timed interaction diagram subject to Definition 12 can be obtained by further merging some bundles that does not introduce precedence loops in the infimum. This can be proven in a similar way as Theorem 6.

As shown above, the infimum in the lattice of bundled interaction diagram offers the largest improvement of the feasibility checking while the supremum offers the least improvement (since the complexity is the same as that without bundling). It is also easy to show that the complexities of feasibility checking of bundled interaction diagrams comply with the partial order of the lattice: two bundled interaction diagrams $bid_1 = \{(E_1, \Delta_1), (E_2, \Delta_2), \dots, (E_n, \Delta_n)\}$ and $bid_2 = \{(E_1 \cup E_2, \Delta_1 \cup \Delta_2), \dots, (E_n \cup E_n, \Delta_n \cup \Delta_n)\}$ would have $bid_1 \prec bid_2$ in the lattice (if further bundling of some bundles in bid_1 to form bid_2 does not introduce precedence loop; also note that \prec is overloaded here to denote partial orders in the lattice) and feasibility checking in bid_2 would take longer time than in bid_1 since

$$T_{bid_2} - T_{bid_1} = \mathcal{O}(|E_1| \cdot |\Delta_2| + |E_2| \cdot |\Delta_1|).$$

6 Conclusion

In distributed real-time systems, it is crucial to be able to reason about whether two systems have equivalent behaviors. Instead of using traditional system state-based transition systems to study the properties of distributed real-time systems, we base our formal reasoning on the observable interactions among distributed entities. A timed interaction graph is defined to represent the systems both functional (observable) properties and timing properties. Based on the timed interaction graph, we have shown that:

- The basic interaction diagram is not sufficient for describing distributed real-time systems. However, extensions to it will require feasibility checking on the new event structures.
- Given an interaction graph $(\rho, \chi)(I, O, \prec)$ and a feasible timing constraint graph $(I \cup O, \delta)$, the composed diagram, i.e., timed interaction diagram $(\rho, \chi)(I, O, \Delta)$, preserves both precedence constraints \prec and real-time constraints δ .
- By bundling events in a timed interaction diagram, we obtain an event structure called bundled interaction diagram whose interaction paths defined on bundled events preserve validity after timing constraints are added to the original interaction diagram.
- Instead of checking the entire timed interaction diagram, we show that the feasibility checking can be done within each bundle in the bundled interaction diagram so that the efficiency of feasibility checking can be improved.
- All bundled interaction diagrams of a given timed interaction diagram satisfying Definition 12 form a lattice with a unique infimum given by Algorithm 3 and a unique supremum given by bundling all events. If $bid_1 \prec bid_2$ in the lattice, feasibility checking using bid_1 takes less time than using bid_2 .

The discussion in this paper has been on the semantics and feasibility problems when the precedence/real-time constraints are directly imposed upon computational units. However, when computational units are clustered into groups based on their functional behaviors, and the precedence/real-time constraints are imposed upon those groups instead of directly upon individual units, the feasibility problems become different since infeasible timed interaction diagrams may become feasible by replicating and actively coordinating computational units within each group. Our immediate future work is to investigate the feasibility problems in a simple three-tier coordination model [14] and the relationship between bundling based on timing properties and grouping based on behaviors. We will further look into the role of grouping and the role of coordinating of homogeneous behaviors within a group as well as heterogeneous behaviors among different groups.

References

1. Agha, G.: *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge (1986)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: Towards a theory of actor computation. In: Cleaveland, W.R. (ed.) *CONCUR 1992*. LNCS, vol. 630, pp. 565–579. Springer, Heidelberg (1992)
3. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. *J. Funct. Program.* 7, 1–72 (1997), <http://portal.acm.org/citation.cfm?id=969900.969901>
4. Agha, G.A., Thati, P., Ziaei, R.: *Actors: a model for reasoning about open distributed systems*, pp. 155–176. Cambridge University Press, New York (2001), <http://portal.acm.org/citation.cfm?id=566795.566806>
5. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. *Science of Computer Programming* 55(1-3), 3–52 (2005), formal Methods for Components and Objects: Pragmatic aspects and applications
6. Clinger, W.D.: *Foundations of Actor Semantics*. Ph.D. thesis (1981), aI-TR-633
7. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: *Introduction to Algorithms*, 2nd edn. McGraw-Hill Higher Education (2001)
8. Jiang, J., Wu, J.: The preservation of interleaving equivalences. In: *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 580–589. IEEE Computer Society, Washington, DC, USA (2005)
9. Lee, C.G., Mok, A.K., Konana, P.: Monitoring of timing constraints with confidence threshold requirements. *IEEE Trans. Comput.* 56, 977–991 (2007), <http://dx.doi.org/10.1109/TC.2007.1026>
10. Lee, E.A.: Concurrent semantics without the notions of state or state transitions. In: Asarin, E., Bouyer, P. (eds.) *FORMATS 2006*. LNCS, vol. 4202, pp. 18–31. Springer, Heidelberg (2006)
11. Mason, I.A., Talcott, C.L.: Actor languages. their syntax, semantics, translation, and equivalence. *Theor. Comput. Sci.* 220, 409–467 (1999), <http://portal.acm.org/citation.cfm?id=308049.308053>
12. Mok, A.K., Liu, G.: Efficient run-time monitoring of timing constraints. In: *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS 1997)*, p. 252. IEEE Computer Society, Washington, DC, USA (1997), <http://portal.acm.org/citation.cfm?id=523983.828388>
13. Raju, S., Rajkumar, R., Jahanian, F.: Monitoring timing constraints in distributed real-time systems. In: *Real-Time Systems Symposium, 1992*, pp. 57–67 (December 1992)
14. Ren, S., Yu, Y., Chen, N., Marth, K., Poirot, P.-E., Shen, L.: Actors, roles and coordinators! a coordination model for open distributed and embedded systems. In: Ciancarini, P., Wiklicky, H. (eds.) *COORDINATION 2006*. LNCS, vol. 4038, pp. 247–265. Springer, Heidelberg (2006)
15. Talcott, C.L.: Interaction semantics for components of distributed systems. In: *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS 1996* (1996)
16. Talcott, C.L.: Composable semantic models for actor theories. *Higher Order Symbol. Comput.* 11, 281–343 (1998)
17. Yu, Y., Ren, S., Frieder, O.: Prediction of timing constraint violation for real-time embedded systems with known transient hardware failure distribution model. In: *27th IEEE International on Real-Time Systems Symposium, RTSS 2006*, p. 454–466 (December 2006)

Puff, The Magic Protocol

Farhad Arbab^{1,2}

¹ Foundations of Software Engineering, CWI, Science Park 123, 1098 XG Amsterdam

² Leiden Institute for Advanced Computer Science, Leiden University,
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands

farhad@cwi.nl

Abstract. Traditional models of concurrency resort to peculiarly indirect means to express interaction and study its properties. Formalisms such as process algebras/calculi, concurrent objects, actors, agents, shared memory, message passing, etc., all are primarily action-based models that provide constructs for the direct specification of *things that interact*, rather than a direct specification of *interaction* (protocols). Consequently, interaction in these formalisms becomes a derived or secondary concept whose properties can be studied only indirectly, as the side-effects of the (intended or coincidental) couplings or clashes of the *actions* whose compositions comprise a model.

Treating interaction as an explicit first-class concept, complete with its own composition operators, allows to specify more complex interaction protocols by combining simpler, and eventually primitive, protocols. Reo [20,11,12,6] serves as a premier example of such an interaction-based model of concurrency. In this paper, we describe Reo and its support tools. We show how exogenous coordination in Reo reflects an interaction-centric model of concurrency where an interaction (protocol) consists of nothing but a relational constraint on communication actions. In this setting, interaction protocols become explicit, concrete, tangible (software) constructs that can be specified, verified, composed, and reused, independently of the actors that they may engage in disparate applications.

*Puff, the magic dragon ad-libbed concurrency,
As he frolicked through the mist of code disguised invisibly.
Little Jackie Paper loved that rascal Puff,
And brought him threads and semaphores and other fancy stuff. Oh ...*

–PeterPaulAndMary

1 Introduction

Concurrency is inherently difficult because it involves complex interaction protocols. Yet, it is always possible to make already difficult subjects even more difficult by increasing the complexity of their treatment. We take full advantage of this fact in cryptography by seeking easy disguising transformations whose

inverses are so complex as to make them prohibitively difficult (if not impossible) to perform, without knowing a key (piece of information). We use unnecessary complexity to disguise things for entertainment, as in puzzles, for instance, as well. In most other situations, though, we do not choose to increase the complexity of a problem; at least not intentionally. However, it seems to me that the historically justifiable optimum path that led us out of the realm of sequential programming, into the new world of concurrency, has hindered us from realizing how our old-country world view unnecessarily increases the complexity of life in this new world. Once we had mastered the skills to navigate through the realm of sequential programming, the simplest models of concurrency seemed to require only the addition of a few new constructs to our models of sequential programming: a befitting selection of locks, semaphores, mutual exclusion, monitors, send/receive primitives, message passing, rendezvous, etc., would do. Refinements in concurrency theory abstracted away inconsequential sequential computations to offer process calculi and algebras. In many ways, these models are indeed simple. Alas, simple models are not always simple to use.

With the availability of today's low-cost multicore commodity hardware that can scale up to offer massively parallel computing platforms, high-speed communication networks that interconnect the globe, plus every indication that both of these phenomena constitute trends that will continue in the future, the need for programming techniques to harness the massive concurrency that they offer has become more vivid than ever. The inadequacy of traditional models for programming of concurrent systems to serve this purpose stems from the fact that the way in which they express interaction protocols generally does not scale up.

In spite of the fact that *interaction* constitutes the most challenging aspect of concurrency, traditional models of concurrency predominantly treat interaction as a secondary or derived concept. Shared memory, message passing, calculi such as CSP [50], CCS [83], the π -calculus [84,97], process algebras [33,25,46], and the actor model [8] represent popular approaches to tackle the complexities of constructing concurrent systems. Beneath their significant differences, all these models share one common characteristic, inherited from the world of sequential programming: they all constitute *action*-based models of concurrency.

For example, consider developing a simple concurrent application with two producers, which we designate as Green and Red, and one consumer. The consumer must repeatedly obtain and display the contents alternately made available by the Green and the Red producers.

Figure 1 shows the pseudo code for an implementation of this simple application in a Java-like language. Lines 1-4 in this code declare four globally shared entities: three semaphores and a buffer. The semaphores `greenSemaphore` and `redSemaphore` are used by their respective Green and Red producers for their turn keeping. The semaphore `bufferSemaphore` is used as a mutual exclusion lock for the producers and the consumer to access the shared `buffer`, which is initialized to contain the empty string. The rest of the code defines three processes: two producers and a consumer.

```

Global Objects:
1 private final Semaphore greenSemaphore = new Semaphore(1);
2 private final Semaphore redSemaphore = new Semaphore(0);
3 private final Semaphore bufferSemaphore = new Semaphore(1);
4 private String buffer = EMPTY;

Green Producer:
14 while (true) {
15     sleep(5000);
16     greenText = ...;
17     greenSemaphore.acquire();
18     bufferSemaphore.acquire();
19     buffer = greenText;
20     bufferSemaphore.release();
21     redSemaphore.release();
22 }

Consumer:
5 while (true) {
6     sleep(4000);
7     bufferSemaphore.acquire();
8     if (buffer != EMPTY) {
9         println(buffer);
10        buffer = EMPTY;
11    }
12    bufferSemaphore.release();
13 }

Red Producer:
23 while (true)
24     sleep(3000);
25     redText = ...;
26     redSemaphore.acquire();
27     bufferSemaphore.acquire();
28     buffer = redText;
29     bufferSemaphore.release();
30     greenSemaphore.release();
31 }

```

Fig. 1. Alternating producers and consumer

The consumer code (lines 5-13) consists of an infinite loop where in each iteration, it performs some computation (which we abstract as the `sleep` on line 6), then it waits to acquire exclusive access to the buffer (line 7). While it has this exclusive access (lines 8-11), it checks to see if the buffer is empty. An empty buffer means there is no (new) content for the consumer process to display, in which case the consumer does nothing and releases the buffer lock (line 12). If the buffer is non-empty, the consumer prints its content and resets the buffer to empty (lines 9-10).

The Green producer code (lines 14-22) consists of an infinite loop where in each iteration, it performs some computation and assigns the value it wishes to produce to local variable `greenText` (lines 14-15), and waits for its turn by attempting to acquire `greenSemaphore` (line 17). Next, it waits to gain exclusive access to the shared buffer, and while it has this exclusive access, it assigns `greenText` into `buffer` (lines 18-20). Having completed its turn, the Green producer now releases `redSemaphore` to allow the Red producer to have its turn (line 21).

The Red producer code (lines 23-31) is analogous to that of the Green producer, with “red” and “green” swapped.

This is a simple concurrent application whose code has been made even simpler by abstracting away its computation and declarations. Apart from their trivial outer infinite loops, each process consists of a short piece of sequential code, with a straight-line control flow that involves no inner loops or non-trivial branching. The protocol embodied in this application, as described in our problem statement, above, is also quite simple. One expects it be easy, then, to answer a number of questions about what specific parts of this code manifest the various properties of our application. For instance, consider the following questions:

1. Where is the green text computed?
2. Where is the red text computed?
3. Where is the text printed?

The answers to these questions are indeed simple and concrete: lines 16, 25, and 9, respectively. Indeed, the “computation” aspect of an application typically correspond to coherently identifiable passages of code. However, the perfectly legitimate question “Where is the protocol of this application?” does not have such an easy answer: the protocol of this application is intertwined with its computation code. More refined questions about specific aspects of the protocol have more concrete answers:

1. What determines which producer goes first?
2. What ensures that the producers alternate?
3. What provides protection for the global shared buffer?

The answer to the first question, above, is the collective semantics behind lines 1, 2, 17, and 26. The answer to the second question is the collective semantics behind lines 1, 2, 17, 26, 21, and 30. The answer to the third question is the collective semantics of lines 3, 18, 20, 27, and 29. These questions can be answered by pointing to fragments of code scattered among and intertwined with the computation of several processes in the application. It is far more difficult to identify other aspects of the protocol, such as possibilities for deadlock or livelock, with concrete code fragments. While both concurrency-coordinating actions and computation actions are concrete and explicit in this code, the interaction protocol that they induce is implicit, nebulous, and intangible. In applications involving processes with even slightly less trivial control flow, the entanglement of data and control flow with concurrency-coordination actions makes it difficult to determine which parts of the code give rise to even the simplest aspects of their interaction protocol.

When the protocol in a typical concurrent application consists of 623 send and receive (or lock/unlock, etc.) primitives, sprinkled over 783,961 lines of C code, chopped up into 387 different source files, how simple is it to understand this protocol, reason about its properties, debug it, adapt it, or imagine reusing it in another application? How can a hapless programmer (who may very well be the original author of the code, six months down the road) even *see* what this protocol actually does before he can contemplate to do anything with it? Even in the case of our simple program in Figure 1, which we just examined, do we see all of its properties? We asked about and identified the buffer protection mechanism in this application. But does this mechanism provide adequate protection that we expect?

It is only tactful of me to say that I am sure all my readers have already spotted what may be considered a bug in this code that may in fact remain undetected in practice for a very long time, depending on the circumstances that determine the relative speeds of the producer and consumer threads in this application. There is no protection in this code preventing the producers from over-writing each other in the buffer, regardless of whether or not their output

```

Green Producer:
14 while (true) {
15     sleep(5000);
16     greenText = ...;
17     greenSemaphore.acquire();
18     while (greenText !=EMPTY) {
19         bufferSemaphore.acquire();
20         if (buffer == EMPTY) {
21             buffer = greenText;
22             greenText = EMPTY;
23         }
24         bufferSemaphore.release();
25     }
26     redSemaphore.release();
27 }

Red Producer:
28 while (true)
29     sleep(3000);
30     redText = ...;
31     redSemaphore.acquire();
32     while (redText !=EMPTY) {
33         bufferSemaphore.acquire();
34         if (buffer == EMPTY) {
35             buffer = redText;
36             redText = EMPTY;
37         }
38         bufferSemaphore.release();
39     }
40     greenSemaphore.release();
41 }

```

Fig. 2. Busy waiting consumer

has actually been consumed by the consumer. Strictly speaking, the original statement of our requirements does not forbid this behavior, so whether this is a bug (in the specification or implementation) is unclear. Suppose the intention in fact was for the consumer to alternately consume what the two producers produce, which means the implementation in Figure 1 is incorrect and we need to alter it.

One solution is to make the producers sensitive to the emptiness of the buffer. The code for the new producers appears in Figure 2. A disadvantage of this code is that it more heavily uses the busy-waiting mechanism that already existed in the consumer code in Figure 1. A better alternative is to use a different protocol that explicitly respects the turn taking, as described below.

In the program shown in Figure 3, the consumer too has its own turn-taking semaphore, the new `blueSemaphore` (line 3), which is initialized to be locked, just as the `redSemaphore`, because initially, there is nothing for the consumer to do before any of the producers produces something. The initialization of the `bufferSemaphore` is also changed (line 4), making the buffer initially locked on behalf of the first producer. The consumer and the two producers all can proceed until each reaches its own turn-taking lock on lines 8, 15, and 24, respectively. The consumer and the Red producer suspend themselves on their turn-taking locks, but the Green producer can proceed beyond its turn-taking lock (line 15), where it fills the buffer (line 16), releases the turn-taking lock of the consumer (line 17), and suspends itself on the buffer lock (line 18). Only the consumer can now proceed, printing the content of the buffer (line 9), and releasing the buffer lock (line 10), after which it proceeds with its next iteration in which it suspends itself on its turn-taking lock (line 8). Only the Green producer can now proceed, having obtained the buffer lock. It now completes its iteration by releasing the turn-taking lock of the Red producer (line 19), and starts its next iteration in which it suspends itself on its own turn-taking lock (line 15). Now, only the Red


```

Global Objects:
1 private final Semaphore greenSemaphore = new Semaphore(1);
2 private final Semaphore redSemaphore = new Semaphore(0);
3 private final Semaphore blueSemaphore = new Semaphore(0);
4 private final Semaphore bufferSemaphore = new Semaphore(0);
5 private String buffer = EMPTY;

Green Producer:
12 while (true) {
13     sleep(5000);
14     greenText = ...;
15     greenSemaphore.acquire();
16     buffer = greenText;
17     blueSemaphore.release();
18     bufferSemaphore.acquire();
19     redSemaphore.release();
20 }

Consumer:
6 while (true) {
7     sleep(4000);
8     blueSemaphore.acquire();
9     println(buffer);
10    bufferSemaphore.release();
11 }

Red Producer:
21 while (true)
22     sleep(3000);
23     redText = ...;
24     redSemaphore.acquire();
25     buffer = redText;
26     blueSemaphore.release();
27     bufferSemaphore.acquire();
28     greenSemaphore.release();
29 }

```

Fig. 3. Revised alternating producers and consumer

producer can proceed to fill the buffer (line 25), release the turn-taking lock of the consumer (line 26), and suspend itself on the buffer lock (line 27). The consumer now goes through another iteration, at the end of which it releases the buffer lock, allowing only the Red producer to proceed. The Red producer now releases the turn-taking lock of the Green producer (line 29), and starts its next iteration in which it suspends itself on its own turn-taking lock (line 24) again.

Now that we have a correct protocol (if we indeed do) that does what we expect it to do (if it indeed does), what can we do with this protocol? How easy is it, for instance to reuse this same protocol in a more elaborate application where the control flow of the processes is more complex than the essentially linear, sequential flow of these simple processes? Is it possible to bundle up this protocol and parameterize it such that we can instantiate the protocol with arbitrary numbers of and computation code for processes, the same way that we can package a piece of code into a parameterized function to compute the inverse of a matrix of any size, or find the minimum element in a list of any size? It would certainly help in software development for multicore platforms, for instance, if we could simply specify the desired numbers and participants for an abstract parameterized protocol, as easily as passing arguments in a function call, to tailor the desired concurrency on the available cores. How easy is it to alter this protocol to change the imposed ordering or to allow a pair of considerably fast producers go as fast as they wish, while the slower consumer merely *samples* their output? Such manipulations are difficult with this and similar incarnations of a protocol because they require *seeing* and *touching* the protocol as a tangible concrete entity.

Seeing concurrency protocols through the mist of source code, reminds me of my experience with autostereograms that suddenly burst into popularity in the 1990's in *Magic Eye* books. In fact, there are different types of autostereograms

and this particular type is called *random dot autostereograms* which hide a 3D image behind a pattern of seemingly random dots. The hidden 3D image emerges and becomes perceptible only when the incoherent 2D picture of random dots is viewed just the *right way*. To accomplish this feat, one needs to learn the skill to overcome the brain's normally automatic coordination between its mechanisms for the eyes' *focus* and *vergence*. With the correct vergence, the 3D scene suddenly pops into existence, but let the normal brain mechanism that ties vergence to focus take over, and puff, it's gone! It is inaccurate to call this phenomenon an optical illusion, because the 3D image is really there: all the depth information as well as its other characteristics truly exist embedded within the mist of random dots. It is nontrivial to learn the skill to see these 3D images because to do so is contrary to how our brains are wired to tell each eye where to look as we focus on what we see.

The protocol in a concurrent program is as real as the 3D image in a random dot autostereogram: all information necessary for its manifestation really exists, scattered, embedded within the bulk of the source code, most of which is just as irrelevant to the protocol and hinders its recognition as the random dots are to the 3D image and hinder its recognition. Seeing the protocol requires nontrivial skills that defy our natural balance of mental vergence and focus of attention. Constructing a random dot autostereogram requires intricate mathematical models and sophisticated calculations that do not resemble anything like sculpting or drawing a 3D image. Constructing a protocol in this form requires intricate mathematical models and sophisticated calculations that do not resemble anything like sequential programming. The 2D picture of a random dot autostereogram only indirectly contains its embedded 3D image, whose manifestation requires the active participation of an observer. The source code of a concurrent program only indirectly contains its embedded protocol, whose manifestation requires the active participation of a human or computer *observer*. Both the 3D images of random dot autostereograms and the protocols of concurrent programs can be constructed and manipulated only indirectly, through generally non-intuitive manipulations of seemingly unrelated tangible concrete objects scattered throughout the scene. Even the simplest manipulations of an autostereogram, such as scaling, can change the 3D image non-intuitively and produce strange unexpected results. It is just as perilous and misguided to try to alter a protocol or reuse (perhaps a part of) it in another program by directly manipulating or copying source code, as it is to try to alter a 3D image or reuse (perhaps a part of) it in another autostereogram by directly manipulating or copying random dots.

Process algebraic models of concurrency fair only slightly better in this regard than, e.g., programming with threads: they too embody an action-based model of concurrency. Figure 4 shows a process algebraic model of our alternating producers and consumer application. This model consists of a number of globally shared names, i.e., *g*, *r*, *b*, and *d*. Generally, these shared names are considered as abstractions of channels and thus are called “channels” in the process algebra/calculi community. However, since these names in fact serve no purpose

```

Global Names:                                Green Producer:
synchronization-points g, r, b, d           G := genG(t) . ?g(k) . !b(t) . ?d(j) . !r(k) . G

Consumer:                                    Red Producer:
B := ?b(t) . print(t) . !d("done") . B     R := genR(t) . ?r(k) . !b(t) . ?d(j) . !g(k) . R

Application:
G | R | B | !g("token")

```

Fig. 4. Alternating producers and consumer in a process algebra

other than synchronizing the I/O operations performed on them, and because we will later use the term “channel” to refer to entities with more elaborate behavior, we use the term “synchronization points” here to refer to “process algebra channels” to avoid confusion.

A process algebra consists of a set of atomic actions, and a set of composition operators on these actions. In our case, the atomic actions include the primitive actions read $?_-(_)$ and write $!_-(_)$ defined by the algebra, plus the user-defined actions $\text{genG}(_)$, $\text{genR}(_)$, and $\text{print}(_)$, which abstract away computation. Typical composition operators include sequential composition $_ \cdot _$, parallel composition $_ | _$, nondeterministic choice $_ + _$, definition $_ := _$, and implicit recursion.

In our model, the consumer B waits to read a data item into t by synchronizing on the global name b , and then proceeds to print t (to display it). It then writes a token “done” on the synchronization point d , and recurses. The Green producer G first generates a new value in t , then waits for its turn by reading a token value into k from g . It then writes t to b , and waits to obtain an acknowledgment j through d , after which it writes the token k to r , and recurses. The Red producer R behaves similarly, with the roles of r and g swapped. The application consists of a parallel composition of the two producers and the consumer, plus a trivial process that simply writes a “token” on g to kick off process G to go first.

Observe that a model is constructed by composing (atomic) actions into (more complex) actions, called processes. True to their moniker, such formalisms are indeed *algebras of processes* or actions. Just as in the version in Figure 3, while communication actions are concrete and explicit in the incarnation of our application in Figure 4, *interaction* is a manifestation of the model with no direct explicit structural correspondence. Nevertheless, process algebraic incarnations of concurrency protocols are obviously simpler and more concise than their incarnations in typical programming languages, primarily because they abstract away the clutter of computation.

Returning to our autostereogram analogy, it is as if we compare a random dot autostereogram with a so-called *wallpaper autostereogram*. A wallpaper autostereogram is the simplest type of autostereogram and consists of a horizontally repeating pattern of nearly identical pictures. Roughly, it is the random dot autostereogram with the cluttering random dots peeled off, which allows even casual observers to get a good idea of what the 3D image is all about, without requiring them to exert their perception skills to actually experience the 3D

image. The individually identifiable repeating patterns of a wallpaper autostereogram seem more concrete, and in some sense more well-packaged and more reusable than the almost amorphous expanse of a random dot autostereogram. Nevertheless, a cavalier attempt to edit or cut and paste parts of a wallpaper autostereogram is no more likely to produce the desired alteration to its 3D image than in the case of a random dot autostereogram. Successful alteration of a process algebraic specification requires the same unnatural detachment of focus (on local manipulation) and vergence (to see its global effects) as is required to successfully alter the protocol of a concurrent C or Java application.

Indeed, in all action-based models of concurrency, interaction becomes a by-product of processes executing their respective actions: when a process A happens to execute its i_{th} communication action a_i on a synchronization point, at the same time that another process B happens to execute its j_{th} communication action b_j on that same synchronization point, the actions a_i and b_j “collide” with one another and their collision yields an interaction. Manifested this way, an interaction protocol consists of a desired temporal sequence of such (coincidental or planned) collisions. It is non-trivial to distinguish between the essential and the coincidental collision sequences, when the protocol itself is only such an ephemeral manifestation.

Generally, the reason behind the specific collision of a_i and b_j remains debatable. Perhaps it was just dumb luck. Perhaps it was divine intervention. Some may prefer to attribute it to intelligent design! What is not debatable is the fact that, a split second earlier or later, perhaps in another run of the same application, completely random cosmic rays may zap a memory bit and trigger the automatic hardware error correction of the affected memory cell, and thus change the relative timing of the running processes, making a_i and b_j collide not with each other, but with two other actions (of perhaps other processes) yielding completely different interactions. Action based models of concurrency make protocols more difficult than necessary to specify, manipulate, verify, debug, and next to impossible to reuse.

Instead of explicitly composing (communication) actions to indirectly specify and manipulate implicit interactions, is it possible to devise a model of concurrency where interaction (not action) is an explicit, first-class construct? We tend to this question in the next section and in the remainder of this paper describe a specific language based on an interaction-centric model of concurrency. We show that making interaction explicit leads to a clean separation of computation and communication, and produces reusable, tangible protocols that can be constructed and verified independently of the processes that they engage.

2 Interaction Centric Concurrency

The most salient characteristic of *interaction* is that it transpires among two or more actors. This is in contrast to *action*, which is what a single actor manifests. In other words, interaction is not about the specific actions of individual actors, but about the relations that (must) hold among those actions. A model of interaction, thus, must allow us to directly specify, represent, construct, compose,

decompose, analyze, and reason about those relations that define what transpires among two or more engaged actors, without the necessity to be specific about their individual actions. Making interaction a first-class concept means that a model must offer (1) an explicit, direct, concrete representation of the interaction among actors, independent of their (communication) actions; (2) a set of primitive interactions; and (3) composition operators to combine (primitive) interactions into more complex interactions.

Wegner has proposed to consider coordination as constrained interaction [101]. We propose to go a step further and consider interaction itself as a constraint on (communication) actions. Features of a system that involve several entities, for instance the clearance between two physical objects, cannot conveniently be associated with any one of those entities. It is quite natural to specify and represent such features as *constraints*. The interaction among several active entities has a similar essence: although it involves them, it does not *belong* to any one of those active entities. Constraints have a natural formal model as mathematical relations, which are non-directional. In contrast, actions correspond to functions or mappings which are directional, i.e., transformational.

A constraint declaratively specifies *what* must hold in terms of a relation. Typically, there are many ways in which a constraint can be enforced or violated, leading to many different sequences of actions that describe precisely *how* to enforce or maintain a constraint. Action-based models of concurrency lead to the precise specification of *how* in terms of sequences of actions interspersed among the active entities involved in a protocol. In an interaction-based model of concurrency, only *what* a protocol represents is specified as a constraint over the (communication) actions of some active entities; as in constraint programming, the responsibility of how the protocol constraints are enforced or maintained is relegated to an entity other than those active entities.

Generally, composing the sequences of actions that manifest two different protocols does not yield a sequence of actions that manifests a composition of those protocols. Thus, in action-based models of concurrency, protocols are not compositional. Represented as constraints, in an interaction-based model of concurrency, protocols can be composed as mathematical relations.

Banishing the actions that comprise protocol fragments out of the bodies of processes produces simpler, cleaner, and more reusable processes. Expressed as constraints, pure protocols become first-class, tangible, reusable constructs in their own right. As concrete software constructs, such protocols can be embodied into architecturally meaningful *connectors*.

In this setting, a process (or thread, component, service, actor, agent, etc.) offers no methods, functions, or procedures for other entities to call, and it makes no such calls itself. Moreover, processes cannot exchange messages through targeted send and receive actions. In fact, a process cannot refer to any foreign entity, such as another process, the mailbox or message queue of another process, shared variables, semaphores, locks, etc. The only means of communication of a process with its outside world is through *blocking I/O operations* that it may perform exclusively on its own *ports*, producing and consuming passive data.

A port is a construct analogous to a file descriptor in a Unix process, except that a port is unidirectional, has no buffer, and supports blocking I/O exclusively.

If i is an input port of a process, there are only two operations that the process can perform on i : (1) blocking input $\text{get}(i, v)$ waits indefinitely or until it succeeds to obtain a value through i and assigns it to variable v ; and (2) input with time-out $\text{get}(i, v, t)$ behaves similarly, except that it unblocks and returns false if the specified time-out t expires before it obtains a value to assign to v . Analogously, if o is an output port of a process, there are only two operations that the process can perform on o : (1) blocking output $\text{put}(o, v)$ waits indefinitely or until it succeeds to dispense the value in variable v through o ; and (2) output with time-out $\text{put}(o, v, t)$ behaves similarly, except that it unblocks and returns false if the specified time-out t expires before it dispenses the value in v .



Fig. 5. Protocol in a connector

Inter-process communication is possible only by mediation of connectors. For instance, Figure 5 shows a producer, P and a consumer C whose communication is coordinated by a simple connector. The producer P consists of an infinite loop in each iteration of which it computes a new value and writes it to its local output port (shown as a small circle on the boundary of its box in the figure) by performing a blocking put operation. Analogously, the consumer C consists of an infinite loop in each iteration of which it performs a blocking get operation on its own local input port, and then uses the obtained value. Observe that, written in an imperative programming language, the code for P and C is substantially simpler than the code for the Green/Red producers and the consumer in Figures 1, 2, and 3: it contains no semaphore operations or any other inter-process communication primitives.

The direction of the connector arrow in Figure 5 suggests the direction of the dataflow from P to C . However, even in the case of this very simple example, the precise behavior of the system crucially depends on the specific protocol that this simple connector implements. For instance, if the connector implements a synchronous protocol, then it forces P and C to iterate in lock-step, by synchronizing their respective put and get operations in each iteration. On the other hand the connector may have a bounded or an unbounded buffer and implement an asynchronous protocol, allowing P to produce faster than C can consume. The protocol of the connector may, for instance enable it to replicate data items, e.g., the last value that it contained, if C consumes faster and drains the buffer. The protocol may mandate an ordering other than FIFO on the contents of the connector buffer, perhaps depending on the contents of the exchanged data. It may retain only some of the contents of the buffer (e.g., only the first or the last

item) if P produces data faster than C can consume. It may be unreliable and lose data nondeterministically or according to some probability distribution. It may retain data in its buffer only for a specified length of time, losing all data items that are not consumed before their expiration dates. The alternatives for the connector protocol are endless, and composed with the very same P and C, each yields a totally different system.

A number of key observations about this simple example are worth noting. First, Figure 5 is an architecturally informative representation of this system. Second, banishing all inter-process communication out of the communicating parties, into the connector, yields a “good” system design with the beneficial consequences that:

- changing P, C, or the connector does not affect the other parts of the system;
- although they are engaged in a communication with each other, P and C are oblivious to each other, as well as to the actual protocol that enables their communication;
- the protocol embodied in the connector is oblivious to P and C.

In this architecture, the composition of the components and the coordination of their interactions are accomplished *exogenously*, i.e., from outside of the components themselves, and without their “knowledge”¹. In contrast, the interaction protocol and coordination in the examples in Figures 1, 2, 3, and 4 are *endogenous*, i.e., accomplished through (inter-process communication) primitives from inside the parties engaged in the protocol. It is clear that exogenous composition and coordination lead to simpler, cleaner, and more reusable component code, simply because all composition and coordination concerns are left out. What is perhaps less obvious is that exogenous coordination also leads to reusable, pure coordination code: there is nothing in any incarnation of the connector in Figure 5 that is specific to P or C; it can just as readily engage any producer and consumer processes in any other application.

Obviously, we are not interested in only this example, nor exclusively in connectors that implement exogenous coordination between only two communicating parties. Moreover, the code for any version of the connector in Figure 5, or any other connector, can be written in any programming language: the concepts of exogenous composition, exogenous coordination, and the system design and architecture that they induce constitute what matters, not the implementation language. Focusing on multi-party interaction/coordination protocols reveals that they are composed out of a small set of common recurring concepts. They include synchrony, atomicity, asynchrony, ordering, exclusion, grouping, selection, etc. Compliant with the constraint view of interaction advocated above, these concepts can be expressed as constraints, more directly and elegantly than as compositions of actions in a process algebra or an imperative programming

¹ By this anthropomorphic expression we simply mean that a component does not contain any piece of code that directly contributes to determine the entities that it composes with, or the specific protocol that coordinates its own interactions with them.

language. This observation behooves us to consider the interaction-as-constraint view of concurrency as a foundation for a special language to specify multi-party exogenous interaction/coordination protocols and the connectors that embody them, of which the connector in Figure 5 is but a trivial example. Reo, described in the next section, is a premier example of such a language.

3 An Overview of Reo

Reo [20,11,12,6] is a channel-based exogenous coordination language wherein complex coordinators, called connectors, are compositionally built out of simpler ones. Exogenous coordination imposes a purely local interpretation on each inter-components communication, engaged in as a pure I/O operation on each side, that allows components to communicate anonymously, through the exchange of untargeted passive data. We summarize only the main concepts in Reo here. Further details about Reo and its semantics can be found in the cited references.

Complex connectors in Reo are constructed as a network of primitive binary connectors, called *channels*. Connectors serve to provide the protocol that controls and organizes the communication, synchronization and cooperation among the components/services that they interconnect. Formally, the protocol embodied in a connector is a *relation*, which the connector imposes as a *constraint* on the actions of the communicating parties that it inter-connects.

A channel is a medium of communication that consists of two ends and a constraint on the dataflows observed at those ends. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. Every channel (type) specifies its own particular behavior as constraints on the flow of data through its ends. These constraints relate, for example, the content, the conditions for loss, and/or creation of data that pass through the ends of a channel, as well as the atomicity, exclusion, order, and/or timing of their passage. Reo places no restriction on the behavior of a channel and thus allows an open-ended set of different channel types to be used simultaneously together.

Although all channels used in Reo are user-defined and users can indeed define channels with any complex behavior (expressible in the semantic model) that they wish, a very small set of channels, each with very simple behavior, suffices to construct useful Reo connectors with significantly complex behavior. Figure 6 shows a common set of primitive channels often used to build Reo connectors.

A **Sync** channel has a source and a sink end and no buffer. It accepts a data item through its source end iff it can simultaneously (i.e., atomically) dispense it through its sink.

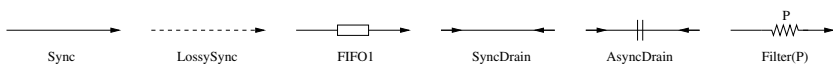


Fig. 6. A typical set of Reo channels

A **LossySync** channel is similar to a synchronous channel except that it always accepts all data items through its source end. This channel transfers a data item if it is possible for the channel to dispense the data item through its sink end; otherwise the channel loses the data item. Observe that the behavior of this channel is fully deterministic; the channel is never free to choose between passing or losing a data item: if it is possible for a data item to be consumed through its sink end, the channel *must* pass the data item exactly as a **Sync**. Thus, the context of (un)availability of a ready consumer at its sink end determines the (context-sensitive) behavior a **LossySync** channel.

A **FIFO1** channel represents an asynchronous channel with a buffer of capacity 1: it can contain at most one data item. In the graphical representation of an empty **FIFO1** channel, no data item is shown in the box (this is the case in Figure 1). If the buffer of a **FIFO1** channel contains a data element d , then d appears inside the box in its graphical representation. When its buffer is empty, a **FIFO1** channel blocks I/O operations on its sink, because it has no data to dispense. It dispenses a data item and allows an I/O operation at its sink to succeed, only when its buffer is full, after which its buffer becomes empty. When its buffer is full, a **FIFO1** channel blocks I/O operations on its source, because it has no more capacity to accept the incoming data. It accepts a data item and allows an I/O operation at its source to succeed, only when its buffer is empty, after which its buffer becomes full.

More exotic channels are also permitted in Reo, for instance, synchronous and asynchronous *drains*. Each of these channels has two source ends and no sink end. No data value can be obtained from a drain channel because it has no sink end. Consequently, all data accepted by a drain channel are lost. **SyncDrain** is a synchronous drain that can accept a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. **AsyncDrain** is an asynchronous drain that accepts data items through its source ends and loses them exclusively one at a time, but never simultaneously.

For a *filter channel*, or **Filter(P)**, its pattern $P \subseteq Data$ specifies the type of data items that can be transmitted through the channel. This channel accepts a value $d \in P$ through its source end iff it can simultaneously dispense d through its sink end, exactly as if it were a **Sync** channel; it always accepts all data items $d \notin P$ through its source end and loses them immediately.

Synchronous and asynchronous *Spouts* are the duals of their respective drain channels, as each has two sink ends through which it produces nondeterministic data items. Further discussion of these and other primitive channels is beyond the scope of this paper.

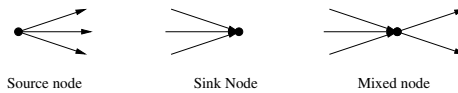


Fig. 7. Reo nodes

Complex connectors are constructed by composing simpler ones via the *join* and *hide* operations. Channels are joined together in *nodes*, each of which consists of a set of channel ends. A Reo node is a logical place where channel ends coincide and coordinate their dataflows as prescribed by its *node type*. Figure 7 shows the three possible node types in Reo. A node is either *source*, *sink*, or *mixed*, depending on whether all channel ends that coincide on that node are source ends, sink ends, or a combination of the two. Reo fixes the semantics of (i.e., the constraints on the dataflow through) Reo nodes, as described below. The *hide* operation is used to hide the internal topology of a Reo connector. A hidden nodes can no longer be accessed or observed from outside.

The source and sink nodes of a connector are collectively called its *boundary nodes*. Boundary nodes define the interface of a connector. Processes (or components, actors, agents, etc.) connect to the boundary nodes of a connector and interact anonymously with each other through this interface. Connecting a process to a (source or sink) node of a connector consists of the identification of one of the (respectively, output or input) ports of the component with that node. At most one process can be connected to a (source or sink) node at a time. Processes interact by performing their blocking I/O operations on their own local ports, which trigger dataflow through their respectively identified nodes of the connector(s): the *get* and *put* operations mentioned in the description of the components in Figure 5 trigger *write* and *take* operations of Reo on the channel ends of their respective nodes.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a synchronous replicator.

A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic merger.

A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot connect to, take from, or write to mixed nodes.

Because a node has no buffer, data cannot be stored in a node. Specifically, a mixed node cannot take a data item out of one of its coincident sink channel ends, unless it can atomically replicate and write it into all of its coincident source channel ends. Hence, nodes instigate the propagation of synchrony and exclusion constraints on dataflow throughout a connector. Deriving the semantics of a Reo connector amounts to resolving the composition of the constraints of its constituent channels and nodes [43]. This is not a trivial task. In the sequel, we present examples of Reo connectors that illustrate how non-trivial dataflow behavior emerges from composing simple channels using Reo nodes. The local

constraints of individual channels propagate through (the synchronous regions of) a connector to its boundary nodes. This propagation also induces a certain context-awareness in connectors. See [41] for a detailed discussion of this.

Reo has been used for composition of Web services [65,77,21], modeling and analysis of long-running transactions in service-oriented systems [69], coordination of multi-agent systems [13], performance analysis of coordinated compositions [23,16,17,86,87], modeling of business processes and verification of their compliance [98,68,19], and modeling of coordination in biological systems [40].

Reo offers a number of operations to reconfigure and change the topology of a connector at run-time: operations that enable the dynamic creation of channels, splitting and joining of nodes, hiding internal nodes. The hiding of internal nodes allows to permanently fix the topology of a connector, such that only its boundary nodes are visible and available. The resulting connector can then be viewed as a new primitive connector, or primitive for short, since its internal structure is hidden and its behavior is fixed.

4 Examples

Recall our alternating producers and consumer example of Section 1. We revise the code for the Green and Red producers to make them suitable for exogenous coordination (which, in fact, makes them simpler). Similar to the producer P in Figure 5, this code now consists of an infinite loop, in each iteration of which the producer computes a new value and writes it to its output port. Analogously, we revise the consumer code, fashioning it after the consumer C in Figure 5. Figure 8 shows this code.

<pre>Consumer: 1 while (true) { 2 sleep(4000); 3 get(input, displayText); 4 print(displayText); 5 }</pre>	<pre>Green Producer: 6 while (true) { 7 sleep(5000); 8 greenText = ...; 9 put(output, greenText); 10 }</pre>	<pre>Red Producer: 11 while (true) 12 sleep(3000); 13 redText = ...; 14 put(output, redText); 15 }</pre>
---	--	--

Fig. 8. Generic reusable producers and consumer

In the remainder of this section, we present a number of protocols to implement different versions of the alternating producers and consumer example of Section 1, using the producers and consumer processes in Figure 8. These examples serve three purposes. First, they show a flavor of programming of pure interaction coordination protocols as Reo circuits. Second, they present a number of generically useful circuits that can serve as connectors in many other applications, or as sub-circuits in the circuits for construction of many other protocols. Third, they illustrate the utility of exogenous coordination by showing how trivial it is to change the protocol of an application, without altering any of the processes involved.

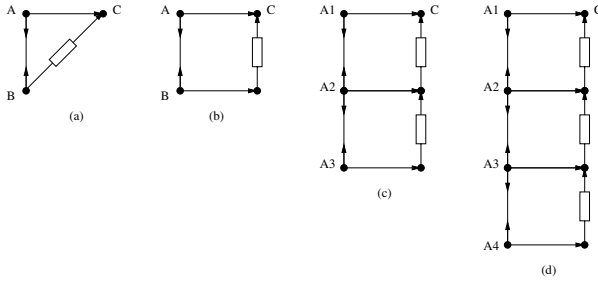


Fig. 9. Reo circuits for Alternators

4.1 Alternator

The connector shown in Figure 9(a) is an *alternator* that imposes an ordering on the flow of the data from its input nodes A and B to its output node C . The **SyncDrain** enforces that data flow through A and B only synchronously (i.e., atomically). The empty buffer of the the **FIFO1** channel together with the **SyncDrain** guarantee that the data item obtained from A is delivered to C while the data item obtained from B is stored in the **FIFO1** buffer. After this, the buffer of the **FIFO1** is full and data cannot flow in through either A or B , but C can dispense the data stored in the **FIFO1** buffer, which makes it empty again. Thus, subsequent take operations at C obtain the data items written to A, B, A, B, \dots , etc.

The connector in Figure 9(b) has an extra **Sync** channel between node B and the **FIFO1** channel, compared to the one in Figure 9(a). It is trivial to see that these two connectors have the exact same behavior. However, the structure of the connector in Figure 9(b) allows us to generalize its alternating behavior to any number of producers, simply by replicating it and “juxtaposing” the top and the bottom **Sync** channels of the resulting copies, as seen in Figure 9(c) and Figure 9(d).

The two **SyncDrain** channels in the connector shown in Figure 9(c) require data to flow through $A1, A2$, and $A3$ only simultaneously (i.e., atomically). The empty buffers of the **FIFO1** channels, together with these **SyncDrain** channels guarantee that the data item obtained from $A1$ is delivered to C while the data items obtained from $A2$ and $A3$ are stored in the buffers of their respective **FIFO1** channels. Subsequently, as long as the buffer of at least one of the **FIFO1** channels remains full, no data can flow through any of the nodes $A1, A2$, and $A3$, but C can dispense the data stored in the buffers of the **FIFO1** channels, with their order preserved. Thus, the first 3 take operations on C deliver the data items obtained through $A1, A2$, and $A3$, in that order. At this point, all **FIFO1** buffers become empty and the next round of input becomes possible.

The connector in Figure 9(d) is obtained by replicating the one in Figure 9(b) 3 times. Following the reasoning for the connector in Figure 9(c), it is easy to see that the connector in Figure 9(d) delivers the data items obtained from $A1, A2, A3$, and $A4$ through C , in that order.

A version of our alternating producers and consumer example of Section 1 can now be composed by attaching the output port of the revised Green producer in Figure 8 to node A , the output port of the revised Red producer in Figure 8 to node B , and the input port of the consumer in Figure 8 to node C of the Reo circuit in Figure 9(a).

A closer look shows, however, that the behavior of this version of our example is *not* exactly the same as that of the one in Figures 3 and 4. As explained above, the Reo circuit in Figure 9(a) requires the availability of a pair of values on A (from the Green producer) and B (from the Red producer) before it allows the consumer to obtain them, first from A and then from B . Thus, if the Green producer and the consumer are both ready to communicate, they still have to wait for the Red producer to also attempt to communicate, before they can exchange data. The versions in Figures 3 and 4 allow the Green producer and the consumer to go ahead, regardless of the state of the Red producer. Our original specification of this example in Section 1 was abstract enough to allow both alternatives. A further refinement of this specification may indeed prefer one and disallow the other. If the behavior of the connector in Figure 9(a) is *not* what we want, we need to construct a different Reo circuit to impose the same behavior as in Figures 3 and 4. This is precisely what we describe below.

4.2 Sequencer

Figure 10(a) shows an implementation of a sequencer by composing five Sync channels and four FIFO1 channels together. The first (leftmost) FIFO1 channel is initialized to have a data item in its buffer, as indicated by the presence of the symbol e in the box representing its buffer cell. The connector provides only the four nodes A, B, C and D for other entities (connectors or component instances) to take from. The take operation on nodes A, B, C and D can succeed only in the strict left-to-right order. This connector implements a generic sequencing protocol: we can parameterize this connector to have as many nodes as we want simply by inserting more (or fewer) Sync and FIFO1 channel pairs, as required.

Figure 10(b) shows a simple example of the utility of the sequencer. The connector in this figure consists of a two-node sequencer, plus a SyncDrain and two Sync channels connecting each of the nodes of the sequencer to the nodes A and C , and B and C , respectively. Similar to the circuit in Figure 9(a), this connector imposes an order on the flow of the data items written to A and B , through C : the sequence of data items obtained by successive take operations

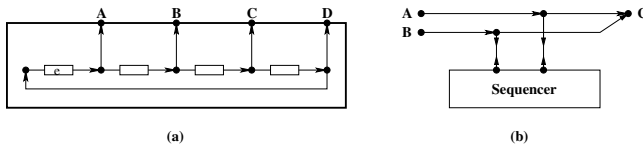


Fig. 10. Sequencer

on C consists of the first data item written to A , followed by the first data item written to B , followed by the second data item written to A , followed by the second data item written to B , and so on. However, there is a subtle difference between the behavior of the two circuits in Figures 9(a) and 10(b). The alternator in Figure 9(a) delays the transfer of a data item from A to C until a data item is also available at B . The circuit in Figure 10(b) transfers from A to C as soon as these nodes can satisfy their respective operations, regardless of the availability of data on B .

We can obtain a new version of our alternating producers and consumer example by attaching the output port of the Green producer in Figure 8 to node A , the output port of the Red producer in Figure 8 to node B , and the input port of the consumer in Figure 8 to node C . The behavior of this version of our application is now the same as the programs in Figure 4 and in Figure 1 (after replacing its producers with the ones in Figure 2). The circuit in Figure 10(b) embodies the same protocol that is implicit in Figure 4.

A characteristic of this protocol is that it “slows down” each producer, as necessary, by delaying the success of its data production until the consumer is ready to accept its data. Our original problem statement in Section 1 does not explicitly specify whether or not this is a required or permissible behavior. While this may be desirable in some applications, slowing down the producers to match the processing speed of the consumer may have serious drawbacks in other applications, e.g., if these processes involve time-sensitive data or operations. Perhaps what we want is to bind our producers and consumer by a protocol that decouples them such as to allow each process to proceed at its own pace. We proceed, below, to present a number of protocols that we then compose to construct a Reo circuit for such a protocol.

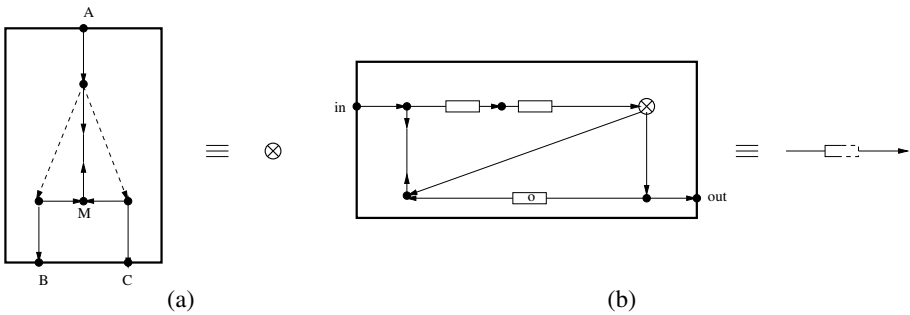


Fig. 11. An exclusive router and a ShiftLossyFIFO1

4.3 Exclusive Router

The connector shown in Figure 11(a) is a binary *exclusive router*: it routes data from A to either B or C (but not both). This connector can accept data only if there is a write operation at the source node A , and there is at least one taker at the sink node B or C . If both B and C can dispense data, the choice

of routing to B or C follows from the non-deterministic decision by the mixed node M : it can accept data only from one of its sink ends, excluding the flow of data through the other, which forces the latter's respective **LossySync** to lose the data it obtains from A , while the other **LossySync** passes its data as if it were a **Sync**.

By connecting the source node of a binary exclusive router to one of the sink nodes of another binary exclusive router we obtain a ternary exclusive router. This is possible in Reo because synchrony and exclusion constraints propagate through its nodes. More generally, an n -ary exclusive router (with a single source and n sink ends) can be composed out of $n - 1$ binary exclusive routers. Because the exclusive routers are so commonly useful, we use a graphical short-hand to represent them in circuits. The crossed circle shown on the right-hand side of Figure 11(a) is the symbol that we use to represent a generic n -ary exclusive router.

4.4 Shift-Lossy FIFO1

Figure 11(b) shows a Reo circuit for a connector that behaves as a lossy **FIFO1** channel with a shift loss-policy. This channel is called shift-lossy **FIFO1** (**ShiftLossyFIFO1**). This connector is composed of an exclusive router (shown in Figure 11(a)), an initially full **FIFO1** channel, two initially empty **FIFO1** channels, and four **Sync** channels. Intuitively, it behaves as a normal **FIFO1** channel, except that if its buffer is full then the arrival of a new data item deletes the existing data item in its buffer, making room for the new arrival. As such, this channel implements a “shift loss-policy” losing the older contents in its buffer in favor of the newer arrivals. This is in contrast to the behavior of an overflow-lossy **FIFO1** channel, whose “overflow loss-policy” loses the new arrivals when its buffer is full. See [31] for a more formal treatment of the semantics of this connector.

The **ShiftLossyFIFO1** circuit in Figure 11(b) is indeed so frequently useful as a connector in construction of more complex circuits, that it makes sense to have a special graphical symbol to designate it as a short-hand. The symbol shown on the right-hand side of Figure 11(b) is the what we use to represent this circuit, and also take the liberty to refer to it as a **ShiftLossyFIFO1** “channel”. This symbol is intentionally similar to that of a regular **FIFO1** channel, because the behavior of this circuit closely resembles that of a regular **FIFO1** channel. The dashed sink-side half of the box representing the buffer of this channel suggests that it loses the older values to make room for new arrivals, i.e., it shifts to lose.

4.5 Decoupled Alternating Producers and Consumer

Figure 12(a) shows how the **ShiftLossyFIFO1** circuit of Figure 11(b) can be used to construct a version of the example in Figure 5, where the producer and the consumer are partially decoupled from one another. Whenever, as initially is the case, the **ShiftLossyFIFO1** buffer is empty, the consumer has no choice but to wait for the producer to place a value into this buffer. However, the producer

never has to wait for the consumer: it can work at its own pace and write to the connector whenever it wishes. Every write by the producer replaces the current contents of the `ShiftLossyFIFO1` buffer. A subsequent take by the consumer obtains the current value out of `ShiftLossyFIFO1` buffer and makes it empty. The producer never has to wait for the consumer, but if the consumer is faster than the producer, it has to wait for the next data item to arrive. It is instructive to compare the behavior of this system with that of a single `LossySync` channel connecting a producer and a consumer: the two are not exactly the same.

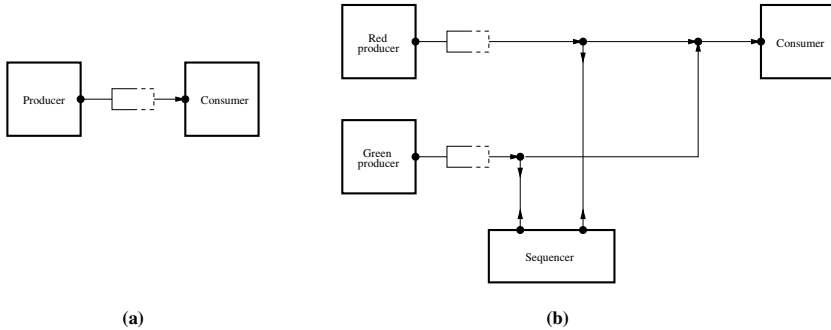


Fig. 12. Decoupled producers and consumer

The connector in Figure 12(b) is a small variation of the Reo circuit in Figure 10(b), with two instances of the `ShiftLossyFIFO1` circuit of Figure 11(b) spliced in. In this version of our alternating producers and consumer, these three processes are partially decoupled: each producer runs at its own pace, never having to wait for any of the other two processes. Every take by the consumer, always obtains and empties the latest value produced by its respective producer. If the consumer runs slower than a producer, the excess data that they produce is lost in the producer’s respective `ShiftLossyFIFO1`, which allows the consumer to effectively “sample” the data generated by this producer. If the consumer runs faster than a producer, it will block on its respective empty `ShiftLossyFIFO1` until a new value becomes available.

4.6 Dataflow Variable

The Reo circuit in Figure 13 implements the behavior of a dataflow variable. It uses two instances of the `ShiftLossyFIFO1` connector shown Figure 11(b), to build a connector with a single input and a single output nodes. Initially, the buffers of its `ShiftLossyFIFO1` channels are empty, so an initial take on its output node suspends for data. Regardless of the status of its buffers, or whether or not data can be dispensed through its output node, every write to its input node always succeeds and resets both of its buffers to contain the new data item. Every time a value is dispensed through its output node, a copy of this value is

“cycled back” into its left `ShiftLossyFIFO1` channel. This circuit “remembers” the last value it obtains through its input node, and dispenses copies of this value through its output node as frequently as necessary: i.e., it can be used as a dataflow variable.

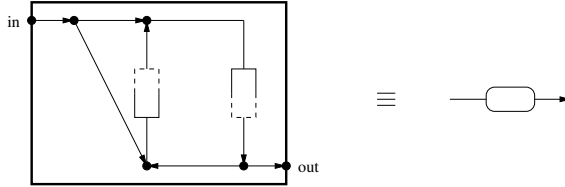


Fig. 13. Dataflow variable

The variable circuit in Figure 13 is also very frequently useful as a connector in construction of more complex circuits. Therefore, it makes sense to have a short-hand graphical symbol to designate it with as well. The symbol shown on the right-hand side of Figure 13 is the what we use to represent this circuit, and also take the liberty to refer to it as a `Variable` “channel”, or just a “variable” for short. This symbol is intentionally similar to that of a regular `FIFO1` channel, because the behavior of this circuit closely resembles that of a regular `FIFO1` channel. We use a rounded box to represent its buffer: the rounded box hints at the recycling behavior of the variable circuit, which implements its remembering of the last data item that it obtained or dispensed.

4.7 Fully Decoupled Alternating Producers and Consumer

Figure 14(a) shows how the variable circuit of Figure 13 can be used to construct a version of the example in Figure 5, where the producer and the consumer are fully decoupled from one another. Initially, the variable contains no value, and therefore, the consumer has no choice but to wait for the producer to place its first value into the variable. After that, neither the producer, nor the consumer ever has to wait for the other one. Each can work at its own pace and write to or take from the connector. Every write by the producer replaces the current contents of the variable, and every take by the consumer obtains a copy of the current value of the variable, which always contains the most recent value produced.

The connector in Figure 14(b) is a small variation of the `Reo` circuit in Figure 10(b), with two instances of the variable circuit of Figure 13 spliced in. In this version of our alternating producers and consumer, these three processes are fully decoupled: each can produce and consume at its own pace, never having to wait for any of the other two. Every take by the consumer, always obtains the latest value produced by its respective producer. If the consumer runs slower than a producer, the excess data is lost in the producer’s respective variable,

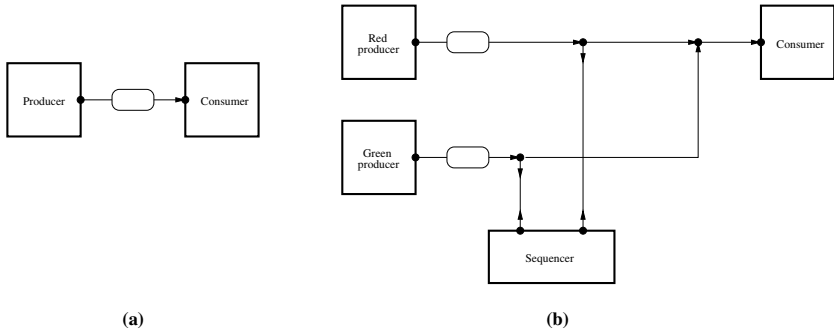


Fig. 14. Fully decoupled producers and consumer

and the consumer will effectively “sample” the data generated by this producer. If the consumer runs faster than a producer, it will read (some of) the values of this producer multiple times.

4.8 Flexibility

Figures 9(a), 10(b), 12(b), and 14(b) show four different connectors, each imposing a different protocols for the coordination of two alternating producers and a consumer. The exact same producers and consumer processes can be combined with any of these circuits to yield different applications. It is instructive to compare the ease with which this is accomplished in our interaction-centric world, with the effort involved in modifying the action-centric incarnations of this same example in Figures 3 and 4, which correspond to the protocol of the circuit in Figure 10(b), in order to achieve the behavior induced by the circuit in Figure 9(a), 12(b), or 14(b).

For the sake of completeness, the behavior of the protocol in Figures 1 corresponds to the behavior of the connector in Figure 15. Just as in the case of the program in Figures 1, this connector allows the producers at nodes *A* and *B* alternate and over-write each other in buffer of the `ShiftLossyFIFO1`. The consumer at *C* can obtain only the latest value produced by either of the producers.

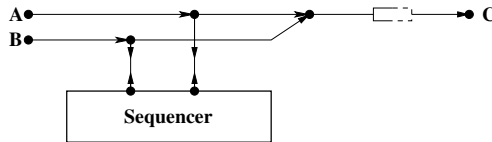


Fig. 15. Alternating and over-writing

The Reo connector binding a number of distributed processes, such as Web services, can even be “hot-swapped” while the application runs, without the knowledge or the involvement of the engaged processes. A prototype platform to demonstrate this capability is available at [3].

5 Semantics

Reo allows arbitrary user-defined channels as primitives; arbitrary mix of synchrony and asynchrony; and relational constraints between input and output. This makes Reo more expressive than, e.g., dataflow models, Kahn networks, synchronous languages, stream processing languages, workflow models, and Petri nets. On the other hand, it makes the semantics of Reo quite non-trivial.

Various models for the formal semantics of Reo have been developed, each to serve some specific purposes. In the rest of this section, we briefly describe the main ones.

5.1 Timed Data Streams

The first formal semantics of Reo was formulated based on the coalgebraic model of stream calculus [95,94,96]. In this semantics, the behavior of every connector (channel or more complex circuit) and every component is given as a (maximal) relation on a set of *timed-data-streams* [24]. This yields an expressive compositional semantics for Reo where coinduction is the main definition and proof principle to reason about properties involving both data and time streams. The timed-data-stream model serves as the reference semantics for Reo.

Table 1. TDS Semantics of Reo primitives

Sync	$\langle \alpha, a \rangle \text{Sync} \langle \beta, b \rangle \equiv \alpha = \beta \wedge a = b$
LossySync	$\langle \alpha, a \rangle \text{LossySync} \langle \beta, b \rangle \equiv \begin{cases} \beta(0) = \alpha(0) \wedge \langle \alpha', a' \rangle \text{LossySync} \langle \beta', b' \rangle & \text{if } a(0) = b(0) \\ \langle \alpha', a' \rangle \text{LossySync} \langle \beta, b \rangle & \text{if } a(0) < b(0) \end{cases}$
empty FIFO1	$\langle \alpha, a \rangle \text{FIFO1} \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b < a'$
FIFO1 initialized with x	$\langle \alpha, a \rangle \text{FIFO1}(x) \langle \beta, b \rangle \equiv \alpha = x.\beta \wedge b < a < b'$
SyncDrain	$\langle \alpha, a \rangle \text{SyncDrain} \langle \beta, b \rangle \equiv a = b$
AsyncDrain	$\langle \alpha, a \rangle \text{SyncDrain} \langle \beta, b \rangle \equiv a \neq b$
Filter(P)	$\langle \alpha, a \rangle \text{Filter}(P) \langle \beta, b \rangle \equiv \begin{cases} \beta(0) = \alpha(0) \wedge b(0) = a(0) \wedge \langle \alpha', a' \rangle \text{Filter}(P) \langle \beta', b' \rangle & \text{if } \alpha(0) \ni P \\ \langle \alpha', a' \rangle \text{Filter}(P) \langle \beta, b \rangle & \text{otherwise} \end{cases}$
Merge	$\text{Mrg}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv \begin{cases} \alpha(0) = \gamma(0) \wedge a(0) = c(0) \wedge \text{Mrg}(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) < b(0) \\ \beta(0) = \gamma(0) \wedge b(0) = c(0) \wedge \text{Mrg}(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) > b(0) \end{cases}$
Replicate	$\text{Rpl}(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \alpha = \beta \wedge \alpha = \gamma \wedge a = b \wedge a = c$

A *stream* over a set X is an infinite sequence of elements $x \in X$. The set of *data streams* DS consists of all streams over an uninterpreted set of *Data* items. A *time stream* is a monotonically increasing sequence of non-negative real numbers. The set TS represents all time streams². A *Timed Data Stream* (TDS) is a twin pair of streams $\langle \alpha, a \rangle$ in $TDS = DS \times TS$ consisting of a data stream $\alpha \in DS$ and a time stream $a \in TS$, with the interpretation that for all $i \geq 0$, the observation of the data item $\alpha(i)$ occurs at the time moment $a(i)$. We use a' to represent the tail of a stream a , i.e., the stream obtained after removing the first element of a ; and $x.a$ to represent the stream whose first element is x and whose tail is a .

Table 1 shows the TDS semantics of the primitive channels in Figure 6, as well as that of the merge and replication behavior inherent in Reo nodes. The semantics for every primitive is expressed as a binary (in the case of channels) or ternary (for the merger and the replicator) relation on timed-data-streams that represent the observations at their respective source and sink ends. We can use relational composition to combine the semantics of these primitives to obtain the semantics of more complex connectors. For instance, by composing the relation that defines a binary merger in Table 1 with that of another, we can obtain the semantics for a ternary merger. Thus, the semantics of an m -ary sink node in Reo can be obtained as the composition of $m - 1$ binary mergers. Analogously, the semantics of an n -ary source node in Reo can be obtained as the composition of $n - 1$ binary replicators. The semantics of a Reo mixed node with m coincident sink and n coincident source channel ends is obtained as the relational composition of $m - 1$ binary mergers and $n - 1$ binary replicators.

The semantics of a Reo circuit is the relational composition of the relations that represent the semantics of its constituents (including the merge and replication inherent in its nodes). This compositional construction for instance, yields

$$XRout(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \begin{cases} \alpha(0) = \gamma(0) \wedge a(0) = c(0) \wedge XRout(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) < b(0) \\ \beta(0) = \gamma(0) \wedge b(0) = c(0) \wedge XRout(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) > b(0) \end{cases}$$

as the semantics of the circuit in Figure 11(a).

5.2 Constraint Automata

Constraint automata provide an operational model for the semantics of Reo circuits [31]. The states of an automaton represent the configurations of its corresponding circuit (e.g., the contents of the FIFO channels), while the transitions encode its maximally-parallel stepwise behavior. The transitions are labeled with the maximal sets of nodes on which dataflow occurs simultaneously, and a data constraint (i.e., boolean condition for the observed data values). For example, Figure 16 shows the constraint automata semantics for some of the common Reo primitives.

² The real numbers that appear in a time stream must also satisfy an additional technical condition to prevent Zeno's paradox, but for simplicity, we ignore this condition here.

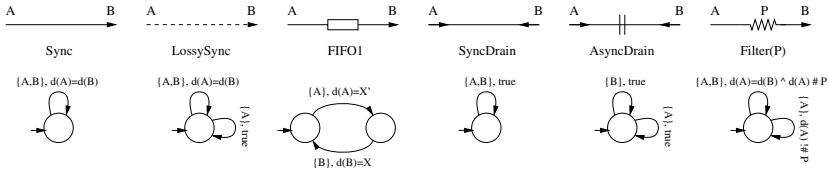


Fig. 16. Constraint automata of some typical Reo Channels

The constraint automaton for the **Sync** channel consists of a single state. It has only a single transition, labeled by the pair of *synchronization constraint*, and *data constraint*. The synchronization constraint $\{A, B\}$ states that this transition is possible iff both nodes A and B can *fire* synchronously (i.e., atomically), allowing their respective pending I/O operations to succeed. The data constraint $d(A) = d(B)$ states that this transition is possible iff the data observed at node A is identical to the data observed at node B . Because these two nodes are respectively the source and the sink nodes (of the **Sync** channel), this data constraint requires a transfer of data from A to B .

The constraint automaton for the **LossySync** channel in fact expresses the semantics of a *nondeterministic LossySync* channel, *not* that of our *context sensitive LossySync* described in Section 3. The difference is significant, but it is not important for our purposes in this paper.³ This automaton has a single state and two transitions. One of these transitions is identical to that of the **Sync** channel, modeling its identical behavior. The other, labeled by $\{A\}, true$ simply states that the automaton can make this transition iff A can fire by itself and imposes no constraint of the data of A : this data is lost.

The constraint automaton for the **FIFO1** channel has two states, representing its empty (initial) and full states. To simplify our presentation, we consider a variant of constraint automata that allow states to have local memory variables. The label $\{A\}, d(A) = X'$ of the transition that takes the automaton from its empty to its full state allows it to make this transition iff node A can fire by itself, and the *new value* of the memory variable X in the target state (identified by X' in the data constraint) is the same as the data value observed on node A : the value obtained from the source node A gets assigned to the X variable of the target state to satisfy this constraint. The label $\{B\}, d(B) = X$ of the transition that takes the automaton from its full to its empty state allows it to make this transition iff node B can fire by itself, and the value of the memory variable X in the source state (identified by X in the data constraint) is the same as the data value observed on node B : the value of the X variable of the source state is dispensed through the sink node B to satisfy this data constraint.

³ In fact, constraint automata do not have the expressiveness required to directly represent context sensitivity. Other more expressive semantic models, including more sophisticated automata models, have been devised for this purpose [35,44]. A recent work shows that, although constraint automata cannot directly represent context sensitivity, it is possible to *encode* context sensitivity using constraint automata as well [56,70].

The constraint automaton for the **SyncDrain** channel has a single state and a single transition, whose constraints require its ends to fire synchronously ($\{A, B\}$), but imposes no constraints (*true*) on their data. Because these are both source ends, their data are simply lost.

The constraint automaton for the **AsyncDrain** channel has a single state and two transitions, each of which allow it to fire and lose the data obtained through one of its ends (but never both synchronously).

The constraint automaton for the **Filter(P)** channel has a single state and two transitions. If source node *A* can fire and its data value does not match the filter pattern *P*, then the data value of *A* is simply lost. If the data value available on the source node *A* matches the filter pattern *P*, then the only possible transition is one similar to that of the **Sync** channel, by which the data value of *A* is transferred to the sink node *B*.

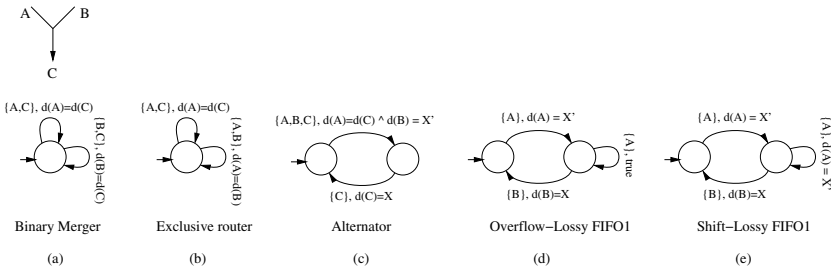


Fig. 17. Constraint automata of a binary merger and some example connectors

The semantics of a Reo circuit is derived by composing the constraint automata of its constituents, through a special form of synchronized product of automata, which automatically accommodates the replication semantics of Reo nodes [31]. The nondeterministic *n*-ary merge semantics inherent in Reo nodes needs to be made explicit as a (product) composition of *n* – 1 nondeterministic binary merge primitives. Figure 17(a) shows the constraint automaton for a nondeterministic binary merge primitive.

Figure 17(b) shows the constraint automaton representing the semantics of the exclusive router Reo circuit of Figure 11(a), which is obtained as the product of the constraint automata of its constituents: 5 **Sync** channels, 2 **LossySync** channels, a **SyncDrain** channel, and a merger.

Figure 17(c) shows the constraint automaton representing the semantics of the alternator circuit of Figure 9(a), obtained as the product of the constraint automata of its constituent **Sync** channel, **SyncDrain** channel, **FIFO1** channel, and merger.

Figure 17(d) shows the constraint automaton representing the semantics of an *overflow lossy* connector, which can be easily composed by connecting the sink end of a **LossySync** to the source end of a **FIFO1**. Although this *is* the semantics that must be obtained, the product of simple constraint automata in Figure 16 does *not* yield this automaton. This automaton can be obtained

using more sophisticated variants of constraint automata [35,44], or an encoding technique [56] which can handle context sensitivity.

Figure 17(e) shows the constraint automaton representing the semantics of the `ShiftLossyFIFO1` circuit of Figure 11(b), which is obtained as the product of the constraint automata of its constituents.

Constraint automata have been used for the verification of protocols through model-checking [7,62,34,28,61,30,29,48]. Results on equivalence and containment of the languages of constraint automata [31] and failure based equivalences [54] provide opportunities for analysis and optimization of Reo circuits.

A constraint automaton essentially captures all behavior alternatives of a Reo connector. Therefore, it can be used to generate a state-machine implementing the behavior of Reo connectors, in a chosen target language, such as Java or C. The constraint automata semantics of Reo is used to generate executable code for Reo [18].

Variants of the constraint automata model have been devised to capture time-sensitive behavior [14,58,59], probabilistic behavior [26], stochastic behavior [32], context sensitive behavior [35,44,52], fairness [53,36], resource sensitivity [79], and the QoS aspects [80,16,17,87,86] of Reo connectors and composite systems.

5.3 Connector Coloring

The Connector Coloring (CC) model describes the behavior of a Reo circuit in terms of the detailed dataflow behavior of its constituent channels and nodes [41]. The semantics of a Reo circuit is the set of all of its dataflow alternatives. Each such alternative is a consistent composition of the dataflow alternatives of each of its constituent channels and nodes, expressed in terms of (solid and dashed) *colors* that represent the basic flow and no-flow alternatives.

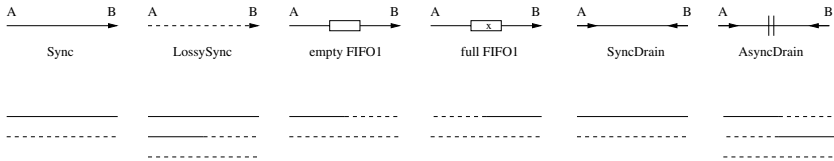


Fig. 18. Connector Coloring semantics of some typical Reo Channels

Figure 18 shows the two-color semantics of some common Reo primitives. The `Sync` channel has two alternative colorings, each representing one possible behavior: either flow on both of its ends (the solid line) or no flow on both ends (the dashed line). The (nondeterministic) `LossySync` has three alternative colorings: it either behaves as the `Sync` channel (the full solid and the full dashed lines), or it allows flow at its source end, with no flow at its sink end (the half-solid-half-dashed line). A `FIFO1` channel has two sets of colorings, one for each of its two states: empty and full. In its empty state, it can allow flow only at its source end (with no flow at its sink), after which it becomes full. In its full

state, it can allow flow only at its sink node (with no flow at its source), which makes it empty. A `SyncDrain` channel has the same coloring as a `Sync` channel: it can allow flow only through both of its ends simultaneously. An `AsyncDrain` allows flow through only one of its ends at a time.

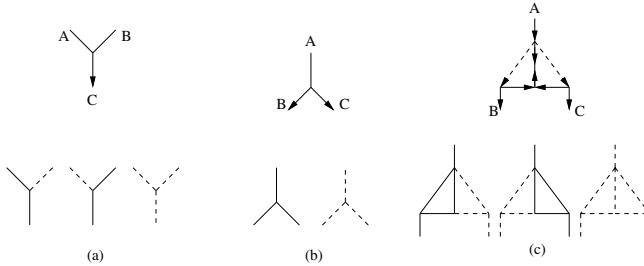


Fig. 19. Connector Coloring semantics for merger, replicator, and exclusive router

To express the semantics of a Reo circuit, the replicator and the merger behavior inherent in Reo nodes must also be explicitly modeled as colorings. Figure 19(a) shows the three alternatives for the behavior of a merger: the merger nondeterministically chooses to allow flow either through its left source and sink, or through its right source and sink, or there is no flow on any of its ends. Figure 19(b) shows the two alternatives for the behavior of a replicator: either there is flow on its source and both sinks, or there is no flow through any of its nodes at all.

The coloring semantics of a Reo circuit can be composed out of the coloring alternatives of its constituents, subject to the obvious requirement that each node in the circuit can either have flow or not, and therefore, the colors of the behavior alternatives of all constituents that coincide on a node must be the same: either dashed or solid. For example, the coloring alternatives of the exclusive router circuit of Figure 11(a) is obtained by matching the alternative colors of its constituent channels, replicators, and merger, as shown in Figure 19(c). As expected, this circuit as a whole allows flow through either its right-hand side, or its left-hand side, exclusively, or there is no flow through the circuit at all.

A more sophisticated model using three colors is necessary to capture the context sensitive behavior of primitives such as the `LossySync` channel. The CC model is primarily used in the implementation of a visualization tool that produces Flash animations depicting the behavior of a connector [44,91,75]. Connector coloring and constraint automata are related [55]. It has been shown that it is possible to encode context sensitive behavior in the two-color CC model as well, using hypothetical extra nodes [56].

Finding a consistent coloring for a circuit amounts to constraint satisfaction. Constraint solving techniques [10,102] have been applied using the CC model to search for a valid global behavior of a given Reo connector [42,43]. In this approach, each connector is considered as a set of constraints, representing the colors of its individual constituents, where valid solutions correspond to a valid

behavior for the current step. Distributed constraint solving techniques can be used to adapt this constraint based approach for distributed environments.

The CC model is at the center of the distributed implementation of Reo [2,91,93] where several engines, each executing a part of the same connector, run on different remote hosts. A distributed protocol based on the CC model guarantees that all engines running the various parts of the connector agree to collectively manifest one of its legitimate behavior alternatives.

5.4 Other Models

Other formalisms have also been used to investigate the various aspects of the semantics of Reo. Plotkin's style of Structural Operational Semantics (SOS) is followed in [89] for the formal semantics of Reo. This semantics was used in a proof-of-concept tool developed in the rewriting logic language of Maude, using the simulation toolkit.

The Tile Model [47] semantics of Reo offers a uniform setting for representing not only the ordinary dataflow execution of Reo connectors, but also their dynamic reconfigurations [15]. An abstraction of the constraint automata is used in [74] to serve as a common semantics for Reo and Petri nets. The application of intuitionistic temporal linear logic (ITLL) as a basis for the semantics of Reo is studied in [38], which also shows the close semantic link between Reo and the zero-safe variant of Petri nets. A comparison of Orc [85,60] and Reo appears in [92], and the authors of [99] compare Reo with ARC and PBRD coordination models.

The semantics of Reo has also been formalized in the Unifying Theories of Programming (UTP) [51]. The UTP approach provides a family of algebraic operators that interpret the composition of Reo connectors more explicitly than in other approaches [81]. This semantic model can be used for proving properties of connectors, such as equivalence and refinement relations between connectors and as a reference document for developing tool support for Reo. The UTP semantics for Reo opens the possibility to integrate reasoning about Reo with reasoning about component specifications/implementations in other languages for which UTP semantics is available. The UTP semantics of Reo has been used for fault-based test case generation [9].

Automatic translation of an automata-based semantics of Reo into its equivalent process algebraic specification is the basis of another input-output conformance testing of protocols specified in Reo [70].

Reo offers operations to dynamically reconfigure the topology of its coordination circuits, thereby changing the coordination protocol of a running application. A semantic model for Reo cognizant of its reconfiguration capability, a logic for reasoning about reconfigurations, together with its model checking algorithm, are presented in [39]. Graph transformation techniques have been used in combination with the connector coloring model to formalize the dynamic reconfiguration semantics of Reo circuits triggered by dataflow [64,63,76,75].

6 Tools

Tool support for Reo consists of a set of Eclipse plug-ins that together comprise the Extensible Coordination Tools (ECT) visual programming environment [3]. The Reo graphical editor supports drag-and-drop graphical composition and editing of Reo circuits. This editor also serves as a bridge to other tools, including animation and code generation plug-ins. The animation plug-in automatically generates a graphical animation of the flow of data in a Reo circuit, which provides an intuitive insight into their behavior through visualization of how they work. This tool maps the colors of the CC semantics to visual representations in the animations, and represents the movement of data through the connector [44,91].

Another graphical editor in ECT supports drag-and-drop construction and editing of constraint automata and its variants. It includes tools to perform product and hiding on constraint automata for their composition. A converter plug-in automatically generates the CA model of a Reo circuit.

Several model checking tools are available for analyzing Reo. The Vereofy model checker, integrated in ECT, is based on constraint automata [7,34,62,27,28,61,30,29,48]. Vereofy supports two input languages: (1) the Reo Scripting Language (RSL) is a textual language for defining Reo circuits, and (2) the Constraint Automata Reactive Module Language (CARML) is a guarded command language for textual specification of constraint automata. Properties of Reo circuits can be specified for verification by Vereofy in a language based on Linear Temporal Logic (LTL), or on a variant of Computation Tree Logic (CTL), called Alternating-time Stream Logic (ASL). Vereofy extends these logics with regular expression constructs to express data constraints. Translation of Reo circuits and constraint automata into RSL and CARML is automatic, and the counter-examples found by Vereofy can automatically be mapped back into the ECT and displayed as Reo circuit animations.

Timed Constraint Automata (TCA) were devised as the operational semantics of timed Reo circuits [14]. A SAT-based bounded model checker exists for verification of a variant of TCA [58,59], although it is not yet fully integrated in ECT. It represents the behavior of a TCA by formulas in propositional logic with linear arithmetic, and uses a SAT solver for their analysis. A tool is available to translate (timed) Reo circuits into models for verification using the Uppaal model checker.

Another means for verification of Reo is made possible by a transformation bridge into the mCRL2 toolset [4,49]. The mCRL2 verifier relies on the parameterized boolean equation system (PBES) solver to encode model checking problems, such as verifying first-order modal-calculus formulas on linear process specifications. An automated tool integrated in ECT translates Reo models into mCRL2 and provides a bridge to its tool set. This translation and its application for the analysis of workflows modeled in Reo are discussed in [67,72,71]. Through mCRL2, it is possible to verify the behavior of timed Reo circuits, or Reo circuits with more elaborate data-dependent behavior than Vereofy supports. The resulting labeled transformation systems can also be used for analysis by a number of tools in the CADP tool set [1].

A CA code generator plug-in produces executable Java code from a constraint automaton as a single sequential thread. A C/C++ code generator is under development. In this setting, components communicate via put and get operations on so-called *SyncPoints* that implement the semantics of a constraint automaton port, using common concurrency primitives. The tool also supports loading constraint automata descriptions at runtime, useful for deploying Reo coordinators in Java application servers, e.g., Tomcat, for applications such as mashup execution [66,78].

A distributed implementation of Reo exists [2] as a middleware in the actor-based language Scala [90], which generates Java source code. A preliminary integration of this distributed platform into ECT provides the basic functionality for distributed deployment through extensions of the Reo graphical editor [91].

A set of ECT plug-in tools are under development to support coordination and composition of Web Services using Reo. ECT plug-ins are available for automatic conversion of coordination and concurrency models expressed as UML sequence diagrams [21,22], UML activity diagrams, BPMN diagrams [19], and BPEL source code into Reo circuits [37].

Tools are integrated in ECT for automatic generation of Quantified Intentional Constraint Automata (QIA) from Reo circuits annotated with QoS properties, and subsequent automatic translation of the resulting QIA to Markov Chain models [16,17,87,86]. A bridge to Prism [5] allows further analysis of the resulting Markov chains [23]. Of course, using Markov chains for the analysis of the QoS properties of a Reo circuit (and its environment) is possible only when the stochastic variables representing those QoS properties can be modeled by exponential distributions. The QIA, however, remain oblivious to the (distribution) types of stochastic variables. A discrete event simulation engine integrated in ECT supports a wide variety of more general distributions for the analysis of the QoS properties of Reo circuits [57,100].

Based on algebraic graph transformations, a reconfiguration engine is available as an ECT plug-in that supports dynamic reconfiguration of distributed Reo circuits triggered by dataflow [18,63,75]. It currently works with the Reo animation engine in ECT, and will be integrated in the distributed implementation of Reo.

7 Concluding Remarks

Action and interaction offer dual perspectives on concurrency. Execution of actions involving shared resources by independent processes that run concurrently, induces pairings of those actions, along with an ordering of those pairs, that we commonly refer to as interaction. Dually, interaction can be seen as an external relation that constrains the pairings of the actions of its engaged processes and their ordering. The traditional action-centric models of concurrency generally make interaction protocols intangible by-products, implied by nebulous specifications scattered throughout the bodies of their engaged processes. Specification, manipulation, and analysis of such protocols are possible only indirectly, through specification, manipulation, and analysis of those scattered actions, which is often made even more difficult by the entanglement of the data-dependent control

flow that surrounds those actions. The most challenging aspect of a concurrent system is *what* its interaction protocol does. In contrast to the *how* which an imperative programming language specifies, declarative programming, e.g., in functional and constraint languages, makes it easier to directly specify, manipulate, and analyze the properties of *what* a program does, because *what* is precisely what they express. Analogously, in an interaction-centric model of concurrency, interaction protocols become tangible first-class constructs that exist explicitly as (declarative) constraints outside and independent of the processes that they engage. Specification of interaction protocols as declarative constraints makes them easier to manipulate and analyze directly, and makes it possible to compose interaction protocols and reuse them.

The coordination language Reo is a premier example of a formalism that embodies an interaction-centric model of concurrency. We used examples of Reo circuits to illustrate the flavor programming pure interaction protocols. Expressed as explicit declarative constraints, protocols espouse exogenous coordination. Our examples showed the utility of exogenous coordination in yielding loosely-coupled flexible systems whose components and protocols can be easily modified, even at run time. We described a set of prototype support tools developed as plug-ins to provide a visual programming environment within the framework of Eclipse, and presented an overview of the formal foundations of the work behind these tools.

A dragon lives forever, but not so little boys. Nevertheless, the ecology of today's society has left no secluded cave for our Puff to sadly slip into. The protocols that our magic dragon manifests in its wake as it frolics through the lines of code of concurrent applications will likely touch many aspects of the daily life of every adult Jackie Paper. We have grown to know our magic dragon well through the intimacy of the childhood games we played with it. Scaled up versions of those games have become integral to the proper functioning of our lives as grownups. Wish as we may to make way for other toys, we cannot abandon this magic dragon any more. We need to develop concise languages to directly communicate with our dragon in concrete terms of a structured dialog that explicitly conveys the constraints of acceptable behavior in the context of our requirements. Reo is a particular dialect of one such language.

References

1. 7CADP home page, <http://www.inrialpes.fr/vasy/cadp/>
2. Distributed Reo,
<http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Redrum/BigPicture>
3. Extensible Coordination Tools home page,
<http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools>
4. mCRL2 home page, <http://www.mcrl2.org>
5. Prism, <http://www.prismmodelchecker.org>
6. Reo home page, <http://reo.project.cwi.nl>
7. Vereofy home page, <http://www.vereofy.de/>

8. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press (1986)
9. Aichernig, B.K., Arbab, F., Astefanoaei, L., de Boer, F.S., Meng, S., Rutten, J.J.M.M.: Fault-based test case generation for component connectors. In: Chin, W.-N., Qin, S. (eds.) *TASE*, pp. 147–154. IEEE Computer Society (2009)
10. Apt, K.: *Principles of Constraint Programming*. Cambridge University Press, Cambridge (2003)
11. Arbab, F.: Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.* 14(3), 329–366 (2004)
12. Arbab, F.: Abstract Behavior Types: a foundation model for components and their composition. *Sci. Comput. Program.* 55(1-3), 3–52 (2005)
13. Arbab, F., Aștefănoaei, L., de Boer, F.S., Dastani, M.M., Meyer, J.-J., Tinnermeier, N.: Reo Connectors as Coordination Artifacts in 2APL Systems. In: Bui, T.D., Ho, T.V., Ha, Q.T. (eds.) *PRIMA 2008. LNCS (LNAI)*, vol. 5357, pp. 42–53. Springer, Heidelberg (2008)
14. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M.: Models and temporal logical specifications for timed component connectors. *Software and System Modeling* 6(1), 59–82 (2007)
15. Arbab, F., Bruni, R., Clarke, D., Lanese, I., Montanari, U.: Tiles for Reo. In: Corradini, A., Montanari, U. (eds.) *WADT 2008. LNCS*, vol. 5486, pp. 37–55. Springer, Heidelberg (2009)
16. Arbab, F., Chothia, T., Meng, S., Moon, Y.-J.: Component connectors with QoS guarantees. In: Murphy, A.L., Vitek, J. (eds.) *COORDINATION 2007. LNCS*, vol. 4467, pp. 286–304. Springer, Heidelberg (2007)
17. Arbab, F., Chothia, T., van der Mei, R., Meng, S., Moon, Y.-J., Verhoef, C.: From coordination to stochastic models of QoS. In: Field, Vasconcelos (eds.) [45], pp. 268–287
18. Arbab, F., Koehler, C., Maraïkar, Z., Moon, Y.-J., Proença, J.: Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. Tool demo session at *FACS 2008* (2008)
19. Arbab, F., Kokash, N., Meng, S.: Towards using Reo for compliance-aware business process modeling. In: Margaria, T., Steffen, B. (eds.) *ISoLA. CCIS*, vol. 17, pp. 108–123. Springer, Heidelberg (2008)
20. Arbab, F., Mavaddat, F.: Coordination Through Channel Composition. In: Arbab, F., Talcott, C. (eds.) *COORDINATION 2002. LNCS*, vol. 2315, pp. 22–39. Springer, Heidelberg (2002)
21. Arbab, F., Meng, S.: Synthesis of Connectors From Scenario-Based Interaction Specifications. In: Chaudron, M.R.V., Szyperski, C.A., Reussner, R. (eds.) *CBSE 2008. LNCS*, vol. 5282, pp. 114–129. Springer, Heidelberg (2008)
22. Arbab, F., Meng, S., Baier, C.: Synthesis of Reo circuits from scenario-based specifications. *Electr. Notes Theor. Comput. Sci.* 229(2), 21–41 (2009)
23. Arbab, F., Meng, S., Moon, Y.-J., Kwiatkowska, M.Z., Qu, H.: Reo2MC: a tool chain for performance analysis of coordination models. In: van Vliet, H., Issarny, V. (eds.) *ESEC/SIGSOFT FSE*, pp. 287–288. ACM (2009)
24. Arbab, F., Rutten, J.J.M.M.: A Coinductive Calculus of Component Connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) *WADT 2003. LNCS*, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
25. Baeten, J.C.M., Weijland, W.P.: *Process Algebra*. Cambridge University Press (1990)
26. Baier, C.: Probabilistic models for Reo connector circuits. *Journal of Universal Computer Science* 11(10), 1718–1748 (2005)

27. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal Verification for Components and Connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009)
28. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In: Field, Vasconcelos (eds.) [45], pp. 247–267
29. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S., Leister, W.: Design and Verification of Systems with Exogenous Coordination Using Vereofy. In: Margaria, T., Steffen, B. (eds.) ISO/LA 2010. LNCS, vol. 6416, pp. 97–111. Springer, Heidelberg (2010)
30. Baier, C., Klein, J., Klüppelholz, S.: Modeling and Verification of Components and Connectors. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 114–147. Springer, Heidelberg (2011)
31. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.* 61(2), 75–113 (2006)
32. Baier, C., Wolf, V.: Stochastic Reasoning About Channel-Based Component Connectors. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 1–15. Springer, Heidelberg (2006)
33. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. *Information and Control* 60, 109–137 (1984)
34. Blechmann, T., Baier, C.: Checking equivalence for Reo networks. *Electr. Notes Theor. Comput. Sci.* 215, 209–226 (2008)
35. Bonsangue, M.M., Clarke, D., Silva, A.: Automata for context-dependent connectors. In: Field, Vasconcelos (eds.) [45], pp. 184–203
36. Bonsangue, M.M., Izadi, M.: Automata based model checking for Reo connectors. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 260–275. Springer, Heidelberg (2010)
37. Changizi, B., Kokash, N., Arbab, F.: A unified toolset for business process model formalization. In: Proc. of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2010 (2010); satellite event of ETAPS
38. Clarke, D.: Coordination: Reo, Nets, and Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 226–256. Springer, Heidelberg (2008)
39. Clarke, D.: A basic logic for reasoning about connector reconfiguration. *Fundam. Inform.* 82(4), 361–390 (2008)
40. Clarke, D., Costa, D., Arbab, F.: Modelling Coordination in Biological Systems. In: Margaria, T., Steffen, B. (eds.) ISO/LA 2004. LNCS, vol. 4313, pp. 9–25. Springer, Heidelberg (2006)
41. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. *Sci. Comput. Program.* 66(3), 205–225 (2007)
42. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Deconstructing Reo. *Electr. Notes Theor. Comput. Sci.* 229(2), 43–58 (2009)
43. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. *Sci. Comput. Program.* 76(8), 681–710 (2011)
44. Costa, D.: Formal Models for Context Dependent Connectors for Distributed Software Components and Services. PhD thesis, Vrije Universiteit Amsterdam (2010), <http://dare.ubvu.vu.nl/handle/1871/16380>
45. Field, J., Vasconcelos, V.T. (eds.): COORDINATION 2009. LNCS, vol. 5521, pp. 225–246. Springer, Heidelberg (2009)

46. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer, Heidelberg (1999)
47. Gadducci, F., Montanari, U.: The tile model. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 133–166. MIT Press, Boston (2000)
48. Grabe, I., Jaghoori, M.M., Aichernig, B.K., Baier, C., Blechmann, T., de Boer, F.S., Griesmayer, A., Johnsen, E.B., Klein, J., Klüppelholz, S., Kyas, M., Leister, W., Schlatte, R., Stam, A., Steffen, M., Tschirner, S., Xuedong, L., Yi, W.: Credo methodology: Modeling and analyzing a peer-to-peer system in credo. *Electr. Notes Theor. Comput. Sci.* 266, 33–48 (2010)
49. Groote, J.F., Mathijssen, A., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.: The formal specification language mCRL2. In: Brinksma, E., Harel, D., Mader, A., Stevens, P., Wieringa, R. (eds.) *MMOSS. Dagstuhl Seminar Proceedings*, vol. 06351. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl (2006)
50. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall (1985)
51. Hoare, C.A.R., Jifeng, H.: *Unifying Theories of Programming*. Prentice Hall, London (1998)
52. Izadi, M., Bonsangue, M.M., Clarke, D.: Modeling component connectors: Synchronisation and context-dependency. In: Cerone, A., Gruner, S. (eds.) *SEFM*, pp. 303–312. IEEE Computer Society (2008)
53. Izadi, M., Bonsangue, M.M., Clarke, D.: Büchi automata for modeling component connectors. *Software and System Modeling* 10(2), 183–200 (2011)
54. Izadi, M., Movaghgar, A.: Failure-based equivalence of constraint automata. *Int. J. Comput. Math.* 87(11), 2426–2443 (2010)
55. Jongmans, S.-S.T.Q., Arbab, F.: Correlating formal semantic models of Reo connectors: Connector coloring and constraint automata. In: Silva, A., Bliudze, S., Bruni, R., Carbone, M. (eds.) *ICE. EPTCS*, vol. 59, pp. 84–103 (2011)
56. Jongmans, S.-S.T.Q., Krause, C., Arbab, F.: Encoding context-sensitivity in Reo into non-context-sensitive semantic models. In: Meuter, Roman (eds.) [82], pp. 31–48
57. Kanters, O.: QoS analysis by simulation in Reo (2010)
58. Kemper, S.: SAT-based Verification for Timed Component Connectors. *Electr. Notes Theor. Comput. Sci.* 255, 103–118 (2009)
59. Kemper, S.: Compositional Construction of Real-Time Dataflow Networks. In: Clarke, D., Agha, G. (eds.) *COORDINATION 2010. LNCS*, vol. 6116, pp. 92–106. Springer, Heidelberg (2010)
60. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: The Orc Programming Language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) *FMOODS 2009. LNCS*, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
61. Klein, J., Klüppelholz, S., Stam, A., Baier, C.: Hierarchical Modeling and Formal Verification. An Industrial Case Study Using Reo and Vereofy. In: Salaün, G., Schätz, B. (eds.) *FMICS 2011. LNCS*, vol. 6959, pp. 228–243. Springer, Heidelberg (2011)
62. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. *Electr. Notes Theor. Comput. Sci.* 175(2), 19–37 (2007)
63. Koehler, C., Arbab, F., de Vink, E.P.: Reconfiguring Distributed Reo Connectors. In: Corradini, A., Montanari, U. (eds.) *WADT 2008. LNCS*, vol. 5486, pp. 221–235. Springer, Heidelberg (2009)

64. Koehler, C., Costa, D., Proença, J., Arbab, F.: Reconfiguration of Reo connectors triggered by dataflow. In: Ermel, C., Heckel, R., de Lara, J. (eds.) *Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, vol. 10, pp. 1–13 (2008); Home Page, <http://www.easst.org/eceasst/>, ECEASST ISSN 1863-2122
65. Koehler, C., Lazovik, A., Arbab, F.: ReoService: Coordination modeling tool. In: Krämer, et al. (eds.) [73], pp. 625–626
66. Koehler, C., Lazovik, A., Arbab, F.: ReoService: Coordination Modeling Tool. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) *ICSOC 2007*. LNCS, vol. 4749, pp. 625–626. Springer, Heidelberg (2007)
67. Kokash, N., Krause, C., de Vink, E.P.: Data-aware design and verification of service compositions with Reo and mCRL2. In: *SAC 2010: Proc. of the 2010 ACM Symposium on Applied Computing*, pp. 2406–2413. ACM, New York (2010)
68. Kokash, N., Arbab, F.: Formal behavioral modeling and compliance analysis for service-oriented systems. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) *FMCO 2008*. LNCS, vol. 5751, pp. 21–41. Springer, Heidelberg (2009)
69. Kokash, N., Arbab, F.: Applying Reo to service coordination in long-running business transactions. In: Shin, S.Y., Ossowski, S. (eds.) *SAC*, pp. 1381–1382. ACM (2009)
70. Kokash, N., Arbab, F., Changizi, B., Makhnist, L.: Input-output conformance testing for channel-based service connectors. In: Aceto, L., Mousavi, M.R. (eds.) *PACO. EPTCS*, vol. 60, pp. 19–35 (2011)
71. Kokash, N., Krause, C., de Vink, E.P.: Verification of Context-Dependent Channel-Based Service Models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) *FMCO 2009*. LNCS, vol. 6286, pp. 21–40. Springer, Heidelberg (2010)
72. Kokash, N., Krause, C., de Vink, E.P.: Time and data-aware analysis of graphical service models in Reo. In: Fiadeiro, J.L., Gnesi, S., Maggiolo-Schettini, A. (eds.) *SEFM*, pp. 125–134. IEEE Computer Society (2010)
73. Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.): *ICSOC 2007*. LNCS, vol. 4749. Springer, Heidelberg (2007)
74. Krause, C.: Integrated structure and semantics for Reo connectors and Petri nets. In: *ICE 2009: Proc. 2nd Interaction and Concurrency Experience Workshop*. *Electronic Proceedings in Theoretical Computer Science*, vol. 12, p. 57 (2009)
75. Krause, C.: *Reconfigurable Component Connectors*. PhD thesis, Leiden University (2011), <https://openaccess.leidenuniv.nl/handle/1887/17718>
76. Krause, C., Maraïkar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Sci. Comput. Program.* 76(1), 23–36 (2011)
77. Lazovik, A., Arbab, F.: Using Reo for service coordination. In: Krämer, et al. (eds.) [73], pp. 398–403
78. Maraïkar, Z., Lazovik, A.: Reforming mashups. In: *Proceedings of the 3rd European Young Researchers Workshop on Service Oriented Computing (YR-SOC 2008)*. Imperial College London (June 2008)
79. Meng, S., Arbab, F.: On Resource-Sensitive Timed Component Connectors. In: Bonsangue, M.M., Johnsen, E.B. (eds.) *FMOODS 2007*. LNCS, vol. 4468, pp. 301–316. Springer, Heidelberg (2007)
80. Meng, S., Arbab, F.: QoS-driven service selection and composition. In: Billington, J., Duan, Z., Koutny, M. (eds.) *ACSD*, pp. 160–169. IEEE (2008)
81. Meng, S., Arbab, F.: Connectors as designs. *Electr. Notes Theor. Comput. Sci.* 255, 119–135 (2009)

82. De Meuter, W., Roman, G.-C. (eds.): COORDINATION 2011. LNCS, vol. 6721. Springer, Heidelberg (2011)
83. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
84. Milner, R.: Elements of interaction - turing award lecture. *Commun. ACM* 36(1), 78–89 (1993)
85. Misra, J., Cook, W.R.: Computation orchestration. *Software and System Modeling* 6(1), 83–110 (2007)
86. Moon, Y.-J.: Stochastic Models for Quality of Service of Component Connectors. PhD thesis, Leiden University (2011)
87. Moon, Y.-J., Silva, A., Krause, C., Arbab, F.: A compositional semantics for stochastic Reo connectors. In: Mousavi, Salaün (eds.) [88], pp. 93–107
88. Mousavi, M.R., Salaün, G. (eds.): Proceedings Ninth International Workshop on the Foundations of Coordination Languages and Software Architectures. EPTCS, vol. 30 (2010)
89. Mousavi, M.R., Sirjani, M., Arbab, F.: Formal semantics and analysis of component connectors in Reo. *Electr. Notes Theor. Comput. Sci.* 154(1), 83–99 (2006)
90. Odersky, M.: Report on the programming language Scala (2002), <http://lamp.epfl.ch/~odersky/scala/reference.ps>
91. Proença, J.: Synchronous Coordination of Distributed Components. PhD thesis, Leiden University (2011), <https://openaccess.leidenuniv.nl/handle/1887/17624>
92. Proença, J., Clarke, D.: Coordination models Orc and Reo compared. *Electr. Notes Theor. Comput. Sci.* 194(4), 57–76 (2008)
93. Proença, J., Clarke, D., de Vink, E.P., Arbab, F.: Decoupled execution of synchronous coordination models via behavioural automata. In: Mousavi, M.R., Ravara, A. (eds.) FOCLASA. EPTCS, vol. 58, pp. 65–79 (2011)
94. Rutten.: Behavioural differential equations: A coinductive calculus of streams, automata, and power series. *TCS: Theoretical Computer Science*, 308 (2003)
95. Rutten, J.J.M.M.: Elements of stream calculus (an extensive exercise in coinduction). *Electr. Notes Theor. Comput. Sci.*, 45 (2001)
96. Rutten, J.J.M.M.: A coinductive calculus of streams. *Mathematical Structures in Computer Science* 15(1), 93–147 (2005)
97. Sangiorgi, D., Walker, D.: *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York (2001)
98. Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., van den Heuvel, W.-J.: Business Process Compliance Through Reusable Units of Compliant Processes. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010. LNCS, vol. 6385, pp. 325–337. Springer, Heidelberg (2010)
99. Talcott, C.L., Sirjani, M., Ren, S.: Comparing three coordination models: Reo, ARC, and PBRD. *Sci. Comput. Program.* 76(1), 3–22 (2011)
100. Verhoef, C., Krause, C., Kanters, O., van der Mei, R.: Simulation-based performance analysis of channel-based coordination models. In: Meuter, Roman [82], pp. 187–201
101. Wegner, P.: Coordination as constrained interaction (extended abstract). In: Ciancarini, P., Hankin, C. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 28–33. Springer, Heidelberg (1996)
102. Yokoo, M.: *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Springer Series on Agent Technology. Springer, New York (2000) NTT

A Formal Methodology for Compositional Cross-Layer Optimization*

Minyoung Kim¹, Mark-Oliver Stehr¹, Carolyn Talcott¹
Nikil Dutt², and Nalini Venkatasubramanian²

¹ SRI International, USA

{mkim, stehr, clt}@csl.sri.com

² University of California, Irvine, USA

{dutt, nalini}@ics.uci.edu

Abstract. The xTune framework employs iterative tuning using lightweight formal verification at runtime with feedback for dynamic adaptation of mobile real-time embedded systems. To enable trade-off analysis across multiple layers of abstraction and predict the possible property violations as the system evolves dynamically over time, an executable formal specification is developed for each layer of the system under consideration. The formal specification is then analyzed using statistical analysis, to determine the impact of various policies for achieving a variety of end-to-end properties in a quantifiable manner. The integration of formal analysis with dynamic behavior from system execution results in a feedback loop that enables model refinement and further optimization of policies and parameters. Finally, we propose a composition method for coordinated interaction of optimizers at different abstraction layers. The core idea of our approach is that each participating optimizer can restrict its own parameters and exchange refined parameters with its associated layers. We also introduce sample application domains for future research directions.

1 Vision

An overarching characteristic of next-generation mobile applications is that they are often data intensive and rich in multimedia content with images, video, and audio data that is fused together from disparate distributed information sources. The content-rich data is expected to be obtained from, delivered to, and processed on resource-constrained devices (sensors, PDAs, cellular handsets) carried by users in highly dynamic environments (e.g., delay, jitter, erroneous transmission). Clearly, in such a scenario, the dual goals of ensuring adequate application QoS (Quality of Service) and optimizing resource utilization in the network, devices, and content servers present significant challenges.

* Support from National Science Foundation Grant 0932397 (A Logical Framework for Self-Optimizing Networked Cyber-Physical Systems) is gratefully acknowledged. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

Specific adaptations have been developed within each abstraction layer (application, middleware, OS, hardware) to perform the QoS provision on resource limited devices. For example, the OS adaptations typically change the allocation and scheduling in response to application and resource variations [21,4]. We refer to these individual adaptation techniques as *policies*. Next, we identify *parameters* to manipulate the behavior of a policy. For example, the OS layer policy can be fine tuned by selecting the appropriate tolerance level of QoS in terms of task completion that satisfies its deadline [4].

Understanding interactions across layers and exploiting them in such systems is essential since policy/parameter settings at one layer can have a significant impact on the behavior at other layers. A cross-layer approach is needed to deal with complexity of such systems and the dynamic environment in which such applications execute. Our prior experience (FORGE [3,13]) with developing algorithms for cross-layer adaptation based on QoS/energy trade-offs in distributed mobile multimedia applications has given us valuable insights into the issues to be addressed. In particular, the middleware framework DYNAMO [14] performs joint adaptation at the proxy server to drive on-device adaptation for end-to-end adaptations such as dynamic video transcoding and traffic shaping. GRACE [22] also aims to trade off multimedia quality against energy by introducing a hierarchy of global (i.e., coordinating all layers) and internal (i.e., within the individual layers) adaptation.

While existing work has shown the effectiveness of cross-layer adaptation, many of these efforts try to address the average case behavior without verifiable guarantees on their solutions. As the system evolves dynamically over time, the applications need a mechanism that can be used to formally prove various properties pertaining to energy usage, delays, and so on for any given configuration of policies/parameters to derive, analyze, and validate cross-layer adaptation. Our hypothesis is that a comprehensive design methodology based on a formal reasoning framework will provide an effective basis for tuning mobile embedded systems under a multitude of constraints.

To illustrate the challenges introduced by the cross-layer nature of mobile real-time embedded applications, consider the scenario of a mobile device executing a video conferencing application carried by a user moving from $Zone_0$ to $Zone_4$ in Figure 1. The objective is to support QoS needs by instantiation and tuning of the appropriate policy at each layer with its parameter values (*complexity*). In particular, we strive to achieve quantifiable guarantees with regard to the quality of selected policies and parameters (*verification*). Last, when a user moves to a different zone, we need a way of reflecting it for iterative tuning as well as static instantiation of policies and parameters (*dynamicity*). We elaborate these challenges below.

- **Complexity:** Given a set of application needs and a system configuration, we need to choose appropriate operating points, through selection of both the policy and the parameter settings at each layer as depicted in Figure 1. Considering the composite effect of multiple policies at each layer demands a cross-layer approach. A holistic approach to understanding cross-layer

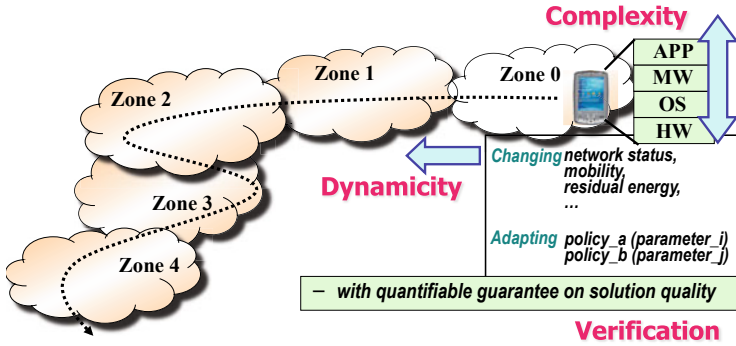


Fig. 1. Challenges: Instantiation and tuning of the appropriate polices and corresponding parameter values (*complexity*) with quantifiable guarantee (*verification*) while reflecting changes in the system and environment (*dynamicity*)

interaction in such systems is essential, since policies made at one layer can (sometimes adversely) affect behavior at other layers.

- **Verification:** During this process, we need to generate a set of candidate policies with possible parameter settings based on the trade-off analysis to determine the best feasible choice among these candidates. If no policy can satisfy the requirements, we must determine how to relax the constraints and may need to repeat policy selection. For such informed selection, it is essential to perform bound/sensitivity analysis on the impact of the policy/parameter that can provide some notion of guarantee on the solution.
- **Dynamicity:** The system and environment may keep evolving as a user moves from one zone to another as depicted in Figure 1, requiring dynamic policy/parameter analysis and tuning. During operation, policy selection and parameter tuning requires the procedure to determine (i) which changes demand our attention, (ii) if it has a significant impact on timing/QoS/performance, and (iii) how the policy/parameter should be recomputed.

To ensure adequate application QoS and resource utilization with timing and reliability concerns, the ability to compensate *on the fly* for property violations at different layers of abstraction is of paramount importance since there are several sources of unpredictability (e.g., delay, packet drop, user mobility) in a mobile embedded system that introduce nondeterminism. Furthermore, system-level optimizations for effective utilization of distributed resources can interfere with the properties of executing applications. For instance, dynamic voltage scaling (DVS) mechanisms slow down processors to achieve power savings, but at the cost of increased execution times for tasks. Many applications have flexible QoS needs that dictate how tolerant they are to delays and errors — the lack of stringent timing needs can be adaptively exploited for better end-to-end resource utilization.

We enumerate sample questions we would like to answer:

1. How does one decide what policies and parameters to assign to each abstraction layer to minimize the overall energy consumption while providing a sufficient level of QoS with verifiable/quantifiable solution quality? This must be achieved for an energy-constrained mobile embedded device dealing with displaying delay-sensitive multimedia data over a lossy network.
2. How can we exploit system state for dynamic adaptations? When the system evolves over time, how can we accommodate it? We need a way of reflecting dynamics. Specifically, this requires determining which attribute can be a trigger for adaptations and how to refine our model.
3. How can we support cross-layer adaptation while individual policies perform their own optimization? Unlike existing research literature that relies on a global coordinator at a certain layer, we address the issue of how to support cross-layer adaptation while allowing autonomy of individual layers' policies.

To lend focus, we (i) choose mobile multimedia as an application domain, and (ii) select performance criteria that require adaptations such as device residual energy, application QoS/timing needs, and reliable content delivery. A preliminary study [5] demonstrated the need for integration of formal methods with experimentally based cross-layer optimization techniques [3,13] for such application domain and performance criterion. Within the xTune framework, we support compositional online optimization of individual policies at each layer [11]. xTune employs statistical formal methods to analyze given cross-layered optimization policies with a quantifiable guarantee on the solution quality [10].

2 Overview of Technical Approach

Our approach starts with a formal specification of the abstraction layers and subsequent statistical evaluation to verify probabilistic properties. We propose a lightweight formal methodology in the sense that we exploit statistical techniques on a system model represented in an executable formal specification. The proposed approach provides statistically meaningful answers from on-demand trace generation rather than keeping the entire spectrum of possible traces.

Our approach supports iterative tuning and compositional cross-layer optimization and can deal with quantifiable guarantees, dynamicity, and complexity issues:

- **Quantifiable Solution Quality:** Our work examines the impact of various resource management techniques on end-to-end timing/QoS properties based on statistical evaluation for verifiable/quantifiable solutions, and enables informed selection of resource management policies along with rules for instantiation of parameters that derive the policies.
- **Iterative Tuning:** We enhance such lightweight formal modeling and analysis by integrating it with observations of system execution behavior to achieve adaptive reasoning by providing more precise information on current execution and future state.

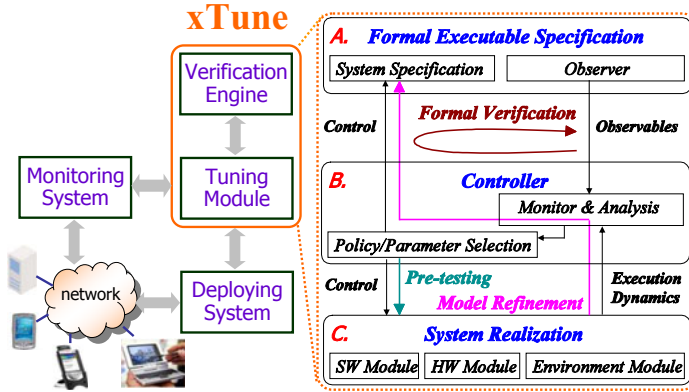


Fig. 2. xTune Cross-layer System Tuning Framework

- **Compositional Cross-Layer Optimization:** We propose a compositional approach for the cross-layer optimization that avoids high overhead introduced by traditional global approaches; our compositional approach allows sublayers’ optimization results to be used by the other sublayer optimizers as constraints.

Our work is validated and tested in the context of distributed mobile multimedia applications that have wide consumer interest. Using multimedia as a demonstrator, we have developed general principles and a framework. In many contexts, enabling verifiable adaptation in terms of timing/QoS guarantees provides an additional degree of confidence to improve other cross-layer reliability measures in the context of multimedia applications.

3 Supporting Model-Based Composition with xTune

The xTune framework uses iterative system tuning to support adaptations. In particular, our approach tunes the parameters in a compositional manner allowing coordinated interaction among sublayer optimizers. The xTune framework initially performs property checking and quantitative analysis of candidate policy/parameter settings via formal executable specifications followed by statistical techniques. Iterative tuning allows model refinement from up-to-date and continuous observations of system execution behavior. Furthermore, the results can be used to improve adaptation by verifying given system properties or by relaxing constraints.

Figure 2 presents the overall flow of our approach. *Box A* represents the formal modeling. The core of our formal modeling approach is to develop formal executable models of system components at each layer of interest. These models express functionality, timing, and other resource considerations at the appropriate level of detail and using appropriate interaction mechanisms (clock ticks,

synchronous or asynchronous messages). Models of different layers are analyzed in isolation and composed to form cross-layer specifications. We use the Maude system for developing and analyzing formal specifications. One advantage of formal executable models is that they can be subjected to a wide range of formal analysis, including single execution scenarios, search for executions leading to states of interest, and model checking to understand properties of execution paths.

Box B in Figure 2 shows the evaluation phase of given specifications to generate statistics for properties and values of interest. Specifically, we have developed new analysis techniques (statistical model checking and statistical quantitative analysis) that combine statistical and formal methods, and applied them to a case study of a videophone application [10]. We have developed a compositional cross-layer optimization by coordinated interaction among local (sublayer) optimizers through constraint refinement. The constraint refinement allows encapsulation of detailed system state information. In compositional optimization, each local optimizer uses refinement results of other optimizers as its constraints. The constraint representation can be used as the generic interface among different local optimizers, leading to substantial improvement of solution quality at low complexity.

Using such models and analysis, tools can be developed to achieve adaptive refinement of an end-to-end system specification into appropriate policy/parameter settings. We use an *iterative* tuning strategy that combines formal methods (verification) with dynamic system execution behavior (obtained by either simulation or implementation). The execution behavior from system realization (*Box C* in Figure 2) is fed back into the formal modeling to refine the executable specification (*model refinement*). In addition, we can assure the quality of a new policy/parameter constructed by the controller. In Figure 2, *pre-testing* on a system realization can lead to improvements because typically the formal model cannot cover all the possible implementation details of a real system.

4 Model-Based Compositional Cross-Layer Optimization

4.1 Understanding the Issue of Cross-Layer Optimization

To enhance system utility capturing the effectiveness of the settings relative to the user and system objectives in the context of mobile applications, researchers have proposed a wide variety of techniques at different system layers. Note that one key performance metric for such techniques is how well they manage utility under a multitude of constraints in a dynamic situation. Since utility comes with cost in terms of performance, energy consumption, storage requirements, and bandwidth used, one needs to optimize utility in the context of the operating conditions. However, most optimization techniques consider only a single system layer, remaining unaware of the strategies employed in the other layers. A cross-layer approach that is cognizant of features, limitations, and dynamic changes at each layer enables better optimization than a straightforward composition of individual layers, because solutions for each individual layer can be globally

suboptimal. To coordinate the individual techniques in a cross-layer manner based on the operating condition, one needs to

- Quantify the effect of various optimization policies at each layer on system properties
- Explore methods of taking the impact of each policy into account and compensating for it at other layers

Abstraction and Model Refinement. We develop a formal methodology to specify and analyze features/constraints/needs at each layer and to correlate them across layers to realize cross-layer tuning. Our approach is to start with an executable formal model based on rewriting logic specifying a space of possible behaviors. In [10,9,11], we use the Maude [2] rewriting logic formalism to develop executable specifications of each layer in isolation and in composition as well as representing their timed behavior.

In most cases, the model cannot be fully characterized in advance and can change while the system is in operation, which is why model refinement is an essential component of our architecture. To reflect execution dynamics, we perform model refinement from observed system execution behavior by equipping the controller with a feedback loop to experiment with the system realization [9]. The system can start with a default model (e.g., a model with default parameters about execution times), which is incrementally refined during the operation of the system. Models can be passively refined by observations — e.g., from CPU usage, while the system is executing its primary function or mission — or it can be actively pursued by exploration, which may require physical actions. Often, combinations of the passive and active modes of model refinement will be needed for acceptable performance with low exploration overhead. Within our framework, there are at least two roles for feedback from observation of system execution behavior: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the policy. We define the former as *long-term tuning*, and the latter as *short-term tuning*. In the xTune framework, we support long-term tuning through model refinement without active exploration.

Statistical Analysis. To analyze the behavior of the system (e.g., in terms of discrete or continuous observable properties) in a probabilistic sense, we have implemented two lightweight formal analysis techniques: statistical model checking and statistical quantitative analysis. To formally verify certain properties, traditional approaches maintain tree-like structures of the entire spectrum of possible traces with probability measures and exhaustively evaluate the system, which leads to excessive memory requirements that limit scalability of the solution. In contrast, the xTune approach is a lightweight formal method, since the optimization problem for adaptation does not require an exact solution. This allows us to generate traces on demand and provide statistically meaningful answers, unlike exhaustive numerical methods, which aim at exact solutions.

In [10], we extended the quantitative approach of [1] by an on-demand sample generation that can compute the sample size sufficient to reach confidence in the normality of data, and then utilize the normal distribution to obtain the error bound and confidence interval for quantitative analysis. The xTune framework also implements two statistical model-checking techniques: the sequential probability ratio test [20] and black-box testing [16]. Given a property, the sequential probability ratio test [20] continues sample generation until its answer about accepting or rejecting the hypothesis can be guaranteed to be correct within the required error bounds. Black-box testing [16] instead computes the statistical significance (p -value) for a given number of samples without having any control over the execution. These statistical techniques can be used to quantify statistical performance (e.g., execution times) with a specific confidence and to verify properties (e.g., battery depletion), which may be satisfied only in a probabilistic sense. They provide a quantifiable solution to enable policy-based operation and adaptation as well as parameter setting and adjustment for selected policies.

4.2 Constraint Refinement and Composition

In the xTune framework, constraint-based optimization is guided by a model of the system to be optimized. The compositional optimization is purely generic in the sense that we can construct an interface language for generic composition (e.g., negotiation and contract), which can be used with heterogeneous application specifications. An interesting extension would be distributed compositional techniques that integrate randomization and symbolic reasoning. For that purpose, the constraint language needs to be expressive enough to support strategies for distributed cooperative optimization.

In the following, we describe our composition method. First, we explain the idea of constraint refinement for robust optimization. Then, we define our compositional cross-layer optimization based on this representation. Our experiments show that the encapsulation of the local optimization at each sublayer leads to substantial improvement of solution quality at low complexity [11].

Constraint Refinement. Given an optimization problem with the model \mathcal{M} and the parameter space \mathbb{P} , our approach attempts to quickly find a region $P \in \mathcal{R}(\mathbb{P})^1$ containing a nearly optimal solution by the following heuristics:

1. **Recursive Resampling:** We obtain observables by Monte Carlo sampling over the current region $P_i \in \mathcal{R}(\mathbb{P})$ using the model \mathcal{M} . Subsequently, we refine P_i to P_{i+1} such that the utility is maximized based on the samples available, and $size(P_{i+1}) = size(P_i) \cdot \tau_i$, where τ_i ($0.0 < \tau_i < 1.0$) represents the i -th refinement ratio. The new region P_{i+1} is then used as the current region and the process is repeated.

¹ Region $P \in \mathcal{R}(\mathbb{P}) \iff P \subseteq \mathbb{P}$ is a closed convex set, (i.e., if $(x, z \in P) \wedge (x < y < z)$, then $(y \in P)$) and P is finitely representable (e.g., interval-based).

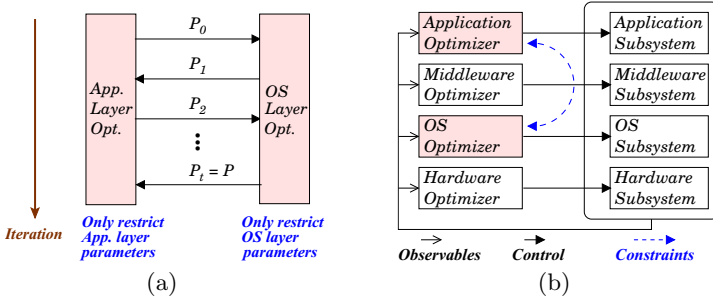


Fig. 3. Constraint Refinement for Composition (a) Parallel Composition of Layers, (b) Compositional Cross-layer Optimization

2. **Interval-based Description:** For simplicity we use regions defined by the Cartesian product of intervals for each of the parameters. For example, an application layer region might be

$$P_{App} = [Param1_{min}, Param1_{max}] \times [Param2_{min}, Param2_{max}].$$

More expressive constraint languages are possible in our framework and should be investigated in the future.

3. **Generic Constraint-based Interface:** The input (P_i) and output (P_{i+1}) of each refinement step are regions (infinite sets), and our approach lifts the level of abstraction by treating P_i as *constraints* (finite symbolic representations) when we restrict the resampling space to find P_{i+1} .

The process of constraint refinement can be stated as a chain

$$\mathbb{P} = P_0 \supseteq P_1 \supseteq P_2 \supseteq \dots \supseteq P_t = P$$

where P is the set of admissible parameter settings at termination after t iterations.

Our experimental results indicate that the constraint refinement can be effectively used for robust parameter selection by refining spectrum of reliable policies and parameters. One key feature of this approach is that we can coordinate parallel composition of individual optimizers as illustrated in Figure 3(a). Each sublayer optimizer controls a subset of parameters. For instance, the application layer optimizer only restricts its own parameters (\mathbb{P}_{App}), while the OS layer optimizer only restricts OS-related parameters (\mathbb{P}_{OS}). The constraints P_i are used as inputs *and* outputs of individual (sublayer) optimizers.

Composition through constraint refinement reduces the possibility of conflicts because of the more general notion of a solution compared with traditional single-point optimizers. More important, constraint refinement enables simple yet powerful cross-layer optimization via composition (Figure 3(b)), as discussed below.

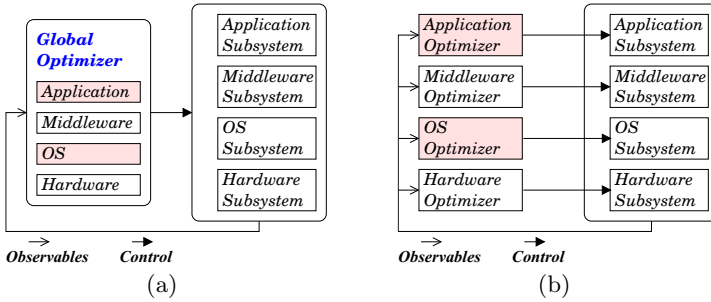


Fig. 4. Comparison among Online Optimizations (a) Global Cross-layer Optimization, (b) Without Cross-layer Optimization

Online Cross-Layer Optimization. The primary goal of our framework is to enable online cross-layer optimization that provides the refined parameter settings from which a system can select any suitable operating point within the region as explained above. The constraint refinement allows encapsulation of detailed system optimization information. This opens up the possibility of coordinated interaction (composition) instead of relying on a global view. Figures 4(a), 4(b), and 3(b) compare the global vs. local vs. compositional approach for cross-layer optimization, respectively. The key idea underlying the *compositional* optimization is to exchange the local optimizer’s decision for an informed selection. This allows us to achieve a balance between *global* optimization’s full awareness with high overhead and *local* optimization’s minimal complexity with poor solution quality.

The sampling strategy explores the search space of potential solutions by constraining the behavior of local optimizers in accordance with the other optimizers’ refinement results. Thus, the constraint language can serve as a generic interface among different local optimizers, leading to improvements of solution quality and convergence speed. In comparison, a global cross-layer optimizer that resides at a certain layer that is fully aware of the complex system dynamics can introduce unacceptable overhead.

A similar strategy can be applied to other optimization techniques (e.g., simulated annealing [12]). The strict convergence to a single point, however, may not be achievable in the sense that at each step the intermediate parameter settings may be totally different from the previous iteration. These types of abrupt and/or constant parameter changes are not desirable in practice. Constraint refinement can still undergo constant parameter changes, but with *lower* impact since any parameter settings (p_i) within the region (P_i) can be chosen, and the probability that p_i is valid after the next iteration ($p_i \in P_{i+1}$) is proportional to the refinement ratio. We can also easily see that the situation will worsen with conflicting local objectives.

Our approach is not limited to a specific constraint refinement protocol scheme. Compositional optimization through constraint refinement enables a controller

to coordinate existing optimizers (possibly distributed) that can accommodate different objectives by treating them as black boxes, which in turn permits them to operate in parallel. Different solutions obtained concurrently can be unified by taking the intersection, which corresponds to the conjunction at the symbolic level.

4.3 Integration of Formal Analysis with System Dynamics

To reflect execution dynamics, the xTune framework performs model refinement from observed system execution behavior by equipping the controller with a feedback loop to experiment with the system realization as highlighted in Figure 2. Within our framework, there are two roles for feedback from observation of system execution behavior: it can be used to improve the model (to make it more accurately match the real environment) as we presented in [9], and it can also be used to directly improve the policy. In [9] the formal specification is refined by replacing timing information with observations from dynamic system execution either by system realization (e.g., simulator) or real implementation, in order to more realistically reflect the actual executions characterizing the system in practice such as data dependent execution times.

5 Sample Application Domains

Here we lay out possible future research directions that we believe can benefit from model-based compositional cross-layer optimization.

5.1 Networked Cyber-Physical Systems

As elaborated in [18], system control and optimization in networked cyber-physical systems (NCPS) is a challenging task. Traditional optimization techniques that strive for optimal solutions based on precise models are not suitable for most real-world problems, where models have many dimensions of uncertainty, and optimality is neither desirable nor achievable. What is needed in practice are strategies to find acceptable and robust solutions that are sufficient to achieve the goal while taking into account the limitations of the models and of the available resources. Capabilities to explore the state space of NCPS have fundamental limitations. The exploration can be expensive in terms of computation, and physical actions can be costly in terms of time, energy, and other resources or even harmful to humans or to the environment.

Furthermore, the overall goal of NCPS cannot be simply decomposed top-down into goals that are optimized locally at each node and each layer, because solutions may require cooperation across layers and across nodes. It is important to keep in mind that even abstract models can be quite complex with multiple and nonoverlapping regions of potential solutions so that purely local gradient-based optimization strategies are clearly insufficient. Given that modifications in parameters (e.g., node position) cannot always be achieved instantaneously, reaching a new improved solution may require transition through intermediate

states with lower utility (e.g., lower performance). The distributed nature of NCPS, the limited communication capabilities, the uncertainties in the environment, and the possibility of failures further exacerbate this situation, because system operation is inherently asynchronous.

On the other hand, NCPS with a large number of nodes offers many advantages including fault tolerance, distributed sensing, coordinated actions, and inherent parallelism for computational processes. Technically, a vast range of capabilities is already available at the hardware level, but the challenge to design a software architecture that can exploit those capabilities and to present them as a single cyber-physical system is far from being met. In this regard, [7] provides a prototype of a distributed logical framework based on the partially ordered knowledge-sharing model and an API for cyber-physical devices that enables interaction with the physical world (see <http://ncps.cs1.sri.com> for details). The proposed API provides a uniform abstraction for a wide range of NCPS applications, especially those concerned with distributed sensing, optimization, and control. Using the API with or without a distributed logic, NCPS can be programmed to adapt to a wide range of operating points between autonomy and cooperation to overcome limitations in connectivity and resources, as well as uncertainties and failures [7,17,8]. Along this line of research, our methodology can be extended to consider multiple distributed cyber-physical nodes as local optimizers (horizontal composition in addition to vertical (layered) composition). To capture the distributed and heterogeneous nature of NCPS, the compositional optimization strategies need to be generalized to include composition among various local optimizers across layers as well as across nodes, leading us to distributed cooperative constraint refinement.

5.2 Dependable Instrumented Cyber-Physical Spaces

The ability to integrate sensing and communication platforms with large-scale distributed storage/computing facilities and software services enables the creation of instrumented cyber-physical spaces (ICPS). Applications dictate application-specific constraints on the timeliness and accuracy/quality at which information must be captured and delivered from the infrastructure. Repurposing the infrastructure and its software/hardware resources dynamically to realize different application functionalities presents challenges. In this context, it is natural to develop a framework that can customize the operation of ICPS to meet the varying needs of applications and users, based on an observe-analyze-adapt philosophy [6]. The xTune formal modeling and analysis framework can be extended to support specification of properties at both infrastructure and application levels, including multidimensional QoS properties and the relationships among them. Tuning processes at both the application and infrastructure levels can use compositional optimization to derive and validate the tuning and adaptation factors (sensors, policies, parameters).

Among many crosscutting concerns (e.g., security, privacy), let us take an example of cross-layer and end-to-end dependability issues. ICPS should be dependable despite disruptions/failures in sensing, communication, and computation.

Dependability of ICPS thus includes attributes such as availability, reliability, maintainability, safety, and integrity [15]. Realizing dependability requires monitoring and management of parameters at different layers of the system. Composition of nonfunctional needs such as dependability cannot be addressed in a single layer or device due to the inherent dependencies/trade-offs among them (e.g., techniques at any layer to improve dependability usually have implications on timing and power.). At the infrastructure level, a broad array of devices is interconnected by various communication channels (e.g., Ethernet, cellular, Wi-Fi) with distributed middleware support to execute cyber-physical applications. We view each device as a vertically layered architecture consisting of application, middleware, network, OS, and hardware layers. At each layer, the system can enforce policies that are (i) independent of other layers, (ii) a vertical composition of policies on the device across layers, and (iii) a horizontal composition of policies distributed across nodes.

To illustrate dependability across layers, consider the following example. If the data has high importance with a short expiration time, the middleware layer must adjust the frequency of dissemination appropriately. Similarly, CPU slowdown to control thermal runaway (hotspot) at the hardware layer may increase deadline misses in the OS task scheduling layer; this anomaly bubbles up to the application layer and is manifested as a failure to provide up-to-date data. Furthermore, deadline misses may lead to the delayed delivery of the network packets, which in turn results in a failure for timely delivery of messages. From a dependability perspective, both permanent and transient errors need to be modeled and mitigated. For instance, heavy utilization of the device hardware (e.g., for peak performance) can result in excessively high temperatures that may cause thermal errors; to alleviate this, we may trigger task replication or re-execution at the OS layer. The mitigation strategy might cause packet loss due to buffer overflow, since it requires more processing time. Under such circumstances, the dynamic choice of routing algorithms and their parameters needs to consider higher-layer QoS constraints, (partial) knowledge about the network (e.g., sensor density, coverage), heterogeneous devices (with different error sources), and operational context (e.g., prioritizing information flow).

5.3 Physical Infrastructure Protection

Physical infrastructure availability relies on the process control systems that can gather, handle, and share real-time data on critical processes from and to networked entities. For example, wireless sensor networks are now being applied in the industrial automation to lower systems and infrastructure costs, improve process safety, and guarantee regulatory compliance. Harsh environments such as remote areas with potential toxic contamination where mobile ad hoc networks can be the only viable means for communications and information access often necessitate the use of mobile nodes (e.g., surveillance robots with camera and position-changing capability). Optimized control based on continuous observation is an integral part because availability is becoming a fundamental concern to reduce the vulnerabilities of such systems.

Let us take an example of a surveillance system, consisting of a collection of sensors deployed at fixed locations together with mobile nodes, that monitors critical national infrastructure by distributed sensing and actuating. Due to possible jamming attacks and mobility of nodes, the wireless sensors and mobile nodes need to communicate via opportunistic links that enable the sharing and evaluation of data such as video streams in the presence of unstable connectivity. The challenge here is enabling networked entities to respond to dynamic situations in an informed, timely, and collaborative manner so that the physical infrastructure can safely recover after a cyber-disruption. The idea of automated verification and configuration of situation- and resource-aware cross-layer security needs to be investigated since security goals at each layer can be counterproductive and even harmful.

Furthermore, the implementation of security goals is constrained by the available resources. Various solutions ranging from event-driven or on-demand power cycling to reduce transmission power are possible, but the security effects cannot be understood at a single layer. This is why security should be viewed as a multidimensional cross-layer objective for which reasonable trade-offs must be found in a situation- and resource-aware manner. The resources of the wireless sensors and mobile nodes need to be provisioned to ensure a certain level of security while avoiding the depletion of residual energy and avoiding congestion. This requires the dynamic configuration of individual (seemingly independent) techniques to compose the appropriate protection against attack situations while also making optimal use of resources. By supporting specification of security properties across layers and exploiting the composition methods among them, the response to cyber-disruption is adapted to the situation and resource constraints.

6 Concluding Remarks

We have elaborated on the need for a unified framework for analyzing, deriving, and validating cross-layer adaptations for mobile applications operating in highly dynamic environments. Specifically, we have presented the design principles and implementation of the xTune framework [19]. We have developed formal analytical methods for understanding cross-layer optimization issues in mobile real-time embedded systems that incorporate resource-limited devices, and to integrate these methods into the design and adaptation processes for such systems. We have focused on the primary problem of identifying how to tune policies and parameters for cross-layer adaptation that aims to manage resource usage, and to satisfy the multifaceted constraints while providing a sufficient level of QoS with a verifiable/quantifiable solution quality.

We have presented our approach of iterative system tuning for mobile real-time embedded systems that has been applied in a case study treating the video-phone mode of a multimode multimedia terminal. The integration of lightweight formal methods with the observation of dynamic system execution results in a feedback loop that includes the formal models, simulation, and monitoring of running systems. Within the xTune framework, we proposed compositional cross-layer optimization to achieve robust and sufficiently good parameter settings

with low overhead by coordinated interaction among local optimizers through refinement of constraints that can be used further as a basis of local optimization.

The underlying formal executable models are moderately simple to develop, and their analysis is feasible. The experiments on a fairly complex case study demonstrate the applicability of our framework to cross-layer adaptation of mobile real-time embedded systems. The work on xTune complements our previous work on experimentally based cross-layer strategies (FORGE [3]) and conclusively shows that the xTune framework provides a uniform methodology for deriving, analyzing, and validating cross-layer adaptation.

The xTune framework essentially combines simulation, monitoring, and execution with formal methods. Lightweight formal analysis seems sufficient for multimedia applications in general. However, in the presence of mission-critical applications, context awareness and situation awareness (e.g., live video feed should be undisturbed in case of emergency evacuation) need to be further explored. Even though our current study using the xTune framework has produced encouraging results, the discussions in Section 5 present strong motivation for future work as mentioned in the sample application domains.

References

1. Agha, G., Meseguer, J., Sen, K.: PMAude: Rewrite-based specification language for probabilistic object systems. In: 3rd Workshop on Quantitative Aspects of Programming Languages, QAPL 2005 (2005)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
3. Forge Project, <http://forge.ics.uci.edu>
4. Hua, S., Qu, G., Bhattacharyya, S.S.: Energy reduction techniques for multimedia applications with tolerance to deadline misses. In: Proceedings of the 40th Conference on Design Automation (DAC 2003), pp. 131–136 (2003)
5. Kim, M., Dutt, N., Venkatasubramanian, N.: Policy construction and validation for energy minimization in cross layered systems: A formal method approach. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006) Work-in-Progress Session (2006)
6. Kim, M., Massaguer, D., Dutt, N., Mehrotra, S., Ren, M.-O.S.S., Talcott, C., Venkatasubramanian, N.: A semantic framework for reconfiguration of instrumented cyber physical spaces. In: Workshop on Event-based Semantics (WEBS 2008), CPS Week (2008)
7. Kim, M., Stehr, M.-O., Kim, J., Ha, S.: An application framework for loosely-coupled networked cyber-physical systems. In: 8th IEEE/IFIP Int. Conf. Embedded and Ubiquitous Computing, EUC 2010 (2010)
8. Kim, M., Stehr, M.-O., Talcott, C.: A distributed logic for networked cyber-physical systems. In: IPM Int. Conf. Fundamentals of Software Engineering, FSEN 2011 (2011)
9. Kim, M.-Y., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: Combining Formal Verification with Observed System Execution Behavior to Tune System Parameters. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 257–273. Springer, Heidelberg (2007)

10. Kim, M.-Y., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: A Probabilistic Formal Analysis Approach to Cross Layer Optimization in Distributed Embedded Systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 285–300. Springer, Heidelberg (2007)
11. Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: Constraint refinement for online verifiable cross-layer system adaptation. In: Design, Automation and Test in Europe Conference and Exposition, DATE 2008 (2008)
12. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. *Science* (4598), 671–680 (1983)
13. Mohapatra, S., Cornea, R., Oh, H., Lee, K., Kim, M., Dutt, N.D., Gupta, R., Nicolau, A., Shukla, S.K., Venkatasubramanian, N.: A cross-layer approach for power-performance optimization in distributed mobile systems. In: IEEE 19th International Parallel and Distributed Processing Symposium, IPDPS 2005 (2005)
14. Mohapatra, S., Dutt, N., Nicolau, A., Venkatasubramanian, N.: Dynamo: A cross-layer framework for end-to-end QoS and energy optimization in mobile handheld devices. *IEEE Journal on Selected Areas in Communications* 25(4), 722–737 (2007)
15. IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance, <http://www.dependability.org/wg10.4/>
16. Sen, K., Viswanathan, M., Agha, G.: Statistical Model Checking of Black-Box Probabilistic Systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
17. Stehr, M.-O., Kim, M., Talcott, C.: Toward Distributed Declarative Control of Networked Cyber-Physical Systems. In: Yu, Z., Liscano, R., Chen, G., Zhang, D., Zhou, X. (eds.) UIC 2010. LNCS, vol. 6406, pp. 397–413. Springer, Heidelberg (2010)
18. Stehr, M.-O., Talcott, C., Rushby, J., Lincoln, P., Kim, M., Cheung, S., Poggio, A.: Fractionated software for networked cyber-physical systems: Research directions and long-term vision. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Festschrift to the Honor of Carolyn Talcott. LNCS, vol. 7000, pp. 111–144. Springer, Heidelberg (2011)
19. xTune Framework, <http://xtune.ics.uci.edu>
20. Younes, H.: Ymer: A statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005), <http://www.tempastic.org/ymer>
21. Yuan, W., Nahrstedt, K.: Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In: 9th ACM Symposium on Operating Systems Principles (SOSP 2003), pp. 149–163. ACM Press (2003)
22. Yuan, W., Nahrstedt, K., Adve, S.V., Jones, D.L., Kravets, R.H.: Grace-1: Cross-layer adaptation for multimedia quality and battery energy. *IEEE Transactions on Mobile Computing* 5(7), 799–815 (2006)

From Service Identification to Service Selection: An Interleaved Perspective

Devis Bianchini¹, Francesco Pagliarecci², and Luca Spalazzi²

¹ Dipartimento di Ingegneria dell'Informazione
Universita' degli Studi di Brescia, Via Branze, 38, 25123 Brescia
bianchin@ing.unibs.it

² Dipartimento di Ingegneria Informatica, Gestionale e dell'Automazione
Universita' Politecnica delle Marche, Via Brece Bianche, 60131 Ancona
{pagliarecci,spalazzi}@univpm.it

Abstract. Business process implementation can be fastened by identifying component services that can be used to implement one or more process tasks and by selecting them from a repository of already implemented services. In this paper, we provide an iterative procedure to address this issue, by combining the two macro-phases of service identification and service selection. Starting from a workflow-based specification of the business process, service identification is firstly executed. The result of this phase is a decomposition tree, where basic process tasks are progressively organized into sub-processes (the candidate services) by applying an agglomerative clustering algorithm, based on cohesion and coupling metrics. Within the decomposition tree, a set of candidate services that minimize the coupling/cohesion ratio for the overall process is chosen. The service selection phase works on this decomposition and looks for available services. If the service selection phase fails for some candidate services, a revised set of candidate services is selected by leveraging on the decomposition tree.

1 Introduction

Business process implementation can be fastened by identifying component services that can be used to implement one or more process tasks and by selecting them from a repository of already implemented services. Service identification is a debated topic in the recent literature. It is defined as a procedure which starts from the business process specification and decomposes it into candidate component services, that can be used to implement one or more process tasks [1,4]. Candidate services can be either retrieved among existing ones or implemented from scratch. According to this perspective, service identification comes before service selection or service implementation and precedes service invocation and deployment. Service identification and service selection are mainly considered as distinct phases, sequentially executed.

In this paper, we investigate a different perspective, where the service identification and selection phases are more interleaved. Starting from the methodology

presented in [4], service identification is firstly executed. The result of this phase is a decomposition tree, where basic process tasks are progressively organized into sub-processes (the candidate services) by applying an agglomerative clustering algorithm, based on cohesion and coupling metrics inspired by analogous metrics in the software engineering field. Within the decomposition tree, a set of candidate services that minimize the coupling/cohesion ratio for the overall process is chosen. The service selection phase starts from this decomposition and looks for available services. If the service selection phase fails for some candidate services, a feedback is propagated back to the service identification phase, that proposes a revised set of candidate services by relying on the decomposition tree. The result is a business process decomposition that takes into account both given guidelines for service design, such as high cohesion and low coupling (*to-be* perspective) and concrete services actually available and already implemented, among which candidate services must be selected (*as-is* perspective). The main contribution of this paper is the proposal of the decomposition tree structure and of an algorithm based on the tree to enable a combined application of the service identification and selection phases.

The paper is organized as follows: in the next section some related work are discussed; in Section 3 we present the problem statement with the help of an application scenario; Section 4 briefly introduces the semantic annotation of processes; Section 5 summarizes the service identification phase and introduces the decomposition tree; Section 6 describes the service selection phase; Section 7 describes the iterative combination of the two phases; finally, Section 8 closes the paper.

2 Related Work

Service identification. Ghosh et al. [14] listed three kinds of service identification approaches:

- *top-down*, also known as *domain decomposition*, which focuses on analysis of business domains and business process modeling to identify services, components and flows; key elements are the business entities and business functionalities and roles responsible for those entities; process modeling and decomposition offer also the opportunity not only to identify services, but also the service flows that will be used to orchestrate them;
- *bottom-up*, IT-centric approach focusing on discovery and characterization of existing IT assets and services; quantitative metrics such as cohesion, coupling, service reusability are used to evaluate the quality of existing assets and, eventually, to perform reengineering strategies;
- *meet-in-the-middle*, referred to as *goal-service modeling*, where a generalized statement of business goals relevant to the scope of the business process is decomposed into subgoals that must be met by existing IT assets and services.

In [29] a top-down methodology in which several methods are combined in order to identify services starting from the analysis of organizational domain and processes is proposed. Authors guide the service designer by suggesting the order through which the different techniques should be used and provide some tips for the evaluation of the results. In [18] authors define a set of heuristics to support service identification, but do not propose any quantitative models to efficiently support the analysis.

Examples of bottom-up approaches have been described in [15,23,28,30]. In [30] the authors design and execute business processes by using Web services. To define a Web service as part of a process, the main issue is the identification of the actual Web services that match the specification of the designer. Furthermore, taking a conceptual modeling approach, the relationships between ontology concepts and syntactic Web services should be identified. The enrichment of the process description is also addressed by [15], in which authors state that businesses lack a machine readable representation of their process space as a whole on a semantic level. They explain how the use of an appropriate language (i.e., WSMO) can provide a unified view on business processes in a form that allows querying process spaces by logical expressions and easily link process activities to services. In [23] an abstract process represents a Web process whose control and data flow are defined at design time, but the actual services are not chosen until run-time. Run-time service selection can be automated with the semantic representation of the knowledge of the domain experts in ontologies and rules (semantic Web processes). In [28] the notion of process template is introduced. Process templates are reusable business process skeletons that are devised to reach particular goals and are made up of states and transitions. A state corresponds to the execution of a service (called component service) that is member of a Web service community. A community is a collection of services with a common functionality, but different non functional properties such as different QoS parameters, that are exploited to select the most suitable service at run-time.

The recommendation given in [14] is to follow a meet-in-the-middle strategy. The last step is a service refactoring and rationalization step, based on the Service Litmus Test. The refactoring is performed by grouping lower-level services that have some kind of logical affinity. Subsequently, the SLT (rationalization) is applied as a set of criteria to resolve whether a candidate service should be exposed, based on the evaluation of business alignment, composability, externalized service description, redundancy elimination. However, bottom and meet-in-the-middle strategies are especially useful in environments where component services are relatively fixed and processes are designed on the basis of the available services.

Service lifecycle is thoroughly presented in [26], which discusses how business processes should be described so that services can be properly identified and provides strategies and principles regarding functional and non-functional aspects of Web service design. Furthermore, in [25] authors propose a methodology that aims at defining a foundation of development principles for Web services based on which business processes can be assembled into business scenarios. Birkmeier

et al. [5] classify existing approaches for systematic service identification along a series of dimensions that include availability of procedural guidelines, development direction (top-down, bottom-up, meet-in-the-middle), use of quantitative techniques, metrics to evaluate the phases of service identification procedure, tool support, quality assessment and validation techniques. The result of the analysis given in [5] asserts that none of the existing approaches fulfills all the criteria.

Service Selection. Service-Oriented Architectures follow the find-bind-execute paradigm in which service providers register their services in public or private registries, which clients use to locate web services. Service selection mechanisms plays an essential role in Service-Oriented Architectures, because most of the applications want to use services that accurately meet their requirements. The increasing availability of Web services that offer similar functionalities with different characteristics increases the need for more sophisticated selection processes to match user requests. Most of the existing techniques rely on syntactic descriptions of service interfaces to find web services with disregard to non-functional service parameters. Previous research demonstrates how this situation generates major problems [20,22]. To solve some of problems, Web service descriptions are enhanced with annotations of ontological concepts, semantic matching and by considering non-functional properties (NFPs) [8,9]. Several QoS based service selection techniques have been investigated [34,33]. The service selection problem is investigated in [34] by using a combinatorial model and a graph model. In [33], a formal service model is defined and then a dynamic programming based approach is proposed to select the best service providers.

Our Contribution. With respect to existing approaches, we follow a meet-in-the-middle perspective where the service identification phase is strictly related to the service selection one. In particular, the decomposition tree structure proposed in this paper is meant to enable a better interaction of these phases, ensuring the maximal exploitation of existing implemented services from one hand and the selection of a set of services that are featured as much as possible by high cohesion and low coupling, according to recommended guidelines in the literature.

3 Problem Statement

Processes are usually expressed using a workflow-based notation (e.g., BPMN [7]), independent from implementation technologies and platforms. A business process \mathcal{BP} can be represented as a set of simple tasks, combined through control flow structures (e.g., sequence, choice, cycle or parallel) and specified through the performed operations and I/O data flow between them. We define an I/O parameter as a pair $\langle n, \mathcal{P} \rangle$, where n is the I/O name and $\mathcal{P} = \{p_i\}$ a set of I/O properties. Collaborative business processes are designed as processes spanning over different actors. Actors are represented as abstract entities that interact

each other as responsible of one or more simple tasks. Beyond the platform-independent implementation of the collaborative business process \mathcal{BP} , we consider a repository of implemented services $\overline{\mathcal{S}}$, represented at the process level (based on BPEL) with minimal semantic annotations and a language that can express requirements on the behavior of the service that has to be selected. The business process \mathcal{BP} can be decomposed into a set of subprocesses \mathcal{S} , that can be totally or partially implemented through the services in the repository. Let's denote with \mathcal{S}' the subprocesses which can be implemented through one of the services in the repository. The problem we address in this paper is the identification of the set \mathcal{S} such that the exploitation of the available services $\overline{\mathcal{S}}$ in the repository is maximized, that is: $\mathcal{S}' \subseteq \mathcal{S}$ and it does not exist another $\mathcal{S}'' \subseteq \mathcal{S}$ such that $\mathcal{S}' \subseteq \mathcal{S}''$ (Constraint 1). The set \mathcal{S}' is the maximum set of component services which have a corresponding service implemented in the repository. On the other hand, as highlighted in [4], component services are a particular kind of sub-process, where the following properties hold: (i) services are self-contained and interact each other using decoupled message exchanges, that is, present high cohesion and low coupling (Constraint 2); (ii) each service is the minimal set of tasks that performed together create an output that is a tangible value for one of the actors involved in the overall process execution (Constraint 3). A value has been defined as an intermediary process output that is not used as input of another task of the same actor in the process (for example, in the considered case study, the invoice issued by the external supplier is a value for the retailer). All the constraints 1, 2 and 3 must be pursued as much as possible.

As a case study, we consider an application from the computer supplying domain. A computer retailer receives computer orders and, after their approval, generates the bill of materials (BOM) and sends orders for components to his/her suppliers. Some suppliers have an arrangement with the retailer, while other suppliers are external to the retailer business network. The procedure to order components is different depending on the type of the suppliers. In particular, for external ones, an invoice is prepared and sent to the retailer. After receiving all the components, the retailer assembles the computer, prepares the final invoice and ships the product to the client. The described process is shown in the upper part of Figure 1. In the process, different candidate component services can be identified taking into account quality criteria such as internal service cohesion and coupling between distinct services [31]. Among the identified component services, redundancies can be detected. For example, the $\{\text{OrderComponent}, \text{ShipComponent}\}$ and the $\{\text{SalesOrder}\}$ subprocesses perform similar functionalities and can be implemented by the same service. After identification, candidate services must be selected from a repository of available ones. The decomposition shown in Figure 1 not always reflects the services available in the repository. For example, if the $\{\text{IssuePurchase}, \text{Approved}\}$ subprocess cannot be found as an implemented service in the repository, an efficient procedure is necessary to relax the decomposition with respect to high cohesion and low coupling criteria, by further splitting the considered process (this could decrease coupling) or by aggregating it with other tasks (this could decrease

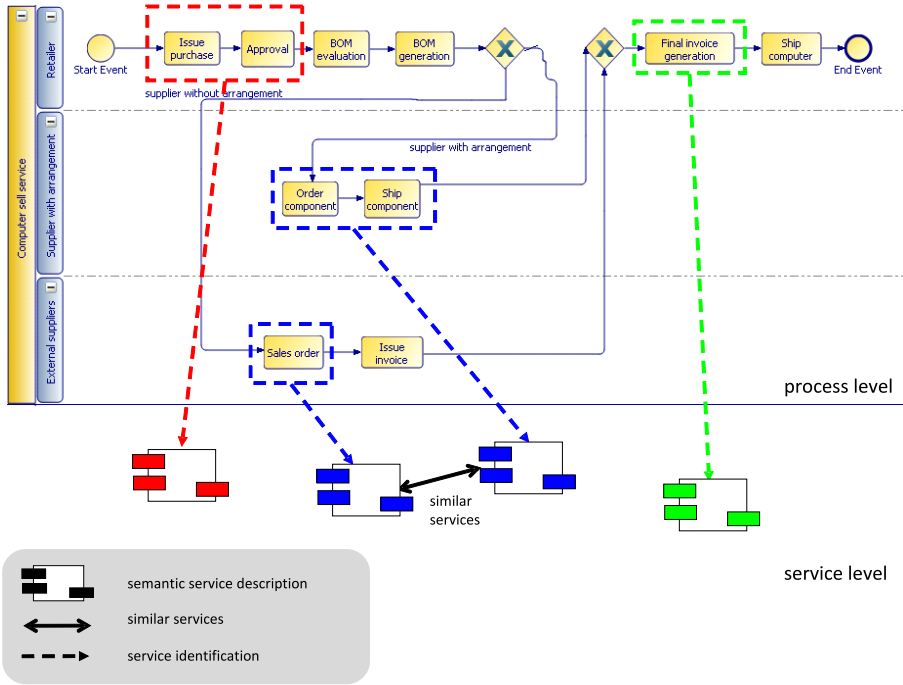


Fig. 1. Running example

cohesion) in order to find services in the repository which implement the new subprocess. In the following, we describe how the identification and selection phases are performed and how they interact to address these issues.

4 Semantic Annotation of Business Processes

Automatic service identification and selection require that both the business process BP (represented using BPMN) and the services \bar{S} in the repository (represented using BPEL) are semantically described. We do this by extending the BPMN and BPEL with semantic annotations.

The semantics of the application domain is formally modelled as a *Domain Ontology* and, following the mainstream, this ontology is described by means of a language belonging to the *Description Logic* family [2]. In this work it has been used a subset of WSML [32], namely WSML-Core that is a subset of *SHIQ(D)*. The ontology consists of all the terms that can be used and are relevant for the application domain to be modeled. It should be a unique, commonly accepted formalization of the given domain. This is indeed a strong assumption, but it is supported by the recent trend. As a matter of fact, several companies and organizations have already defined standard ontologies for some specific domains

(e.g. airline and travel companies [24], economics and accounting [17], cultural heritage information [16], geopolitical information [13]) or are defining them (e.g. railway systems [19]). Nevertheless, many organizations have their own ontologies. Therefore, a problem of ontology matching or ontology integration arises. In the field of Artificial Intelligence, these problems have received a great deal of attention, but they are out of the scope of this paper. In the rest of the paper it is assumed that there is just one shared domain ontology.

The semantic annotations to link Functionality and Behavior of a given business process or service with ontology elements (WSML) are defined according to the *Semantic Web Service Annotation Language* (SWSAL) proposed in [10,11]. The language is based on XML and the links are represented by means of XPath expressions. From a theoretical point of view, it belongs to the assertional part of a Description Logic. This approach has several advantages. First of all, it preserves the original syntax of standard languages as BPMN or BPEL, thus allowing the service description to be compatible with the most popular development environments. Second, SWSAL can be easily extended to semantically annotate any kind of XML-based process or service description language, e.g. WS-CDL, WS-Policy, and so on. Third, SWSAL allows to annotate only what it is needed. Finally, as presented in [6], it has been developed a plug-in for Eclipse that allows to perform the annotation by means of a graphical interface. This tool allows to load and browse an ontology, to draw a BPEL or BPMN diagram, and to annotate them by means of a “drag-and-drop” action.

5 Service Identification

A service identification methodology, called P2S (Process-to-Services) methodology, has been extensively described in [4]. The core phases of the P2S methodology are the following:

- *semantic process annotation* - in a distributed heterogeneous environment, where different IEs provide independently developed process representations, business process elements (inputs and outputs, task names) must be semantically annotated with concepts extracted from shared ontologies;
- *identification of candidate services* - candidate component services must be identified ensuring the same decomposition granularity, thus enabling better service comparison for sharing and reuse purposes;
- *reconciliation of similar services* - component services must be clustered on the basis of the similarity of their tasks and I/O data, in order to identify similar services on different processes and enable the design of reusable component services (for example, the {`OrderComponent`,`ShipComponent`} and the `SalesOrder` subprocesses in the application scenario).

In particular, to enable homogeneous identification of services by analyzing process description, the identification of candidate services is performed in two steps: (i) value-based candidate service identification, in which value exchanges are

identified throughout the process flow and the process is split into a preliminary list of candidate services; (ii) refinement of process decomposition through service cohesion/coupling evaluation. The goal of the P2S methodology was to support the identification of candidate services to guide their design according to best granularity, cohesion, coupling and reuse criteria. The result was a portfolio of identified services, to be discovered among available ones within public or private registries or to be developed from scratch. Therefore, the perspective of the P2S methodology is the one of a top-down approach, according to the definition given in Section 2.

In this paper, we work under a different perspective, where we try to exploit the set of concrete services actually available and already implemented, among which identified services must be selected, ensuring at the same time as much as possible the guidelines for service design suggested by the P2S methodology. Our perspective in this paper is a meet-in-the-middle one, where we will relax the optimal decomposition of the original business process to meet the real availability of concrete services. To this aim, we will rely on a decomposition structure, called *decomposition tree*, based on cohesion and coupling metrics: this structure will guide the service selection phase which will follow the service identification one. Hereafter, we will focus on the cohesion and coupling evaluation. Value-based analysis and reconciliation of similar services, that are core phases of the P2S methodology, can be integrated with the phases described in this paper as well, but are not addressed here and are left as future work. We proceed now by summarizing the cohesion and coupling metrics detailed in [4].

5.1 Cohesion and Coupling Evaluation

As for the P2S methodology, the business process must be semantically annotated to identify correspondences among heterogeneous task names and I/Os. The ontological (semantics) part of the specification is expressed here by means of semantic annotations written in SWSAL [10], as explained in Section 4.

As explained above, actors participating in the collaborative business process agree on a common Domain Ontology, that provides atomic concept definitions and equivalence/subsumption relationships between them. However, in a distributed and heterogeneous environment, local terms used by different actors for business process elements do not necessarily coincide with atomic concepts in the Domain Ontology or they may suffer from terminological discrepancies (e.g., synonymies or homonymies). To solve these heterogeneities, the Domain Ontology is extended with a Thesaurus, extracted from a lexical system (e.g., WordNet [12]), where terms are related each other and with the names of ontological concepts by means of terminological relationships. A weight $\sigma_{rel} \in (0, 1]$ is associated with each kind of relationship. The following terminological relationships are defined: (i) *synonymy* (SYN), with $\sigma_{SYN} = 1.0$, established between two terms that can be used in an interchangeable way in the process (e.g., **ShippingAddress** SYN **Address**); (ii) *narrower/broader term* (BT/NT), with $\sigma_{BT/NT} = 0.8$, established between a term n_1 and another term n_2 that has a more generic (resp., specific) meaning (e.g., **InvoicedQuantity** NT **Quantity**); (iii) *related term* RT,

with $\sigma_{RT} = 0.5$, established between two terms whose meaning is related in the considered application scenario (e.g., **Order RT Invoice**). In [3] techniques apt to guide the process designer in the construction of the Thesaurus and its combined use with a Domain Ontology are detailed. Starting from process descriptions, it is possible to make semantic analysis in order to identify similarity correspondences between inputs requested in a given task and outputs provided in another task. These correspondences are the basis for the identification of component services. Specifically, ontological and terminological relationships are used to define different kinds of affinity functions applied to process elements, as formally defined in Table 1.

Table 1. Name and structural affinity coefficients

NAME AFFINITY FUNCTION
$NAff(n_1, n_2) = \begin{cases} 1 & \text{if } n_1 = n_2 \\ \max_m(\rho(n_1 \rightarrow^m n_2)) & \text{if } n_1 \neq n_2 \wedge \max_m(\rho(n_1 \rightarrow^m n_2)) \geq \alpha \\ 0 & \text{otherwise} \end{cases}$
where α is an affinity threshold, that is used to filter out names with high affinity values
STRUCTURAL AFFINITY FUNCTION
$SAff(d_1, d_2) = \frac{1}{2} \cdot \left[NAff(n_1, n_2) + \frac{2 \cdot \sum_{p_1, p_2} NAff(p_1, p_2)}{ P_1 + P_2 } \right] \in [0, 1]$
where $d_1 = \langle n_1, P_1 \rangle$ and $d_2 = \langle n_2, P_2 \rangle$ (either input or output of simple tasks)

Definition 1 (Name Affinity). *Given two terms n_1 and n_2 used as names of I/O parameters, I/O properties or process tasks, the Name Affinity $NAff$ between n_1 and n_2 is defined on the basis of the existence of a path of m terminological relationships between n_1 and n_2 , denoted with $n_1 \rightarrow^m n_2$. In particular, $NAff$ is based on the strength of the path $n_1 \rightarrow^m n_2$, denoted with $\rho(n_1 \rightarrow^m n_2)$, computed as the product of the weights associated to the relationships belonging to the path itself.*

Definition 2 (Structural Affinity). *Given a pair of I/O parameters $d_1 = \langle n_1, P_1 \rangle$ and $d_2 = \langle n_2, P_2 \rangle$, the Structural Affinity function combines the affinity between the I/O names with the affinity between each pair of properties $p_1 \in P_1$ and $p_2 \in P_2$, as shown in Table 1.*

The total Structural Affinity $Aff_{TOT}(D_1, D_2)$ between two sets of I/O data D_1 and D_2 is defined as the sum of Structural Affinity for each pair of items $d_1 \in D_1$ and $d_2 \in D_2$, normalized with respect to the cardinality of D_1 and D_2 .

Name and Structural Affinity are the basis for evaluating the tasks cohesion and coupling. The adopted cohesion/coupling metrics have been inspired by their well-known application in software engineering field [21,31]. They have been detailed in [4] and are summarized in Table 2. The cohesion coefficient evaluates how much tasks within a single service contribute to obtain a service output. The coupling coefficient evaluates how much tasks belonging to different services need to interact.

Table 2. Cohesion and coupling coefficients

TASK DEPENDENCY COEFFICIENT	
$\tau(t_i, t_j) = \begin{cases} Aff_{TOT}(OUT(t_j), IN(t_i)) & \text{if } t_j \mapsto t_i \\ Aff_{TOT}(IN(t_j), OUT(t_i)) & \text{if } t_i \mapsto t_j \\ \frac{Aff_{TOT}(IN(t_i), IN(t_j)) + Aff_{TOT}(OUT(t_i), OUT(t_j))}{2} & \text{if } t_i t_j \\ 0 & \text{otherwise} \end{cases}$	
where: $t_i \neq t_j$, $t_i \mapsto t_j$ means that there is a data dependency from t_i to t_j (see [4] for a formal definition of data dependency between tasks) and $t_i t_j$ means that t_i and t_j are executed in two parallel branches of the business process.	
SERVICE COHESION COEFFICIENT	
$coh(\mathcal{S}) = \begin{cases} 2 \cdot \frac{\sum_{i,j} \tau(t_i, t_j)}{ \mathcal{S} \cdot (\mathcal{S} - 1)} \quad \forall t_i, t_j \in \mathcal{S} & \mathcal{S} > 1 \\ 1 & \mathcal{S} = 1 \end{cases}$	
where $ \mathcal{S} $ is the number of tasks in \mathcal{S}	
SERVICE COUPLING COEFFICIENT	
$coup(\mathcal{S}_1, \mathcal{S}_2) = \frac{\sum_{i,j} \tau(t_i, t_j)}{ \mathcal{S}_1 \cdot \mathcal{S}_2 } \quad \forall t_i \in \mathcal{S}_1 \wedge t_j \in \mathcal{S}_2, \mathcal{S}_1 \neq \mathcal{S}_2$	

5.2 Decomposition Tree

Starting from the definition of tasks cohesion and coupling, our identification methodology applies an agglomerative clustering of tasks according to their coupling. The result is a *decomposition tree* (DT), where leaves are single tasks and intermediate nodes are subprocesses collecting tasks which present high coupling. The decomposition tree is constructed as follows. The agglomerative clustering algorithm starts from the single tasks that are the leaves of the tree and merges first those tasks that present the highest coupling. Merging at intermediate levels is performed similarly: two subprocesses are merged if these are two tasks, one from the first subprocess and one from the second subprocess, which present the highest coupling. Merging continues until all the tasks are grouped together, that is, the root of the tree represents the overall process. The decomposition tree for the running example is shown in Figure 2 and can be formalized as follows.

Definition 3 (Decomposition Tree). *A decomposition tree DT is defined as*

$$DT = \langle r, \Sigma, \Lambda \rangle \tag{1}$$

where r is the root of the decomposition tree, that is, the initial collaborative business process \mathcal{BP} , Σ is the set of all the tree nodes, which are possible subprocesses of \mathcal{BP} , $\Lambda = \Sigma \times \Sigma$ is a set of edges between the tree nodes, such that $(\mathcal{S}_1, \mathcal{S}_2) \in \Lambda$ means that \mathcal{S}_2 is a subprocess of \mathcal{S}_1 or, equivalently, \mathcal{S}_1 is decomposed, amongst the others, into \mathcal{S}_2 .

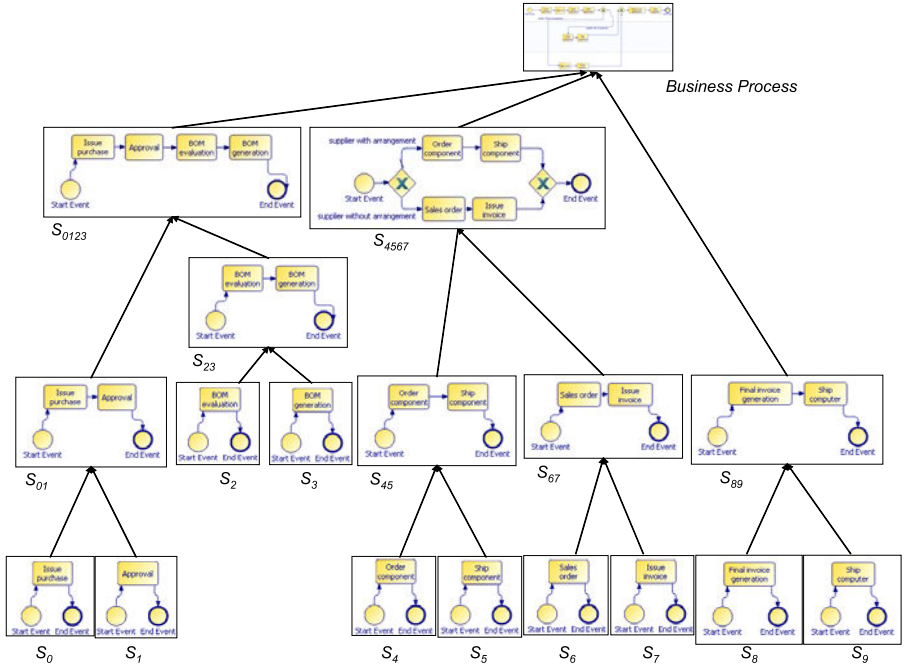


Fig. 2. Decomposition tree for the running example

The nodes of the decomposition tree are all potential services that could be identified. Given a decomposition of the business process \mathcal{BP} , we define the coupling/cohesion ratio Γ as

$$\Gamma = \frac{pcoup(\mathcal{BP})}{pcoh(\mathcal{BP})} = \frac{\sum_{i,j} coup(S_1, S_2)}{|\mathcal{BP}| \cdot (|\mathcal{BP}| - 1)} = \frac{\sum_{i,j} coup(S_1, S_2)}{(|\mathcal{BP}| - 1) \sum_i coh(S_i)} \quad (2)$$

where $|\mathcal{BP}|$ represents the actual number of candidate component services in the process. The best decomposition presents the minimum value for Γ . Each time a node in the decomposition tree is decomposed into its children, there is a variation in the ratio Γ . We define a *Weighted Decomposition Tree* WDT as a decomposition tree where each node is weighted with the variation of ratio Γ caused by the split of the node into its children.

Definition 4 (Weighted Decomposition Tree). A *Weighted Decomposition Tree* WDT obtained starting from a $\mathcal{DT} = \langle r, \Sigma, \Lambda \rangle$ is defined as follows:

$$WDT = \langle r, \mathcal{W}\Sigma, \Lambda \rangle \quad (3)$$

where $\mathcal{W}\Sigma = \mathbb{R} \times \Sigma$; given an element $\langle w, S \rangle \in \mathcal{W}\Sigma$, w is the delta of ratio Γ associated to the decomposition of $S \in \Sigma$. The leaves nodes are associated to a weight $w=0$, since they cannot be further decomposed.

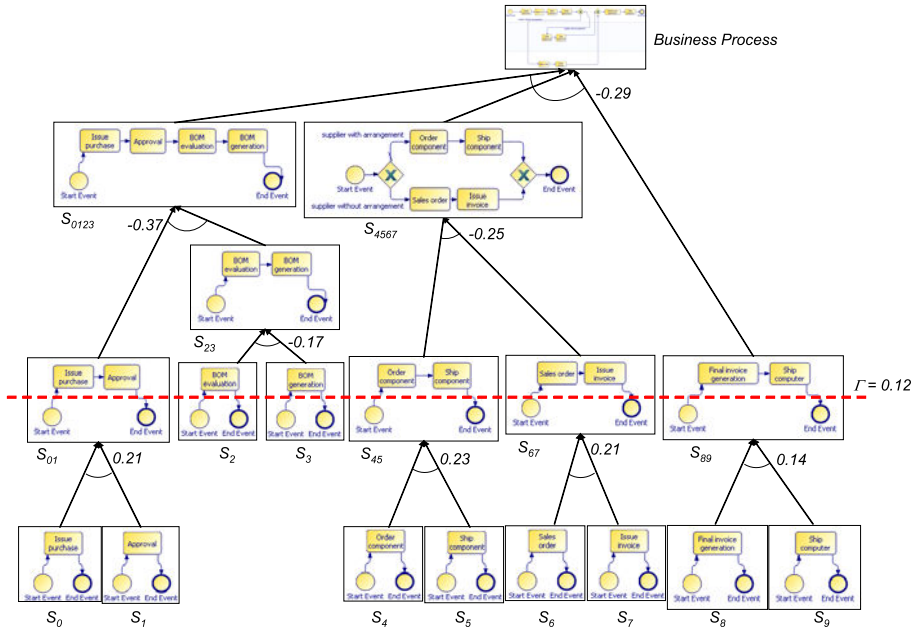


Fig. 3. Weighted decomposition tree for the running example

Figure 3 shows the *WDT* associated to the *DT* in Figure 2. A negative weight means that the split of the node into its children decreases the ratio Γ . For example, if the node \mathcal{S}_{4567} is split into its children \mathcal{S}_{45} and \mathcal{S}_{67} the ratio Γ is decreased by 0.25. A positive weight means that the split of the node into its children increases the ratio Γ . If the children are aggregated into their parent node, the variation of the ratio Γ is equal to the weight associated to the parent node considered with negative sign. For example, if nodes \mathcal{S}_{45} and \mathcal{S}_{67} are aggregated to form \mathcal{S}_{4567} , the ratio Γ is equal to $-(-0.25)$, that is, Γ is increased by 0.25. The best set of candidate component services (highlighted through a dashed line in Figure 3) is identified by the P2S methodology by minimizing the ratio Γ . In the following, we will show how this optimal solution can be relaxed by exploiting the structure of *WDT* to take into account the real availability of implemented services to be selected. Decisions on how to move throughout the *WDT* are made taking into account the weights associated to the nodes of the tree.

6 Service Selection

Generally speaking, service selection consists in finding, among a set of services, the service that satisfy certain requirements. This problem can be considered by two different perspectives: a functional perspective and a structural (process level) perspective. According to the *functional perspective*, each service is described by its functionality and quality of service, its preconditions and effects,

its input and output. Unfortunately, this knowledge is not enough to understand whether a given service can be used to implement a certain business (sub-)process. Indeed, that process is known only through its behavior. According to the *structural (process level) perspective*, each service is described in terms of its structure, its “behavior”; i.e. in terms of state transitions or activities performed. This is a sort of white-box description. This perspective starts from the observation that processes are stateful (each action depends on the state where the process is), their behavior may be nondeterministic (e.g. the search for an item to add to the cart may fail), and they may exchange messages asynchronously. Taking into account the above observations, our approach is therefore based on the following key ideas:

Specification - the requirements about the behavior is expressed by means of a BPMN diagram where tasks are semantically annotated (Section 4). This language allows a user to specify which tasks a service must implement and in which order they must be executed. Notice that this introduces a partial order relationship among tasks. Indeed, with such a kind of requirement, a user states that it does not care what the service to select does between a task and another one. This requirement can be easily (automatically) translated into a temporal logic specification. In our approach, we use the Computation Tree Logic (CTL) enriched with (concept and role) assertions of a Description Logic (assertions are used instead of atomic propositions as in the traditional CTL). This logic is called *Annotated CTL (AnCTL)* [27].

Service - service behavior is represented by means of the BPEL language where activities are semantically annotated (Section 4). This service behavior can be easily (automatically) modeled as a state transition system with semantic annotation. This kind of model is called *Annotated State Transition Systems (ASTS)* [27].

Semantic Model Checking - The problem of verifying whether a service (modeled as an ASTS) satisfies a specification (represented by an AnCTL formula) can be solved by means of the algorithm of Semantic Model Checking that has been defined in [27]. This algorithm has been proved to be sound and complete. Its complexity depends on the expressiveness of the description logic that has been used and ranges from PTIME to CONP.

As a consequence, the input of a *selection problem* consists of (1) a domain ontology (denoted by \mathcal{T}), (2) a set of annotated services described by means of their behavior, and (3) a specification, represented by the (annotated) process that must be implemented by the selected service. Formally, this problem can be defined as follows:

Definition 5 (Selection Problem). Let $\overline{\mathcal{S}}_1, \dots, \overline{\mathcal{S}}_n$ be a set of services annotated according to the same terminology \mathcal{T} . Let \mathcal{S} be a process. Let $\phi(\mathcal{S})$ the AnCTL formula derived by \mathcal{S} . Then the process-level service selection problem, denoted by $\sigma_{\mathcal{S}}(\overline{\mathcal{S}}_1, \dots, \overline{\mathcal{S}}_n)$, is the problem of finding an annotated service $\overline{\mathcal{S}}_i$ such that

$$\overline{\mathcal{S}}_i \models \phi(\mathcal{S}).$$

Notice that, $\phi(\mathcal{S})$ is obtained by the description of \mathcal{S} by means of appropriate translation rules, where \mathcal{S} is represented as BPMN process. Consider that, basically, a BPMN process is a diagram with activities: atomic tasks and sub-processes; and control structures: sequence, choice (called decision gateway), fork (called parallel gateway), and iterations. Therefore, the translation rules to derive the related *AnCTL* formula are reported in Figure 4. The translation rules

1. Let t be an atomic task. Let $ann(t)$ be the set of semantic annotations denoting the situation after the execution of t .
Then $\phi(t) = \text{AF}ann(t)$.
2. Let $a_1; a_2$ be the sequence of two activities a_1 and a_2 .
Then $\phi(a_1; a_2) = \text{AG}(\phi(a_1) \rightarrow \text{AF}\phi(a_2))$.
3. Let $a_1 \cup a_2$ be the decision gateway for two activities a_1 and a_2 .
Then $\phi(a_1 \cup a_2) = \text{EF}\phi(a_1) \wedge \text{EF}\phi(a_2)$.
4. Let $a_1 || a_2$ be the parallel gateway for two activities a_1 and a_2 .
Then $\phi(a_1 || a_2) = \text{AF}(\phi(a_1) \wedge \phi(a_2))$.
5. Let a^* be the iteration of activity a .
Then $\phi(a^*) = \text{AF}\phi(a)$.
Notice that $\phi(a)$ must be an invariant, i.e. it must hold at each step of the iteration.

Fig. 4. Rules for deriving an *AnCTL* formula from a BPMN file

of Figure 4 are sound, as stated by the following lemma:

Lemma 1. *Let \mathcal{S} be a process. Let $\phi(\mathcal{S})$ be the related *AnCTL* formula derived according to the rules of Figure 4. Then*

$$\mathcal{S} \models \phi(\mathcal{S}).$$

Proof (Hint).

This is a straightforward consequence of the operational semantics of a BPMN process (represented as a Kripke structure) and the semantics of *AnCTL* [27].

At this point, it is possible to apply the sound and complete semantic model checking algorithm defined in [27] to a given service $\overline{\mathcal{S}}_i$ in order to verify whether it satisfies the specification $\phi(\mathcal{S})$ (this is denoted as $\overline{\mathcal{S}}_i \vdash \phi(\mathcal{S})$). If so, that service can be considered a solution for the selection problem. This result is based on the following theorem.

Theorem 1 (Selection). *Let $\sigma_{\mathcal{S}}(\overline{\mathcal{S}}_1, \dots, \overline{\mathcal{S}}_n)$ be a selection problem. Let $\overline{\mathcal{S}}_i$ be a service such that*

$$\overline{\mathcal{S}}_i \vdash \phi(\mathcal{S}).$$

Then $\overline{\mathcal{S}}_i$ is a solution for the selection problem.

Proof (Hint)

This is a straightforward consequence of the soundness and completeness theorem for the semantic model checking algorithm reported in [27].

Algorithm

```

input :  $\Delta = \Sigma - \Phi$  /* Set of not found services */
output:  $\Phi'$  /* Set of found services */

{  $\Phi' := \Phi$ ;
  while  $\Delta \neq \emptyset$  do
    {  $\Upsilon := \emptyset$ ; /* Set of services to be found */
      for each  $S \in \Delta$ 
        {  $\Psi(S) := \emptyset$ ; /* Set of siblings of service  $S$  */
           $\Omega(S) := \emptyset$ ; /* Set of not found siblings of service  $S$  */
           $\Psi(S) := \text{SIBLING}(S)$ ;
          if  $(\Psi(S) \neq \emptyset) \wedge (\gamma_p > \gamma_c)$  do
            { for each  $\psi \in \Psi$ 
              if  $\psi \in \Delta$  do  $\Omega(S) := \Omega(S) \cup \{\psi\}$ ;
              if  $(\Omega(S) \equiv \Psi(S)) \wedge (\Omega(S) \subseteq \Delta)$  do
                {  $\Upsilon := \Upsilon \cup \text{PARENT}(S)$ ;
                   $\Delta := \Delta - \Omega(S)$  }
              }
            else if  $\neg \text{LEAF}(S) \wedge (\gamma_p < \gamma_c)$  do
              {  $\Upsilon := \Upsilon \cup \text{CHILDREN}(S)$ ;
                 $\Delta := \Delta - \{S\}$  }
            }
          SELECTION( $\Upsilon$ )
          {  $\Phi' := \Phi' \cup \Phi$ ;
             $\Delta := \Upsilon - \Phi$  }
        }
      }
    return  $\Phi'$ 
  }
}

```

Fig. 5. The algorithm

7 Feedback Strategy

The output of service identification phase is the list of candidate component services which minimize the ratio Γ and the weighted decomposition tree. In our scenario, the candidate component services are $\Sigma = \{\mathcal{S}_{01}, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_{45}, \mathcal{S}_{67}, \mathcal{S}_{89}\}$, as shown in the tree in Figure 3.

We use the BPMN workflow and the semantic annotations introduced in Section 4 to generate semantically annotated BPMN specifications of candidate component services, that we use to check if there is an implemented service in the repository which satisfies each specification (by applying the selection procedure presented in Section 6). If the Semantic Model Checking fails for some specifications, the algorithm proposed in Figure 5 is applied.

The algorithm implements two different strategies: *decomposition* and *aggregation*. Given a candidate component service \mathcal{S}_x that has no implementations in the repository, according to the decomposition strategy, the selection procedure

is applied to \mathcal{S}_x children in order to find them. On the contrary, according to the aggregation strategy, the selection procedure is applied to the parent node of \mathcal{S}_x . The choice among decomposition and aggregation strategies is made taking into account the weights on the Weighted Decomposition Tree.

Let \mathcal{S}_x be the candidate component service that we haven't found, \mathcal{S}_y its parent node, $\{\mathcal{S}_i\}$ the direct children of \mathcal{S}_x and $\{\mathcal{S}_j\}$ the other direct children of \mathcal{S}_y in the tree. We apply the decomposition strategy if the variation of ratio Γ between \mathcal{S}_x and \mathcal{S}_i (γ_c) is smaller than variation of ratio Γ between \mathcal{S}_x and \mathcal{S}_y (γ_p), \mathcal{S}_x is not a leaf node and, during a previous selection, we have find the service for the specifications connected with \mathcal{S}_j . In all the other cases, if γ_p is smaller than γ_c , \mathcal{S}_x is a leaf node and/or we haven't found the service for some specifications connected with \mathcal{S}_j , we apply the aggregation strategy.

In our scenario the service selection phase performed with Σ has found four services $\Phi = \{\overline{\mathcal{S}}_{01}, \overline{\mathcal{S}}_2, \overline{\mathcal{S}}_3, \overline{\mathcal{S}}_{89}\}$ and has not found any service that accurately meet with processes \mathcal{S}_{45} and \mathcal{S}_{67} . We apply the algorithm and, according to aggregation strategy, the output is $\Phi' = \{\overline{\mathcal{S}}_{01}, \overline{\mathcal{S}}_2, \overline{\mathcal{S}}_3, \overline{\mathcal{S}}_{4567}, \overline{\mathcal{S}}_{89}\}$. The algorithm stops if all the candidate component services have been found or the whole Weighted Decomposition Tree has been inspected.

8 Conclusion

The methodology presented in this paper aims at constituting a interleaved approach from service identification to service selection. The result is a meet-in-the-middle business process decomposition that takes into account both given guidelines for service design, such as high cohesion and low coupling, and the availability of concrete services already implemented, among which candidate services can be selected. Currently, single modules that implement the identification and selection phases have been developed and tested in [4] and [27], respectively. The test of their interleaved application in a real case scenario are being under development. The proposed approach will be the basis for an improved design tool that supports the business process designer in the identification and selection of services to implement a collaborative process, avoiding the development from scratch of all the process elements. it is necessary to integrate them and properly test the resulting application.

References

1. Amsden, J.: Modeling SOA: Part 1. Service identification. Technical report, IBM Technical report (2007), <http://www.ibm.com/developerworks/rational/library/07/-1002amsden/>
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
3. Bianchini, D., De Antonellis, V., Melchiori, M.: Flexible Semantic-based Service Matchmaking and Discovery. World Wide Web Journal 11(2), 227–251 (2008)

4. Bianchini, D., Cappiello, C., De Antonellis, V., Pernici, B.: P2S: A methodology to enable inter-organizational process design through web services. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 334–348. Springer, Heidelberg (2009)
5. Bieberstein, N., Laird, R.G., Jones, K., Mitra, T.: Executing SOA: A practical guide for the service-oriented architecture. Pearson Education, Boston (2008)
6. Boaro, L., Glorio, E., Pagliarecci, F., Spalazzi, L.: A business process design framework for b2b collaboration. In: The 2011 International Conference on Collaboration Technologies and Systems (2011)
7. BPMI. Business Process Modeling Notation (BPMN) Version 1.0., <http://www.bpmi.org/downloads/BPMN-V1.0.pdf>
8. Chun, S.A., Atluri, V., Nabil, Adam, R.: Using semantics for policy-based web service composition. *Distributed and Parallel Databases* 18 (2005)
9. Comerio, M., De Paoli, F., Maurino, A., Palmonari, M.: Nfp-aware semantic web services selection. In: IEEE International on Enterprise Distributed Object Computing Conference, p. 484 (2007)
10. Di Pietro, I., Pagliarecci, F., Spalazzi, L.: SWSAL: Semantic Web Service Annotation Language. no. 2008004453, SIAE Sezione Opere Inedite, Roma (October 15, 2008)
11. Di Pietro, I., Pagliarecci, F., Spalazzi, L.: Semantic Annotation for Web Service Processes in a Pervasive Computing. In: Hassanien, A.E., Abraham, A., Hagra, H., Abawajy, J.H. (eds.) *Pervasive Computing: Innovations in Intelligent Multimedia and Applications*. Springer, Berlin (2009)
12. Fellbaum, C.: *Wordnet: An Electronic Lexical Database*. MIT Press (1998)
13. Food and Agriculture Organization of United Nations. Geopolitical Ontology version 1.1 (November 2010), http://www.fao.org/countryprofiles/geopol_v11/index.html
14. Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., Holley, K., Arsanjani, A.: SOMA: A method for developing service-oriented solutions. *IBM Systems Journal* 47, 377–396 (2008)
15. Hepp, M., Leymann, F., Domingue, J., Wahler, A., Fensel, D.: Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Management. In: Proc. of the IEEE Int. Conference on e-Business Engineering (ICEBE 2005), Beijing, China, pp. 535–540 (2005)
16. International Organization for Standardization. Information and documentation — A reference ontology for the interchange of cultural heritage information. ISO 21127:2006 (September 2006), http://www.iso.org/iso/catalogue/catalogue_tc/catalogue_detail.htm?csnumber=34424
17. International Organization for Standardization. Information Technology — Business Operational View — Part 4: Business Transaction Scenarios — Accounting and Economic Ontology. ISO/IEC 15944-4:2000(E) (November 2007), http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=34424
18. Kaabi, R.S., Souveyet, C., Rolland, C.: Eliciting service composition in a goal driven manner. In: Proc. of the 2nd Int. Conference on Service Oriented Computing (ICSOC 2004), New York, NY, USA, pp. 308–315 (2004)
19. Köpf, H., et al.: *InteGrail ? Publishable Final Activity Report*. Technical Report IGR-P-DAP-156-07, InteGrail ? Intelligent Integration of Railway Systems - Project no. FP6 - 012526 (April 2010), http://www.integrail.info/documenti/InteGrail_Final_Project_Report.pdf

20. Kritikos, K., Plexousakis, D.: Semantic qos metric matching. In: ECOWS 2006, pp. 265–274 (2006)
21. Ma, Q., Zhou, N., Zhu, Y., Wang, H.: Evaluating Service Identification with Design Metrics on Business Process Decomposition. In: Proc. of the 2009 IEEE Int. Conference on Service Computing (SCC 2009), Bangalore, India, pp. 160–167 (2009)
22. Michael Maximilien, E., Singh, M.P.: Toward autonomic web services trust and selection, pp. 212–221. ACM Press (2004)
23. Mulye, R., Miller, J., Verma, K., Gomadam, K., Sheth, A.: A Semantic Template Based Designer for Web Processes. In: Proc. of the Third International Conference on Web Services, pp. 461–469 (2005)
24. OpenTravel Alliance, Boxborough, MA, USA. OpenTravel Implementation Guide — Version 1.5 (2010)
25. Papazoglou, M.P., Willem-Jan, H.: Service oriented architectures: approaches, technologies and research issues. VLDB Journal 16(3), 389–415 (2007)
26. Papazoglou, M.P., Yang, J.: Technologies for E-Services, pp. 175–233. Springer, Heidelberg (2002)
27. Di Pietro, I., Pagliarecci, F., Spalazzi, L.: Model checking semantically annotated services. IEEE Transactions on Software Engineering (2011)
28. Sheng, Q.Z., Benatallah, B., Maamar, Z., Dumas, M., Ngu, A.H.H.: Enabling personalized composition and adaptive provisioning of web services. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 322–337. Springer, Heidelberg (2004)
29. Shirazi, H.M., Fareghzadeh, N., Seyyedi, A.: A Combinational Approach to Service Identification in SOA. Journal of Applied Sciences 5(10), 1390–1397 (2009)
30. Toch, E., Gal, A., Dori, D.: Automatically Grounding Semantically-Enriched Conceptual Models to Concrete Web Services. In: Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 304–319. Springer, Heidelberg (2005)
31. Vanderfeesten, I., Reijers, H.A., van der Aalst, W.M.P.: Evaluating workflow process designs using cohesion and coupling metrics. Computer in Industry 59(5), 420–437 (2008)
32. de Bruijn, J., Lausen, H. (eds.) W3C Member Submission. Web Service Modeling Language (WSML) (June 2005), <http://www.w3.org/Submission/WSML/>
33. Yu, Q., Bouguettaya, A.: Framework for web service query algebra and optimization. ACM Trans. Web 2, 6:1–6:35 (2008)
34. Yu, T., Lin, K.-J.: Service selection algorithms for composing complex services with multiple qoS constraints. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 130–143. Springer, Heidelberg (2005)

Towards a System Model for Ensembles

Matthias Hölzl and Martin Wirsing

Institut für Informatik, Ludwig-Maximilians-Universität München
{matthias.hoelzl,martin.wirsing}@ifi.lmu.de

Dedicated to Carolyn Talcott

Abstract. Ensembles—software-intensive systems with massive numbers of nodes or complex interactions between nodes, operating in open and non-deterministic environments and dynamically adapting to changes in their environment or requirements—pose many challenges to software development. We present first steps towards a system model for ensembles that allows us to express requirements using a wide variety of logics and fitness criteria over arbitrary preorders. Using this system model we then give a precise definition of “black-box” adaptation and show how this naturally leads to a preorder of adaptability on ensembles.

1 Introduction

The increasing miniaturization and decreasing cost of computers and microcontrollers has enabled the nearly ubiquitous adoption of software-driven devices that interact with their physical environment, ranging from smartphones to industrial controllers and smart robots. We want these devices to work without the need for user configuration and to exploit local resources as well as “the cloud,” and we expect them to do so without compromising our privacy or security. None of the terms that we currently use—such as software-intensive systems or cyber-physical systems—encompasses the whole range of these new systems. The InterLink project [1] has therefore coined the term “ensemble” to describe them [2]:

Ensembles are software-intensive systems with massive numbers of nodes or complex interactions between nodes, operating in open and non-deterministic environments in which they have to interact with humans or other software-intensive systems in elaborate ways. Ensembles have to dynamically adapt to new requirements, technologies or environmental conditions without redeployment and without interruption of the system’s functionality, thereby blurring the distinction between design-time and run-time.

One of the most important and challenging duties when engineering ensembles is to ensure that an ensemble can continue to work reliably in spite of unforeseen changes in its environment and requirements, and that adaptation does not lead to the system becoming inoperable, unsafe or insecure. To achieve this goal, the ASCENS project researches ways of building ensembles that combine traditional software engineering approaches and techniques from autonomic, adaptive,

knowledge-based and self-aware systems with the assurance about functional and non-functional properties provided by formal methods. The different disciplines involved in the project use different formalisms and techniques that have to be related in a single framework in order to present ensemble engineers with a unified development approach. In this paper we present a first step toward a common system model for ensembles that can integrate different models, logics and objectives, and therefore serve as foundation of our approach to ensemble engineering.

While the meaning of “adaptivity” may seem intuitively obvious, various communities use different and incompatible definitions of the term, and surprisingly few generally applicable, precise definitions are available in the literature. Based on our system model for ensembles we define a notion of “black-box” adaptation that gives rise to a preorder of adaptivity on ensembles based on their ability to satisfy goals or maximize a performance measure in different environments.

In the next section we present related work, then we introduce a mathematical model of ensembles and their composition. Building on this model we propose a definition of one view of adaptivity, the “black-box” view of adaptivity to a range of environments or goals. The final section concludes.

Personal Note: We have both known Carolyn for a long time, and we have both had the opportunity to stay with her research group at SRI for several weeks or months, and to work with her in the InterLink project. This has led to many interesting research results, e.g., the application of soft constraints to software-defined radios [3], the theory of monoidal soft constraints [4], and the prototype of the ConstraintMuse system [5]. Cooperating with Carolyn has always been an enjoyable experience, not only because of the extensive knowledge and excellent ideas that she contributed, but also because of her unpretentious and kind personality. We are looking forward to further stimulating scientific exchanges with Carolyn.

2 Related Work

Various approaches to develop generally applicable formal theories of systems have been presented in the literature. Among the most elegant and epistemologically satisfying ones is Goguen’s categorical approach to general systems [6] which represents objects as functors satisfying the so-called sheaf condition, systems as diagrams and behaviors of systems as limits. However, this definition of behavior is rather abstract and difficult to understand for software engineers.

We have chosen a more concrete approach which describes systems directly by their input/output behavior. Our approach is related to the set-theoretic formalization of General Systems Theory introduced by Mesarovic [7,8], therefore some of the results derived in [7] are also valid for our definition of ensembles. However, in order to accommodate, e.g., distributed systems where a linear notion of time may not be an appropriate model, we define time systems for preordered time structures, whereas Mesarovic requires a total order. Therefore results for timed and dynamic systems do not immediately apply to ensembles.

Another system model based on Mesarovic's theory and notation is *gMobS* [9] which models object-based systems as hierarchical compositions of input/output relations. Since *gMobS* is mainly used to investigate properties of I/O descriptions it models only time systems whose components are connected by message passing, does not define notions of goal satisfaction or fitness, and has no general concept of combination operators.

A system model for component-oriented software and systems engineering based on the notion of streams has been proposed by Broy [10,11]. This approach is similar to our time ensembles in providing a denotational view of systems; however whereas our aim is to provide a generally applicable model for ensembles and adaptation, Broy's goal is to formalize the typical steps of a development process. A similar difference can also be seen in Broy's approach to adaptation [12], which only considers system that can be represented by input/output streams, and mostly deals with the handling of non-determinism in such systems.

A large number of component frameworks and interconnection mechanisms have been proposed, e.g., [13] or, based on Goguen's categorical approach to systems mentioned above, [14], but they either do not deal with adaptation at all, or only in the form of reconfiguration.

Gul Agha's actor model is an important formalism for specifying concurrent distributed systems [15,16]. It represents systems as collections of actors, which can exhibit behavior, communicate with other actors and spawn new actors. Communication is asynchronous and reliable. Non-deterministic term rewriting as exemplified by Meseguer's Maude language [17] is an algebraic approach to describing dynamically evolving concurrent systems. The Maude system, developed in the research groups of Meseguer at UIUC and Talcott at SRI, has proven to be well-suited for specifying, prototyping and analyzing real-time and probabilistic scenarios [18,19]. The actor model and Maude are leading candidates for operationally realizing and analyzing models which are denotationally specified using our approach.

There is a large amount of research about adaptation in various areas, e.g., in control theory, dynamical systems, machine learning and artificial intelligence, and many notions of adaptivity have been defined in the literature, see [20] for an overview. Many of these results have influenced the present work; we are, however, not aware of other approaches that try to precisely specify an observational notion of adaptation for general systems.

We have based the term "heterostatic system" on Klopff's heterostatic theory of adaptive systems [21], but our system is quite different from Klopff's and the impact of his theory on ours is more indirect and mostly through later developments in reinforcement learning.

3 Ensembles

The description of ensembles given in the introduction is not precise enough to serve as a foundation of a rigorous system development process or to allow

a meaningful definition of adaptation. To be useful, a formal definition has to be general enough to capture the various kinds of ensembles that appear when building practical systems. No notion of ensemble based on a concrete behavioral model, such as state machines, term rewriting or differential equations, is sufficiently general to serve this purpose; in practice many ensembles consist of subsystems that are specified using different mathematical methods. As an example, it is common to describe physical parts of an ensemble using differential equations but to model controllers using state machines; any meaningful mathematical theory of ensembles has to be able to accommodate the integration of such widely different modeling and analysis techniques. Therefore, we describe ensembles and their constituent parts abstractly as relations on sets which we often regard as relations between inputs and outputs.

In the following we write $\mathcal{F}[A \rightarrow B]$ for the set of all functions from A to B . If $\{U_i \mid i \in I\}$ is a set of sets we write $U = \prod\{U_i \mid i \in I\} = \prod_{i \in I} U_i$ for the product of the U_i , i.e., the set of all functions $x \in \mathcal{F}[I \rightarrow \bigcup_{i \in I} U_i]$ such that $x(i) \in U_i$; if there is no risk of confusion we sometimes abbreviate this to $\prod U_i$. We often write elements $x \in U$ in the form $(x_i)_{i \in I}$, and we denote the k -th projection by pr_k , i.e., $\text{pr}_k x = x(k)$. Therefore, $\text{pr}_k(x_i)_{i \in I} = x_k$. We generalize the projection operation to sets of indices, so that for $J \subseteq I$ we have $\text{pr}_J(x_i)_{i \in I} = (x_j)_{j \in J}$. If $I = \{1, \dots, n\}$ we write $U = U_1 \times \dots \times U_n$ and $(x_i)_{i \in \{1, \dots, n\}} = (x_i)_{1 \leq i \leq n} = (x_1, \dots, x_n)$; we also use this notation for finite index sets which are not subsets of the integers. If $f : A \rightarrow B$ and $A' \subseteq A$ we write $f|_{A'} : A' \rightarrow B$ for the restriction of f to values in A' .

Let I be a set and $(V_i)_{i \in I}$ be a family of sets indexed by I . We write $\mathcal{R}((V_i)_{i \in I})$, or $\mathcal{R}(V_i)$ if I is clear from the context, for the set of all relations over $(V_i)_{i \in I}$, i.e., the powerset of the product, $\mathfrak{P}(\prod_{i \in I} V_i)$. For any $R \in \mathcal{R}((V_i)_{i \in I})$ we call $(V_i)_{i \in I}$ its *type* or *schema*; if $J \subseteq I$ we call $(V_j)_{j \in J}$ a *subtype* of $(V_i)_{i \in I}$ and $(V_i)_{i \in I}$ a *supertype* of $(V_j)_{j \in J}$; we say that $R \in \mathcal{R}((V_i)_{i \in I})$ is an *extended relation over* $(V_j)_{j \in J}$. We extend the projection operator to relations and write $\text{pr}_J R$ for the projection $\{(v_j)_{j \in J} \mid (v_i)_{i \in I}\} = \{\text{pr}_J r \mid r \in R\}$. We often write $R(r)$ instead of $r \in R$.

3.1 General Ensembles

Formally we do not distinguish ensembles, other systems and components, but in general we only call a system an ensemble if it is the combination of many interacting parts (using combination operators as described below), and we generally do not call systems components if they are combinations of subsystems.

Definition 1. *Let I be a (finite or infinite) set, and let $\mathcal{V} = (V_i)_{i \in I}$ be a family of sets. An ensemble or (general) system or component of type \mathcal{V} is a relation S of type \mathcal{V} . □*

The use of infinite index sets allows us to model ensembles with dynamically changing size, since we can have an arbitrary number of “inactive” components (e.g., denoted by the special value \top) which can be activated during the computation.

In the most general case, there is no unique notion of “input” or “output” to a system. For example, if $S(w, p)$ is the relation between a robot’s wheel movements and its position (in a given environment) we may regard w as input and p as output when simulating the robot’s movement, and we may regard p as input and w as output when computing the control for a desired path. A single component may even serve as input and output simultaneously: if we consider a clutch for clasping objects as a system in its own right, then the position of the clutch can simultaneously serve as input (as it is determined by the robot to which the clutch is attached) and as output (as it determines the position of a clasped object, provided that this object is light enough to be moved by the robot).

While they are not system-inherent properties it is in many cases helpful to regard some of the sets $V_i \in \mathcal{V}$ as inputs and others as outputs. We can do so by defining an isomorphism which, roughly speaking, divides \mathcal{V} into inputs X and outputs Y , and identifies S with a relation of type (X, Y) . More precisely: Let $I_X, I_Y \subseteq I$ with $I_X \cup I_Y = I$, let $I_{XY} = I_X \cap I_Y$, $X = (V_j)_{j \in I_X}$ and $Y = (V_k)_{k \in I_Y}$. We define a relation $S_{i/o}$ of type (X, Y) as follows:

$$S_{i/o} = \{((v_j)_{j \in I_X}, (v_k)_{k \in I_Y}) \mid (v_i)_{i \in I} \in S\}$$

To illustrate this concept we consider the clutch described above. Ignoring the passage of time and rotational degrees of freedom, we assume that the clutch can be at any location in space, and either open or closed: $S_{\text{clutch}} \in \mathcal{R}(V_{\text{pos}}, V_{\text{closed}}) = \mathcal{R}(\mathbb{R}^3, \mathbb{R}^3 \times \{0, 1\})$. If we take $I_X = \{\text{pos}\}$, $I_Y = \{\text{pos, closed}\}$, we obtain

$$\begin{aligned} S_{i/o} &\in \mathcal{R}(V_{\text{pos}}, V_{\text{pos}} \times V_{\text{closed}}) \\ S_{i/o} &= \{(\mathbf{x}, (\mathbf{x}, b)) \mid \mathbf{x} \in \mathbb{R}^3, b \in \{0, 1\}\} \end{aligned}$$

i.e., the input and output position are equal for each element of $S_{i/o}$. It is easy to see that it is necessary to use the same value for all occurrences of elements of I_{XY} : if we defined $S_{i/o}$ without equating the shared inputs and outputs we would obtain

$$S_{i/o}^{\text{wrong}} = \{(\mathbf{x}, (\mathbf{y}, b)) \mid \mathbf{x}, \mathbf{y} \in \mathbb{R}^3, b \in \{0, 1\}\}$$

where the input and output positions could take their values independently, which clearly contradicts the physical reality.

Since $S_{i/o}$ and S are obviously isomorphic, we often identify them and say that S has or can be given input/output configuration (X, Y) or simply that S has configuration (X, Y) . We call X the *input set* or *inputs* of the configuration and Y the *output set* or *outputs* of the configuration. A configuration for which $I_X = \emptyset$ so that $X = \{\emptyset\}$ is called *input-free*, a configuration with $I_Y = \emptyset$ and thus $Y = \{\emptyset\}$ is called *output-free*. If the configuration is clear from the context we sometimes call X and Y the inputs and outputs of S , and we say S itself is input-free or output-free. If S is a function from X to Y we call it a *functional system* or *functional ensemble* (for that input/output configuration).

It is generally neither feasible nor desirable to specify the behavior of an ensemble as a single relation that captures all relationships between inputs and

outputs. When building or analyzing ensembles the interesting questions are often how the observed behavior of an ensemble arises from the behaviors of its components, or how a desired behavior can be synthesized by combining simpler components. To achieve this we will later introduce operators that combine a family of systems $\mathcal{S} = (S_l)_{l \in L}$ where each S_l has configuration (X_l, Y_l) into a larger system by connecting some output ports of ensembles $(S_l)_{l \in L'}$ where $L' \subseteq L$ to appropriate output ports of (the same or other) ensembles in \mathcal{S} . Since these connections are usually only concerned with a subset of the available inputs and outputs of each S_l it is again convenient to distinguish the “ports” that should be connected by an operator from those that should stay unconnected. Using the technique described in the previous paragraphs we divide each X_l into the *input ports* X_l^{in} and the *non-connectible inputs* X_l^{nc} , and similarly Y_l in the *output ports* Y_l^{out} and the *non-connectible outputs* Y_l^{nc} . We again identify each $S_l \in \mathcal{R}(X_l, Y_l)$ with $S'_l \in \mathcal{R}(X_l^{\text{nc}}, X_l^{\text{in}}, Y_l^{\text{nc}}, Y_l^{\text{out}})$.

3.2 Modal and Time Ensembles

In most cases we are interested in modeling ensembles which change their behavior over time; this can be achieved by considering modal systems and time systems. Recall that a *preorder* \preceq on a set U is a binary relation that is reflexive and transitive, i.e., for all $x, y, z \in U$ we have $x \preceq x$ and $x \preceq y \wedge y \preceq z \implies x \preceq z$. If \preceq is antisymmetric, i.e., if $x \preceq y \wedge y \preceq x \implies x = y$ it is called a (*partial*) *order*; if for all $x, y \in U$ we have $x \preceq y \vee y \preceq x$ then \preceq is a *total order*. We say that a partial order on an ordered monoid (T, \times) is *Archimedean* if for all $x, y \in T$ there exist an integer n such that $n \times x \succeq y$; elements for which this property does not hold are called *infinitesimal relative to each other*.

Definition 2. Let T , the possible worlds or time structure, be a set, let R be a binary relation on T , $(A_i)_{i \in I}$ a family of sets, and let $V_i = \mathcal{F}[T \rightarrow A_i]$. A modal system or modal ensemble over $(A_i)_{i \in I}$ with possible worlds T is a general system S over $(V_i)_{i \in I}$. If R is a preorder we call T a time system or time ensemble over $(A_i)_{i \in I}$ with time structure T . □

If the sets A_i can be interpreted as sets of atoms in a (modal) logic, modal systems give rise to Kripke structures for that logic as follows [22,23]: The pair (T, R) defines a Kripke frame with worlds T and relation R ; a canonical labeling function can be defined by mapping each $t \in T$ to the set of all elements $(a_i)_{i \in I}$ for which an element $(v_i)_{i \in I} \in S$ exists with $v_i(t) = a_i$ for all $i \in I$.

Similarly, if T is preordered it can be seen as a very weak model of time and the system can be interpreted as Kripke model for a temporal logic. Since some models of distributed systems may not provide any stronger guarantees about their temporal behavior we require neither antisymmetry nor a semigroup or monoid structure in the definition and state these requirements explicitly when they are necessary. Note that even if the time structure is an ordered monoid but not Archimedean the system may exhibit unintuitive characteristics, for example *Zeno behaviors* [24,25]: if t_1 is infinitesimal relative to t_2 , then time t_2 cannot be reached by performing any finite number of steps of duration t_1 .

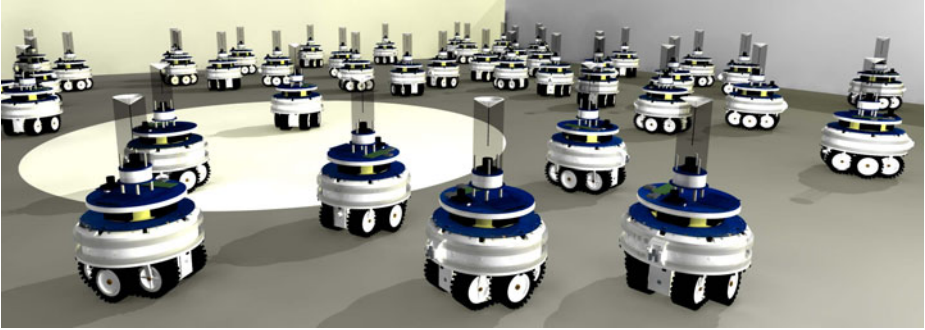


Fig. 1. Simulation of the ensemble of solar-powered robots described in Example 1 using MarXbots [26] running in the Argos simulator [27]. Since neither the real nor the simulated robot have solar cells the light distribution is simulated by the floor color which can be measured using the MarXbot’s ground sensor. In this image, the white circle on the left hand side represents a circular light source.

Example 1 (Robot swarm). Suppose that we have a swarm of n solar-powered robots where each robot has to recharge its battery by staying in bright light for a certain time, see Fig. 1. We make the following assumptions: (1) the robots rest on a flat surface and stay inside a fixed arena with coordinates in the unit square $\mathcal{I}^2 = \{(x, y) \mid 0 \leq x, y \leq 1\}$, (2) the battery charge of each robot at each moment in time is represented as a number in the unit interval $\mathcal{I} = [0, 1]$, (3) the light intensity in the arena at each moment in time can be described by a function $\mathcal{I}^2 \rightarrow \mathcal{I}$, (4) the physical characteristics of the solar cell and battery are known and can be described by functions f_{charge} and f_{dis} (see below), (5) the robots are cylindrical with a fixed radius, and (6) all robots follow the same fixed (but possibly non-deterministic) algorithm prog_d . One possible way to model the robot swarm is as time system over the time structure :

$$S_1^{\text{rsw}} \in \mathcal{R}(X_{\text{light}}, Y_{\text{pos}}, Y_{\text{bat}})$$

where the sets are defined as

$$\begin{aligned} A_{\text{light}} &= \mathcal{F}[\mathcal{I}^2 \rightarrow \mathcal{I}] & A_{\text{pos}} &= (\mathcal{I}^2)^n & A_{\text{bat}} &= \mathcal{I}^n \\ X_{\text{light}} &= \mathcal{F}[\text{ }^+ \rightarrow A_{\text{light}}] & Y_{\text{pos}} &= \mathcal{F}[\text{ }^+ \rightarrow A_{\text{pos}}] & Y_{\text{bat}} &= \mathcal{F}[\text{ }^+ \rightarrow A_{\text{bat}}] \end{aligned}$$

and where the relation $S_1^{\text{rsw}}(x_{\text{light}}, y_{\text{pos}}, y_{\text{bat}})$ holds iff

1. y_{pos} is a function $\text{ }^+ \rightarrow A_{\text{pos}}$ such that $y_{\text{pos}}(0)$ is a tuple of valid initial positions for the robots (i.e., a vector of positions such that no two robots overlap) and $y_{\text{pos}}(t)$ is consistent with all robots starting in $y_{\text{pos}}(0)$ and moving according to prog_d , and
2. y_{bat} is the function

$$t \mapsto \left(\int_{t'=0}^t f_{\text{charge}}(x_{\text{light}}(t')(y_{\text{pos}}^i(t'))) - f_{\text{dis}}(y_{\text{pos}}^i(t')) dt' + e_i^0 \right)_{1 \leq i \leq n}$$

where f_{charge} is the function that describes how light falling on the robot’s solar cell charges the battery, f_{dis} is the function describing the discharge of the battery due to the robot’s movement and possibly self-discharge, e_i^0 is the initial charge status of the battery for robot i , and $y_{\text{pos}}^i = \text{pr}_i \circ y_{\text{pos}}$. Note that the i -th component of y_{bat} is a function of x_{light} and y_{pos}^i , representing the fact that the energy level of each robot depends both on the distribution of light in the arena and on the movement of the robot. The value $x_{\text{light}}(t')$ is the light distribution function for time t' , hence $x_{\text{light}}(t')(y_{\text{pos}}^i(t'))$ is the intensity of the light shining on the i -th robot at time t' .

S_1^{rsw} is not a functional ensemble even if prog_d and the behaviors of colliding robots are deterministic, since every light distribution is related to the movements of the robots in all valid starting positions. To obtain a functional system for ensembles with deterministic movements we could add the initial positions of the robots as additional input x_{pos0} and restrict the resulting relation S_1^{rsw} to tuples $(x_{\text{light}}, x_{\text{pos0}}, y_{\text{pos}}, y_{\text{bat}})$ where y_{pos} satisfies $y_{\text{pos}}(0) = x_{\text{pos0}}$.

If the robots can independently be switched between different programs prog_k taken from a set of programs A_{prog} , then we can model the ensemble by a system S_2^{rsw} as follows:

$$\begin{aligned} A_{\text{prog}} &= \{\text{prog}_k \mid k \in K_{\text{prog}}\}^n \\ X_{\text{prog}} &= \mathcal{F}[+ \rightarrow A_{\text{prog}}] \\ S_2^{\text{rsw}} &\in \mathcal{R}(X_{\text{prog}}, X_{\text{light}}, Y_{\text{pos}}, Y_{\text{bat}}) \end{aligned}$$

with the definition of $S_2^{\text{rsw}}(x_{\text{prog}}, x_{\text{light}}, y_{\text{pos}}, y_{\text{bat}})$ similar to the one for S_1^{rsw} given above, but instead of requiring y_{pos} to model the robot’s movement according to the algorithm prog_d we now take into account the changes in movement strategies described by the input x_{prog} . □

It is easy to see that this notion of system generalizes many of the definitions of “system” or “dynamical system” found in the literature; e.g., by using the positive real numbers or the integers for T and choosing complete metric spaces for X and Y which satisfy the axioms given by Chueshov, we obtain the definition of dynamical systems given in [28]. Similarly, we can express many properties commonly found in component systems for software in the formalism defined in this section.

3.3 Goal Satisfaction and Fitness

When engineering ensembles we are often interested in systems which perform a certain function as well as possible while satisfying given requirements and remaining in a consistent state. Therefore we proceed with definitions for inconsistency, goal satisfaction and fitness. Since different formal methods require different logics and models, we want to be able to specify different properties of the same ensemble using different logical systems. The following definition of goal satisfaction is parametric in the logic used for the specification.

We identify inconsistency with systems that do not specify *any* input-output relation, i.e., a system $S \in \mathcal{R}(\mathcal{V})$ is inconsistent if S is empty. In that case we write $S \models \perp$, otherwise $S \not\models \perp$.

In the following we define a model-theoretic notion of a system satisfying a goal γ , where γ is expressed as a predicate in a suitable logic \mathcal{L} with signature Σ . We write \mathcal{F} for the set of all formulae of Σ and \mathcal{M} for the class of all structures of Σ . Furthermore, we assume that a set Aux of auxiliaries and a satisfaction relation \models is defined, such that $\mathfrak{M}, \alpha \models \gamma$ iff \mathfrak{M} is a model of γ given $\alpha \in \text{Aux}$. The auxiliaries can be used to expose parts of the internal formation of \mathfrak{M} when the relationship between \mathfrak{M} and the system S is established. In the case of first-order logic, the set Aux is usually the set of all substitutions for variables in Σ , and $\mathfrak{M}, \alpha \models \gamma$ is the relation that γ is true in \mathfrak{M} given substitution α . Similarly, in temporal logics, Aux will often be defined as the set of all states and $\mathfrak{M}, \alpha \models \gamma$ then means that γ holds for \mathfrak{M} when the system is in state α .

Since our goal is for formulae of \mathcal{L} to express properties of S , it is necessary to relate (the interpretations of) some formulae in \mathcal{F} with the system S . To achieve this, we assume that a nonempty subset $\mathcal{F}_S \subseteq \mathcal{F}$ is given that contains formulae whose interpretation in a structure \mathfrak{M} characterizes the relevant properties of S , i.e., \mathcal{F}_S should contain a subset of \mathcal{F} so that fixing the meaning of all formulae in \mathcal{F}_S in a structure ensures that this structure also satisfies all properties of S that can be expressed in Σ .

To fix the interpretation of formulae in \mathcal{F}_S in structures of Σ , we assume that for each structure \mathfrak{M} of Σ we have a map $\mathcal{T}_{\mathfrak{M}} : \mathcal{F}_S \times \text{Aux} \rightarrow \{true, false\}$. If $S \models \perp$ we require $\mathcal{T}_{\mathfrak{M}}(P, \alpha) = false$ for all P, α . If for all sentences P in \mathcal{F}_S and all auxiliaries α the relation

$$\mathfrak{M}, \alpha \models P \iff \mathcal{T}_{\mathfrak{M}}(P, \alpha) \quad (1)$$

holds, we say \mathfrak{M} *characterizes* S (for $\mathcal{T}_{\mathfrak{M}}$) and write $\text{char}_{\mathcal{T}_{\mathfrak{M}}}(\mathfrak{M}, S)$ or, if $\mathcal{T}_{\mathfrak{M}}$ is clear from the context, $\text{char}(\mathfrak{M}, S)$. Let γ be a formula in \mathcal{F} . If for all structures \mathfrak{M} and all auxiliaries α we have

$$\text{char}_{\mathcal{T}_{\mathfrak{M}}}(\mathfrak{M}, S) \implies \mathfrak{M}, \alpha \models \gamma \quad (2)$$

we say that S satisfies goal γ and write $S \models_{\mathcal{T}_{\mathfrak{M}}} \gamma$ or $S \models \gamma$. Note that if $S \models \perp$ and $\mathcal{F}_S \neq \emptyset$ then according to our restriction on $\mathcal{T}_{\mathfrak{M}}$ we have $\mathfrak{M} \not\models S$ and hence $S \models \gamma$ for all goals γ .

Example 2 (First-order model of a robot swarm). Let \mathcal{L} be multi-sorted first-order predicate logic with equality and let Σ be a signature that contains sorts τ_{prog} , τ_{light} , τ_{pos} and τ_{bat} , a relation symbol S of sort $\tau_{\text{prog}} \times \tau_{\text{light}} \times \tau_{\text{pos}} \times \tau_{\text{bat}}$ and free variables x_p, x_l, y_p, y_b . Let $\mathcal{F}_S = \{S(x_p, x_l, y_p, y_b)\}$, let \mathfrak{M} be a structure of Σ that interprets τ_{prog} as V_{prog} , τ_{light} as V_{light} etc., let Aux be all substitutions α of x_p, x_l, y_p, y_b in their respective domains, and let

$$\mathcal{T}_{\mathfrak{M}}(S(x_p, x_l, y_p, y_b), \alpha) = S_2^{\text{rsw}}(\alpha(x_p), \alpha(x_l), \alpha(y_p), \alpha(y_b))$$

It is clear that \mathfrak{M} characterizes S_2^{rsw} iff $S^{\mathfrak{M}}$, the interpretation of S in \mathfrak{M} , is equal to S_2^{rsw} .

If we want to specify the property that no robot runs out of energy before t_0 we can define γ_1 as:

$$\forall x_p, x_1, y_p, y_b : S(x_p, x_1, y_p, y_b) \implies \forall i \in [1, n] : \forall t \in [0, t_0] : y_b(i)(t) > 0$$

Let \mathfrak{M} be a structure of Σ that characterizes S_2^{rsw} , so that $S^{\mathfrak{M}} = S_2^{\text{rsw}}$. Then $S^{\mathfrak{M}}$ contains configurations which start with battery levels of 0 and so we have $\mathfrak{M} \not\models \gamma_1$. Hence γ_1 is not satisfied by S_2^{rsw} . If we make the assumption that prog_0 is the program that does not move the robot at all and that the energy consumption f_{dis} is 0 in that case, and if we restrict the predicate γ_2 to inputs x_p mapping all times before t_0 to prog_0 and to initial positions with positive battery charge, we obtain γ_2 as

$$\begin{aligned} \forall x_p, x_1, y_p, y_b : S(x_p, x_1, y_p, y_b) \implies \\ \forall i \in [1, n] : \forall t \in [0, t_0] : y_b(i)(0) > 0 \wedge (\forall t' \in [0, t] : x_p(t') = \text{prog}_0) \implies \\ y_b(i)(t) > 0 \end{aligned}$$

It is easy to see that in this case every model satisfying equation (1), i.e.,

$$\mathfrak{M}, \alpha \models S(x_p, x_1, y_p, y_b) \iff S_2^{\text{rsw}}(\alpha(x_p), \alpha(x_1), \alpha(y_p), \alpha(y_b))$$

also satisfies $\mathfrak{M}, \alpha \models \gamma_2$ for all α , hence we have $S_2^{\text{rsw}} \models \gamma_2$. □

Sometimes we not only want to satisfy a given goal, we want to pick the *best* solution that satisfies the given goal according to some *objective function* or *fitness function*. To this end we define a *heterostatic system* as a quadruple of a system S in input/output configuration (X, Y) , a goal γ , a partially ordered set (G, \leq) and a fitness function $\phi : X \times Y \rightarrow G$. Heterostatic systems can be compared according to their “fitness for a goal” using different notions of ordering. In the following we define the *weak heterostatic order* \preceq which corresponds to the “relational ordering” of domain theory; another suitable ordering would be the stronger Egli-Milner ordering [29,30].

Two heterostatic systems $H_1 = (S_1, \gamma_1, G, \phi_1)$ and $H_2 = (S_2, \gamma_2, G, \phi_2)$ with the same configuration (X, Y) and the same range of the fitness functions G are called *assimilable*. We write $H_1 \preceq H_2$ if for all $(x, y) \in S_1$ that satisfy γ_1 there is $(x, y') \in S_2$ satisfying γ_2 with $\phi_1(x, y) \leq \phi_2(x, y')$. We write $H_1 \prec H_2$ if $H_1 \preceq H_2$ and $H_2 \not\preceq H_1$, i.e., if for at least one $(x, y) \in S_2$ satisfying γ_2 there is no $y' \in Y$ such that $(x, y') \in S_1$, (x, y') satisfies γ_1 and $\phi_2(x, y) \leq \phi_1(x, y')$. Intuitively, $H_1 \prec H_2$ if they work on the same inputs and H_2 performs at least as well as H_1 on all inputs but strictly better than H_1 on at least one input, or if H_2 works on a larger subset of the possible inputs, and performs at least as well as H_1 on the shared inputs. We usually only compare heterostatic systems if $\phi_1 = \phi_2$.

Note that in the definition of fitness we distinguish between inputs and outputs, since the ability to perform on a wider range of inputs seems to justify a higher fitness rating, whereas the ability to generate more behaviors for the same input without being able to improve the fitness value for that input does

not seem to be a worthwhile property. If unpredictability is a desired feature (as it would be, e.g., for a poker-playing system), then it would be more appropriate to model the outputs, e.g., as probability distributions and to rate the fitness according to the expected value.

Our definition of fitness is an “optimistic” one, since we only judge fitness according to the best possible behavior, without taking into account whether the ensemble will actually exhibit that behavior, as can be seen in the following example:

Example 3 (Fitness function). Let $X = \{x_1, x_2\}$, $Y = \{y_1, y_2, y_3\}$, $S_1 = \{(x_1, y_2)\}$, $S_2 = \{x_1\} \times Y$, $S_3 = \{(x_1, y_2), (x_2, y_2)\}$, $\gamma = \text{true}$, and $\phi : X \times Y \rightarrow \mathbb{R}$ with

$$\phi(x_1, y_1) < \phi(x_1, y_2) < \phi(x_1, y_3)$$

Let $H_i = (S_i, \gamma, \phi)$ for $1 \leq i \leq 3$. Then $H_1 \prec H_2$ and $H_1 \prec H_3$. H_2 and H_3 are assimilable, but not comparable: $H_2 \not\prec H_3$, since there is no $(x_1, y') \in H_3$ such that $\phi(x_1, y_3) \leq \phi(x_1, y')$, and $H_3 \not\prec H_2$ since the domain of H_2 does not contain x_2 , i.e., there is no y' such that $(x_2, y') \in H_2$. Therefore H_3 is defined on a larger domain but its maximum performance on the values it shares with H_2 is worse than H_2 . Note that in practice H_2 may perform worse than either H_1 or H_3 , since H_2 may always perform (x_1, y_1) which has a lower objective value than the value (x_1, y_2) performed by H_1 and H_3 . \square

Example 4 (Fitness for robot ensembles). Let S_1 be S_2^{rsw} from Ex. 1 with the set $V_{\text{prog}} = \{\text{prog}_0, \text{prog}_1\}$ consisting of two algorithms with different behaviors, let $\gamma = \gamma_2$ from Ex. 2, and let

$$\phi = \sum_{i \in I} \left(\int_{t=0}^{t_0} y_{\text{bat}}^i dt \right)$$

Let S_2 be the restriction of S_1 to the single algorithm prog_0 , and let S_3 be the restriction of S_1 to those y_{pos} maximizing the fitness for some x_{prog} and x_{light} . Let $H_i = (S_i, \gamma_2, \phi)$. Then, since the algorithms are inputs, $H_1 \prec H_2$, but since the y_{pos} are outputs and H_3 can by definition achieve the same fitness as H_1 for any input, we have $H_1 \preceq H_3$ and $H_3 \preceq H_1$. \square

This last example shows that fitness is only a preorder on ensembles with the same behavioral domain, not a partial order.

3.4 Combination Operators

When developing software for ensembles it is in general not desirable to regard the ensemble as a single entity; usually the ensemble consists of a hierarchy of systems, and we frequently have to change focus from the overall ensemble to one of its components or sub-components and back. To formalize this nested structure of ensembles we define combination operators that build larger systems out of smaller components.

In the simplest case we have systems that don't interact with each other, or that are composed in sequence, so that the outputs of system S_0 , serve as input to system S_1 . In many cases, however, compositions of systems exhibit more complex behaviors. For example, some outputs of a system S may be connected to some of its inputs, resulting in a feedback loop. In addition, the connections between systems may themselves exhibit complex behaviors, e.g., when mobile robots are connected via a wireless network. However, even these complex combinations of systems can be represented by defining those parts of the connection which exhibit complex behaviors as systems in their own right and combining several of these systems via *combination operators* or *connectors*.

Intuitively, a combination operator \otimes builds a new system that preserves all the non-connectible inputs and outputs, but it may additionally add new inputs X_\otimes and outputs Y_\otimes that influence the properties of the connection itself; and \otimes may also make some or all of the connected inputs and outputs available for further interaction with the environment.

More precisely, let $\mathcal{S} = (S_l)_{l \in L}$ be a family of systems where each S_l can be given input/output configuration (X_l, Y_l) . A combination operator for systems in configuration $((X_l, Y_l))_{l \in L}$ is then a function \otimes mapping the S_l to a new system S such that

$$S \in \mathcal{R}\left(\left(\prod_{l \in L} X_l^{\text{nc}} \times X_\otimes\right), \left(\prod_{l \in L} Y_l^{\text{nc}} \times Y_\otimes\right)\right)$$

X_\otimes and Y_\otimes are the new inputs and outputs introduced by the operator and may contain some of the connected inputs and outputs. If the S_l are modal or time systems we assume that they are over the same possible worlds or time structure T and that the sets X_\otimes and Y_\otimes are also sets of functions over T .

Since unconnected inputs and outputs play no role for the connection operator we can usually ignore them in the definition and write the type of \otimes as

$$S \in \mathcal{R}(X_\otimes, Y_\otimes)$$

with the convention that \otimes is applicable to any system configuration that is a supertype of $((X_j^{\text{in}})_{j \in J}, (Y_k^{\text{out}})_{k \in K})$ and that \otimes retains the unconnected inputs and outputs.

Consider, for example the *cascade operator* \triangleright shown in Fig. 2 which connects the (connectable) outputs of system S_1 with the (connectable) inputs of S_2 (see [7], p. 171). This operator can be defined as follows:

$$\begin{aligned} S_1 &\in \mathcal{R}(X_1^{\text{nc}}, (Y_1^{\text{nc}} \times Y_1^{\text{out}})) \\ S_2 &\in \mathcal{R}((X_2^{\text{nc}} \times X_2^{\text{in}}), Y_2^{\text{nc}}) \\ Y_1^{\text{out}} &= X_2^{\text{in}} \\ ((x_1, x_2), (y_1, y_2)) &\in S_1 \triangleright S_2 \iff \\ &\exists z \in Y_1^{\text{out}} : (x_1, (y_1, z)) \in S_1 \wedge ((x_2, z), y_2) \in S_2 \end{aligned}$$

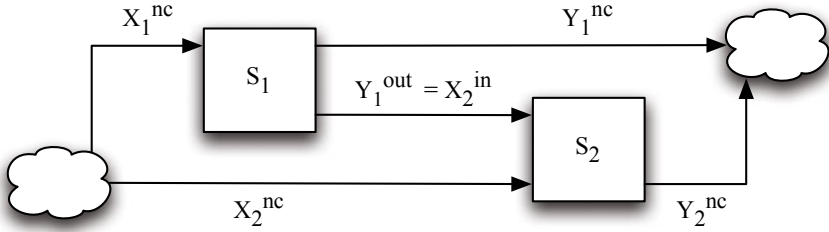


Fig. 2. Cascade operator \triangleright

Example 5 (Cascade operator). Let $\text{prog}_d \in A_{\text{prog}}$ be the algorithm used by the robots in S_1^{rsw} (see Ex. 1), and let $S^{\text{prog}} = Y_1^{\text{out}} = \{t \mapsto (\text{prog}_d)_{1 \leq i \leq n}\}$. Setting $S_2 = S_2^{\text{rsw}}$, $X_2^{\text{nc}} = X_{\text{light}}$, $X_2^{\text{in}} = X_{\text{prog}}$, $Y_2 = Y_{\text{pos}} \times Y_{\text{bat}}$ we obtain

$$S_1^{\text{rsw}} = S^{\text{prog}} \triangleright S_2^{\text{rsw}} \quad \square$$

The cascade operator defined above is only one way of modeling the sequential connection of inputs and outputs. If $S_2 | X_2^{\text{in}} \subset S_1 | Y_1^{\text{out}}$, then whether the cascade operation corresponds to a correct model of the connection between S_1 and S_2 depends on the actual systems modeled. If we connect, e.g., two mechanical systems with a rigid connector then the cascade may in many cases be a faithful representation of the joint behavior. If, on the other hand, S_1 is a computer program that generates values in S_1^{out} as output, it has no way of knowing which of its possible outputs can be accepted by S_2 . Therefore the joint behavior is not correctly modeled by a cascade which assumes that S_1 will necessarily only select outputs that S_2 can accept as inputs. This, however, is not a flaw of the model; rather it represents a property of the real system: we sometimes have to feed-back explicit information about connections or outputs of components to other parts of the system.

Let us, therefore, briefly look at a straightforward example of this happening: a simple feedback loop for a single component arises when the connectable outputs of S are connected to its connectable inputs, see Fig. 3. This operator θ can be described by

$$\begin{aligned}
 & S \in \mathcal{R}((X^{\text{nc}} \times X^{\text{in}}), (Y^{\text{nc}} \times Y^{\text{out}})) \\
 & X^{\text{in}} = Y^{\text{out}} \\
 & (x, y) \in \theta(S) \iff \exists z \in Y^{\text{out}} : ((x, z), (y, z)) \in S
 \end{aligned}$$

When looking at real-world examples of ensembles, we see that there are many cases, where the interaction of smaller systems leads to more pervasive changes in the states and behaviors of the resulting system. If we consider, for example, a single robot R moving in a flat arena, we can describe its state by a two-dimensional vector for its position and an angle for its orientation. Rotating the wheels will result in corresponding changes in position and orientation (assuming that the friction between wheels and ground is sufficiently high). If R is part of an

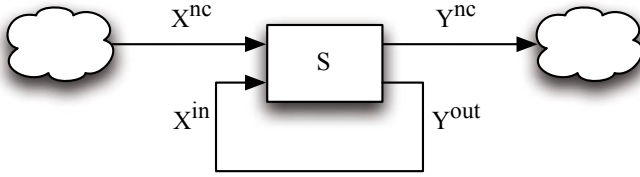


Fig. 3. Feedback loop θ

ensemble these assumptions may no longer be true. For example, another robot in the ensemble may be lifting R , so R 's position in three-dimensional space has to be described by three positional and three angular parameters. Furthermore, rotating the wheels may now not have any effect, or a strongly reduced effect if R is attached to another robot. The above definition of combination operators allows for these kinds of complex and non-compositional effects.

4 Adaptation

Having defined the basic notion of ensembles we now turn our attention to adaptation. There are several notions of adaptation which take into account increasingly large amounts of knowledge about the system's internals and its implementation. We restrict the following observations to the most abstract view that regards the system as a "black box."

4.1 Network and Environment

In the rest of this section we assume that a combination operator \otimes is given; this operator combines three systems (which may themselves be composed of simpler systems): an environment η which we regard as mostly outside the control of the system designer, a "network" or sensor/actuator system ν which simulates or represents the connection of the ensemble to the environment, and the system S . We assume that S is designed by the software developer and that it has to achieve a certain goal in the given environment or in a range of environments in order to be considered successful. To simplify the notation we often write η, ν, S for $\otimes(\eta, \nu, S)$, and we assume that all systems involved can be given the input/output configuration required by \otimes .

Example 6 (Robots operating in an environment). For ensembles S that can be given the input/output configuration of S_1^{rsw} in Ex. 1 we can define \otimes that combines an environment $\eta \in \mathcal{R}(Y_{\text{light}})$ by building a cascade of the environment's output with the input X_{light} of $S \in \mathcal{R}(X_{\text{light}}, Y_{\text{pos}} \times Y_{\text{bat}})$ while ignoring the network ν :

$$\begin{aligned} \otimes &: \mathcal{R}(Y_{\text{light}}) \times \{\emptyset\} \times \mathcal{R}(X_{\text{light}}, Y_{\text{pos}} \times Y_{\text{bat}}) \rightarrow \mathcal{R}(Y_{\text{pos}}, Y_{\text{bat}}) \\ \otimes(\eta, \emptyset, S) &= \eta \triangleright S \end{aligned}$$

In the following we write η_r^P for the environment containing the single function $\mu_r^P : \mathcal{I}^+ \rightarrow (\mathcal{I}^2 \rightarrow \mathcal{I})$ defined as

$$\mu_r^P(t)(x, y) = \begin{cases} 1 & \text{if } d((x, y), P) < r \\ 0 & \text{otherwise} \end{cases}$$

The function μ_r^P illuminates a circle of radius r around P during the whole duration of the experiment. Then, using the component S^{prog} from Ex. 5 that causes each robot to run with fixed algorithm **prog**, the system

$$\eta_r^P, \emptyset, S^{\text{prog}} \triangleright S_2^{\text{rsw}}$$

models running the robots with fixed algorithm **prog** in an environment in which the light source is fixed in a circle with radius r around P .

For given values of P and r the environment η_r^P is a system that exhibits a single deterministic, time-invariant behavior. Often we are interested in environments that allow more variety, either by changing over time or by allowing several different behaviors to occur. Both extensions can easily be achieved in our model:

If P and r are not constant but functions of t we obtain a deterministic, dynamically changing environment where the location of the light source and its radius may vary over time. If we define $\eta_{\geq r}(P_1, \dots, P_n)$ as the relation that contains all $\mu_{\rho}^{P_i}$ for $1 \leq i \leq n$ and $\rho \geq r$ we obtain an environment that models all time-invariant circular light distributions with radius at least r around one of the points P_i . By combining these two extensions we obtain an environment which is neither deterministic nor time invariant. \square

The ability to model non-determinism in a system and in its environment is important even for systems without physical components. For example, in the actor model [15] non-determinism may arise because the arrival-order of messages does not necessarily correspond to the order in which they were sent.

Consider a system S which is executed in environment η with network ν . In order to realize the given specification or goal γ the system has to satisfy following properties:

$$\eta, \nu, S \not\models \perp \tag{3}$$

$$\eta, \nu, S \models \gamma \tag{4}$$

Property (3) is necessary since an inconsistent systems vacuously satisfies any goal γ , property (4) immediately expresses the fact that the system satisfies the goal in question.

Example 7 (Goal satisfaction in an environment). If $S = S^{\text{prog}} \triangleright S_2^{\text{rsw}}$ satisfies

$$\eta_r^P, \emptyset, S \not\models \perp \tag{5}$$

$$\eta_r^P, \emptyset, S \models \gamma_2 \tag{6}$$

(see Ex. 1 and 2), the ensemble can adapt to the goal that no robot runs out of energy before t_0 in an environment where the light is concentrated in a circle of radius r around P when all robots use algorithm `prog`. If S satisfies equations (5) and (6) with $\eta_{\geq r}(P_1, \dots, P_n)$ instead of η_r^P , it can adapt to goal γ_2 for infinitely many environments. \square

Example 8 (Sensor accuracy in robot swarms). To illustrate the effects of the network, we now suppose that the swarm of solar-powered robots should not only ensure that the robots are charged all the time, but we also give the robots the task of cleaning up the environment: We assume that there are boxes randomly distributed throughout the environment and that the robots should move all boxes into a corner of the arena. Therefore η now contains distributions of boxes in addition to light distributions, and the robots contain an additional input X_{box} representing the location of the boxes. In practice X_{box} is obtained by sensors which give only partial information about the true location of boxes in the environment, depending on the quality of the robot's sensors, the interference with other sensors, the weather conditions, etc. This can easily be modeled by the sensor/actuator system ν : let ν_{err}^{range} be such a system that provides as input to each robot the location of all boxes within *range* units of distance, with an error distribution *err*. Varying the values of these parameters while keeping the environment constant gives again rise to a preorder that specifies how well an ensemble of robots can perform in a single environment using different sensors or in different weather conditions. \square

4.2 Adaptation Space

Usually we do not speak of adaptation when a system works in just a single, deterministic environment; we expect an adaptive system to work in a variety of different situations. We have seen in Ex. 6 that this can be achieved by having an environment with non-deterministic and time-invariant behavior. However it often simplifies the comparison and analysis of different systems if we do not merge different behaviors into a single environment and instead model adaptation by having several environments, e.g., η and η' such that the system satisfies the goal for all environments:

$$\eta, \nu, S \models \gamma \quad \text{and} \quad \eta', \nu, S \models \gamma$$

Different network conditions may be modeled in a similar way.

Adaptation to a new environment or network is not the only possible kind of adaptation; it might also be necessary to change the goals that an ensemble pursues while leaving the environment constant, or, in other words, the system may have to adapt to new requirements. In that case we obtain the adaptation condition

$$\eta, \nu, S \models \gamma'$$

If the new goal γ' is implied by the old goal γ , any system satisfying γ already satisfies γ' . Otherwise it is, in general, not possible for a system to adapt to

new goals unless this goal is communicated to the system, either by one of the unconnected inputs of the system or, more frequently, by changes in the environment. For example, we might change the color of a beacon from green to red to signal to a foraging robot that we want it to stop foraging and return to the base station. Therefore, it is usually not sensible to request that a system can adapt to any change in environment and goals, we rather have to restrict adaptation to these scenarios where the goal is correctly communicated to the ensemble.

To formalize these notions we define an *adaptation domain* \mathcal{A} that describes a range of environments \mathcal{E} , networks \mathcal{N} and goals \mathcal{G} , to which we want the system to adapt and define the notion *S can adapt to \mathcal{A}* , written $S \Vdash \mathcal{A}$:

$$\mathcal{A} \subseteq \mathcal{E} \times \mathcal{N} \times \mathcal{G}$$

$$S \Vdash \mathcal{A} \iff \forall (\eta, \nu, \gamma) \in \mathcal{A} : \eta, \nu, S \models \gamma$$

The *adaptation space* \mathfrak{A} is a set of adaptation domains, $\mathfrak{A} \subseteq \mathfrak{P}(\mathcal{E} \times \mathcal{N} \times \mathcal{G})$. It is partially ordered by inclusion; for any adaptation space we define a preorder of adaptivity for systems as follows:

$$S \sqsubseteq S' \iff \forall \mathcal{A} \in \mathfrak{A} : S \Vdash \mathcal{A} \implies S' \Vdash \mathcal{A}$$

In that case we say that S' is *at least as adaptive as S* (with respect to \mathfrak{A}). If $S \sqsubseteq S'$ and there is an adaptation domain $\mathcal{A} \in \mathfrak{A}$ for which $S' \Vdash \mathcal{A}$ but $S \not\Vdash \mathcal{A}$ then we say that S is *less adaptive than S'* or that S' is *more adaptive than S* (with respect to \mathfrak{A}) and write $S \sqsubset S'$.

Example 9 (Adaptivity to different environments). Let \mathcal{A}_r be the set containing all environments that shine a circular light with radius at least r at a fixed spot P , i.e., $\mathcal{A}_r = \{(\eta_\rho^P, \emptyset, \gamma_2) \mid \rho \geq r\}$. Let $\mathfrak{A} = \{\mathcal{A}_r \mid r > 0\}$ and let S_1 and S_2 be two robots swarms with input/output configurations $(X_{\text{light}}, Y_{\text{pos}} \times Y_{\text{bat}})$. Obviously for all ensembles S with this configuration we have the following property: if $r_1 > r_2$ and $S \Vdash \mathcal{A}_{r_2}$ then also $S \Vdash \mathcal{A}_{r_1}$. Suppose there is a value r_0 such that $S_1 \Vdash \mathcal{A}_{r_0}$ but $S_2 \not\Vdash \mathcal{A}_{r_0}$. Then S_1 is more adaptive than S_2 . \square

If we are not only interested in systems satisfying a specification but rather in systems that additionally maximize some performance criteria, we can extend \mathfrak{A} to

$$\mathfrak{A} \subseteq \mathfrak{P}(\mathcal{E} \times \mathcal{N} \times \mathcal{G} \times \mathcal{F}_\otimes)$$

where \mathcal{F}_\otimes is a set of objective functions with suitable domain for \otimes . We write $\text{dom}(\phi)$ for the domain and $\text{cod}(\phi)$ for the range of $\phi \in \mathcal{F}_\otimes$. For $(\eta, \nu, \gamma, \phi) = A \in \mathcal{A} \in \mathfrak{A}$ we write $\mathcal{H}(A, S)$ for the heterostatic system

$$\mathcal{H}((\eta, \nu, \gamma, \phi), S) = (\otimes(\eta, \nu, S), \gamma, \text{cod}(\phi), \phi)$$

Using the previously defined weak heterostatic preorder \preceq we can then define a preorder \sqsubseteq for the adaptive sets

$$S \sqsubseteq S' \iff \forall \mathcal{A} \in \mathfrak{A} : \forall A \in \mathcal{A} : \mathcal{H}(A, S) \preceq \mathcal{H}(A, S')$$

with the obvious condition for $S \sqsubset S'$.

Example 10 (Fitness in different environments). Let $\phi : Y_{\text{pos}} \times Y_{\text{bat}} \rightarrow \mathcal{I}$ be the function that returns the fraction of robots whose battery level never reaches 0 in the interval $[0, t_0]$, let $\mathcal{A}_r = \{(\eta_\rho^P, \emptyset, \text{true}, \phi) \mid \rho \geq r\}$ and let $\mathfrak{A} = \{\mathcal{A}_r \mid r > 0\}$. This definition is similar to Ex. 9, but now the requirement that no robot stops working because of an empty battery is a soft constraint that influences our evaluation of the fitness. Given robot swarms S_1, S_2 , we have $S_1 \preceq S_2$ if for all \mathcal{A}_r and all $A \in \mathcal{A}_r$ we have $\mathcal{H}(A, S_1) \sqsubseteq \mathcal{H}(A, S_2)$, i.e., if for all radii r at least as many robots in S_1 as in S_2 fail because of an empty battery. \square

By equipping the space of environments \mathcal{E} with a topology we can define a pre-order on ensembles that expresses the sensitivity of an ensemble’s adaptivity to small changes in each environment in a similar way; if we introduce a uniformity (or a metric) on \mathcal{E} we can compare ensembles by their global sensitivity to changes across environments. Similarly, we can vary \mathcal{N} to compare the sensitivity of ensembles to errors in the sensor readings.

5 Conclusions and Further Work

We have presented a model for systems that can represent a wide range of different ensembles in a uniform manner, that allows the description of system goals and requirements using a wide variety of different logics and that can be used to define and compare the fitness of ensembles using arbitrary preorders. Using this model we have defined the notion of “black-box adaptivity” and shown that this gives rise to a preorder of adaptivity (to a range of environments, networks or sensors/actuators, and requirements) among ensembles.

The work presented in this paper is only a first step toward a comprehensive system model for ensembles, with many theoretical and practical questions remaining future research: A variety of important topics, such as temporal determinacy, sensitivity to changes in the environment or network, controllability, and stability have not been mentioned at all or only been hinted at. We will investigate how existing approaches can be transferred to our system model as well as some new approaches to these questions that might be made possible by our approach. Another interesting question is reflexivity: currently introspection and reflexivity are implicit in the definition of the relations; given the importance of “self-awareness” for ensembles a more explicit treatment might be desirable.

While our approach allows the representation of non-deterministic ensembles and environments it does not directly support probabilistic behaviors. Various existing approaches for stochastic modeling exist and have proven their usefulness in the specification of systems, e.g., PEPA [31], StoKLAIM/MoSL [32,33] or PMAude [34]. We plan to extend our approach to probabilistic models by defining an additional joint probability distributions over the system sets $(V_i)_{i \in I}$. Another possibility could be to use stochastic relations [35].

In the current paper we have presented the theory without using categorical concepts. This “concrete” presentation could be streamlined, at the cost of some additional abstraction, by rephrasing the theory in the language of categories. In particular, since we allow different logics for the specification of

properties, exploring the relationship of our approach with heterogeneous institutions [36,37,38] would be an obvious possibility for generalization.

For lack of space we have only introduced the notion of black-box adaptation in this paper; in the future we will also define models for adaptation that take into account more internals of the ensemble under consideration—gray-box and white-box adaptation.

On the more practical side we will investigate connections between the denotational model presented in this paper and operational models and tools, such as process calculi and model checkers. Another important task for practical applications is the definition of a language that allows the expression of ensemble models in a more concise manner than purely mathematical notation.

Acknowledgements. We are grateful to Gul Agha, Olivier Danvy and José Meseguer for inviting us to contribute to this volume. Our thanks go to Franco Zambonelli for inspiring discussions about adaptation, and to the anonymous referees and Gul Agha for their helpful comments on a draft version of this paper.

This work has been partially sponsored by the EU project ASCENS, 257414.

References

1. InterLink Project, <http://interlink.ics.forth.gr/central.aspx> (last accessed 2011-05-10)
2. Hölzl, M., Rauschmayer, A., Wirsing, M.: Engineering of software-intensive systems: State of the art and research challenges. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) *Soft-Ware Intensive Systems*. LNCS, vol. 5380, pp. 1–44. Springer, Heidelberg (2008)
3. Wirsing, M., Denker, G., Talcott, C.L., Poggio, A., Briesemeister, L.: A rewriting logic framework for soft constraints. *ENTCS* 176(4), 181–197 (2007)
4. Hölzl, M., Meier, M., Wirsing, M.: Which soft constraints do you prefer? *ENTCS* 238(3), 189–205 (2009)
5. Hölzl, M., Denker, G., Meier, M., Wirsing, M.: Constraint-Muse: A soft-constraint based system for music therapy. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) *CALCO 2009*. LNCS, vol. 5728, pp. 423–432. Springer, Heidelberg (2009)
6. Goguen, J.A.: Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science* 2(2), 159–191 (1992)
7. Mesarović, M.D., Takahara, Y.: *General Systems Theory: Mathematical Foundations*. Mathematics in Science and Engineering, vol. 113. Academic Press, New York (1975)
8. Mesarović, M.D.: Mathematical theory of general systems. In: Chillingworth, D. (ed.) *Proceedings of the Symposium on Differential Equations and Dynamical Systems*. Lecture Notes in Mathematics, vol. 206, pp. 14–15. Springer, Heidelberg (1971)
9. Naundorf, H.: A general model for object-based systems. Preprint, Universität Paderborn (July 1995), http://www-math.uni-paderborn.de/preprints/preprints_data/Naundorf/naungMobS1.ps.gz

10. Broy, M.: Mathematical system models as a basis of software engineering. In: van Leeuwen, J. (ed.) *Computer Science Today*. LNCS, vol. 1000, pp. 292–306. Springer, Heidelberg (1995)
11. Broy, M.: A logical basis for component-oriented software and systems engineering. *Comput. J.* 53(10), 1758–1782 (2010)
12. Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., Winter, S.: Formalizing the notion of adaptive system behavior. In: Shin, S.Y., Ossowski, S. (eds.) *SAC*, pp. 1029–1033. ACM (2009)
13. Arbab, F.: Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.* 14, 329–366 (2004)
14. Lopes, A., Fiadeiro, J.L.: Revisiting the categorical approach to systems. In: Kirchner, H., Ringeissen, C. (eds.) *AMAST 2002*. LNCS, vol. 2422, pp. 426–440. Springer, Heidelberg (2002)
15. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press (1986)
16. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. *J. Funct. Program.* 7(1), 1–72 (1997)
17. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): *All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic*. LNCS, vol. 4350. Springer, Heidelberg (2007)
18. Kim, M., Stehr, M.O., Talcott, C.L., Dutt, N.D., Venkatasubramanian, N.: A probabilistic formal analysis approach to cross layer optimization in distributed embedded systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) *FMOODS 2007*. LNCS, vol. 4468, pp. 285–300. Springer, Heidelberg (2007)
19. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design* 29(3), 253–293 (2006)
20. Brock, J.P.: *The Evolution of Adaptive Systems: The General Theory of Evolution*. Academic Press (2000)
21. Klopff, A.H.: Brain function and adaptive systems—a heterostatic theory. Special Report 133, Air Force Cambridge Res. Lab., USAF (March 1972)
22. Schütte, K.: *Vollständige Systeme modaler und intuitionistischer Logik. Ergebnisse der Mathematik und ihrer Grenzgebiete*, vol. 42. Springer, Heidelberg (1968)
23. Huth, M., Ryan, M.: *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd edn. Cambridge University Press, New York (2004)
24. Kosiuczenko, P., Wirsing, M.: Timed rewriting logic with an application to object-based specification. *Sci. Comput. Program.* 28(2-3), 225–246 (1997)
25. Lamport, L.: Specifying Systems. In: *The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley (2002)
26. Bonani, M., Longchamp, V., Magnenat, S., Rtoraz, P., Burnier, D., Roulet, G., Vaussard, F., Bleuler, H., Mondada, F.: The MarXbot, a Miniature Mobile Robot Opening new Perspectives for the Collective-robotic Research. In: *Int. Conf. on Intel. Robots and Systems (IROS)*, pp. 4187–4193. IEEE Press (2010)
27. Pinciroli, C., Trianni, V., O’Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Caro, G.D., Ducatelle, F., Stirling, T., Gutiérrez, A., Gambardella, L.M., Dorigo, M.: ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics. Technical Report TR/IRIDIA/2011-009, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium (2011)
28. Chushov, I.D.: Introduction to the Theory of Infinite-Dimensional Dissipative Systems. *Acta* (2002)

29. Plotkin, G.D.: Domains (Pisa Notes) (1983)
30. Plotkin, G.D.: A powerdomain construction. *SIAM J. of Computing* (1976)
31. Hillston, J.: Process algebras for quantitative analysis. In: *LICS*. IEEE Computer Society, pp. 239–248 (2005)
32. De Nicola, R., Katoen, J.P., Latella, D., Massink, M.: STOKLAIM: A Stochastic Extension of KLAIM. Technical Report 2006-TR-01, DSIF, Firenze (2006)
33. De Nicola, R., Katoen, J.P., Latella, D., Loretì, M., Massink, M.: MoSL: A Stochastic Logic for StoKlaim. Technical Report ISTI-06-35, DSIF, Firenze (2006)
34. Agha, G.A., Meseguer, J., Sen, K.: Pmaude: Rewrite-based specification language for probabilistic object systems. *Electr. Notes Theor. Comput. Sci.* 153(2), 213–239 (2006)
35. Doberkat, E.E.: Stochastic Relations: Foundations for Markov Transition Systems. *Studies in Informatics*. Chapman & Hall/CRC (2007)
36. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. *J. ACM* 39(1), 95–146 (1992)
37. Mossakowski, T.: Heterogeneous Specification and the Heterogeneous Tool Set. Habilitation thesis, Universität Bremen (2005)
38. Cengarle, M.V., Knapp, A., Tarlecki, A., Wirsing, M.: A heterogeneous approach to UML semantics. In: Degano, P., Nicola, R.D., Meseguer, J. (eds.) *Concurrency, Graphs and Models*. LNCS, vol. 5065, pp. 383–402. Springer, Heidelberg (2008)

Algorithmic Aspects of Risk Management

Ashish Gehani¹, Lee Zaniewski², and K. Subramani²

¹ SRI International

² West Virginia University

Abstract. Risk analysis has been used to manage the security of systems for several decades. However, its use has been limited to offline risk computation and manual response. In contrast, we use risk computation to drive changes in an operating system's security configuration. This allows risk management to occur in real time and reduces the window of exposure to attack. We posit that it is possible to protect a system by reducing its functionality temporarily when it is under siege. Our goal is to minimize the tension between security and usability by trading them dynamically. Instead of statically configuring a system, we aim to monitor the risk level, using it to drive the tradeoff between security and utility. The advantage of this approach is that it provides users with the maximum possible functionality for any predefined level of risk tolerance.

Risk management can be framed as an exercise in managing the constraints on edge and vertex weights of a tripartite graph, with the partitions corresponding to the threats, vulnerabilities, and assets in the system. If a threat requires a specific permission and affects a particular asset, an edge is added between the threat and the permission that mediates access to the vulnerable resource. Another edge is added between the permission and the asset. The presence of a path from a threat, through a permission check, to an asset contributes an element of risk. Risk can be reduced by denying access to a resource that contains a vulnerability or activating data protection measures. We analyze some of the problems that form the algorithmic underpinnings of optimal risk management.

1 Introduction

The frequency of attacks faced by the average host connected to the Internet remains elevated, making reliance on manual intervention for response decreasingly tenable. Operating system and application based mechanisms for automated response have increasing utility in this context. We analyze algorithmic aspects of a framework for systematic fine-grained response that is achieved by dynamically controlling the host's exposure to perceived threats and limiting the consequences of security breaches.

Maintaining the security of a host requires it to be continually monitored. When there is suspicion that an attack may be underway, it is prudent to effect a response. The first course of action would be to interrogate the runtime environment to obtain finer-grain data to cross-check the audit information that

raised the alarm. If the suspicion remains, the next step would be to reconfigure the system (potentially reducing functionality) to limit the exposure of portions that may be vulnerable to the attack in progress. Data that may be affected by the attack should be safeguarded. Measures should be taken to ensure the confidentiality, integrity, and availability of the data after a successful attack. Finally, an effort should be made to gather and preserve forensic information from the environment that may not be available later.

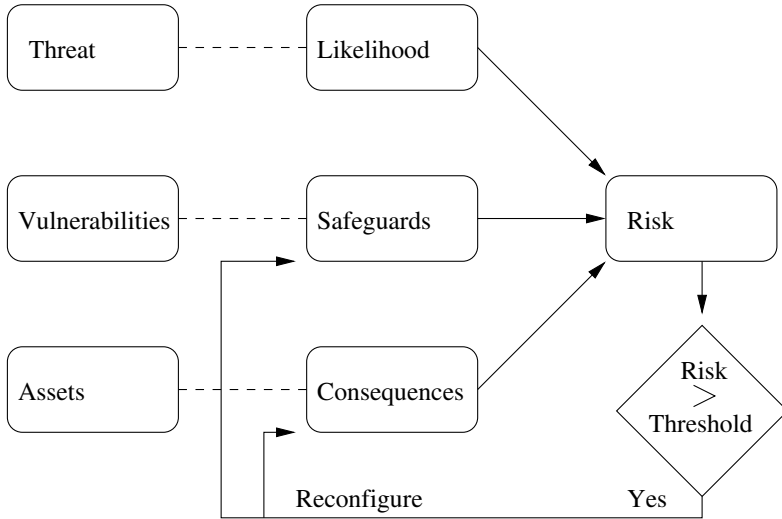


Fig. 1. Risk can be analyzed as a function of the threats, their likelihood, vulnerabilities, safeguards, assets, and consequences. Risk can be managed by using the safeguards to control the exposure of vulnerabilities and manipulating the assets to limit the consequences.

We equate protecting a system with minimizing the risk it faces. The risk is dependent on three factors. The first is the set of threats it faces and their likelihood of occurring. If there are no threats to the system, then it is not at risk. The second factor is the set of vulnerabilities that exist in the system, along with the probability of these being exposed. If there are no vulnerabilities, then even in the presence of a threat, no risk is posed to the system. The third factor is the consequence of an attack succeeding. If there is no consequence, then the system is not at risk.

Whereas threats are under the control of the attacker, vulnerabilities and consequences are within the control of, and can therefore be managed by, the defender. In contrast to previous approaches, we assume that a computation of risk will be used to drive changes in a system’s security posture, as depicted in Figure 1. This allows risk management to occur in real time to reduce the window of exposure. We posit that it is possible to protect a system by reducing

its functionality. Our goal is to minimize the tension between security and usability by trading them dynamically. Instead of statically configuring a system, we aim to monitor the risk level, using it to drive the tradeoff between security and utility. The advantage of this approach is that it provides users with the maximum possible functionality for any predefined level of risk tolerance.

2 Risk Model

We now describe some aspects of our risk model, omitting several algorithmic issues covered in previously published work [9] [10] [11] [13], where we discussed mechanisms to efficiently recalculate the risk, subtle reasons for modeling risk tolerance the way we do, how to track the costs and benefits in real time, and how to adapt the model for risk relaxation to improve system performance without exceeding the threshold of risk tolerance.

2.1 Runtime Risk Factors

We model risk as the flow between the first and last partitions in a tripartite graph, depicted in Figure 2, where T is a partition of vertices t_i each representing a unique threat, W is a partition of vertices w_j each representing a specific weakness in the system, and O is the partition of assets, with the vertices o_k each representing a data object.

Analyzing the risk that a system is faced with requires knowledge of a number of factors. Below we describe each of these factors along with its associated semantics. We define these in the context of the operating system paradigm since our goal is to manage the risk of a host.

Threats. A *threat* is an entity that can cause harm to an asset in the system. We define a threat to be a specific attack against any of the application or system software that is running on the host. It is characterized by an intrusion detection signature. The set of threats is denoted by $T = \{t_1, t_2, \dots\}$, where $t_\alpha \in T$ is an intrusion detection signature. Since t_α is a host-based signature, it is composed of an *ordered set* of events $S(t_\alpha) = \{s_1, s_2, \dots\}$. If this set occurs in the order recognized by the rules of the intrusion detector, it signifies the presence of an attack.

Likelihood. The *likelihood* of a threat is the hypothetical probability of its occurring. If a signature is partially matched, the extent of the match predicts the chance that it will later be completely matched. A function μ is used to compute the likelihood of threat t_α . μ can be threat-specific and depends on the history of system events that are relevant to the intrusion signature. Thus, if $E = \{e_1, e_2, \dots\}$ denotes the ordered set of all events that have occurred, then $\mathcal{T}(t_\alpha) = \mu(t_\alpha, E \overset{\sim}{\cap} S(t_\alpha))$ where $\overset{\sim}{\cap}$ yields the set of all events that occur *in the same order* in each input set.

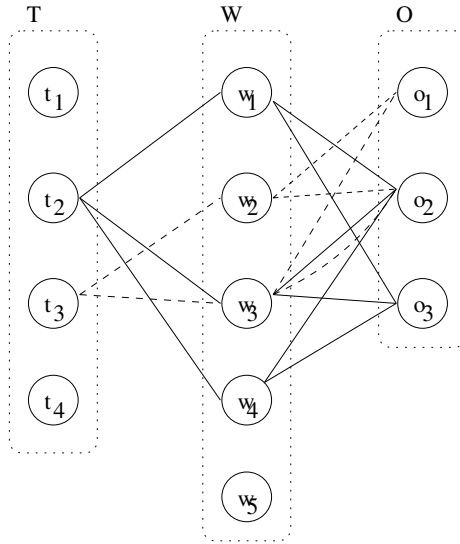


Fig. 2. Operating system risk can be modeled in terms of its constituent components. The threats, weaknesses (corresponding to specific vulnerabilities), and objects (that are the assets) form three disjoint sets. An edge between vertices represents a contribution to the system risk. The system’s risk is the total flow between the first and third sets.

Assets. An *asset* is an item that has value. We define the assets as the data stored in the system. In particular, each file is considered a separate object $o_\beta \in O$, where $O = \{o_1, o_2, \dots\}$ is the set of assets. A set of objects $A(t_\alpha) \subseteq O$ is associated with each threat t_α . Only objects $o_\beta \in A(t_\alpha)$ can be harmed if the attack that is characterized by t_α succeeds.

Consequences. A *consequence* is a type of harm that an asset may suffer. Three types of consequences can impact the data. These are the loss of confidentiality, integrity, and availability. If an object $o_\beta \in A(t_\alpha)$ is affected by the threat t_α , then the resulting costs due to the loss of confidentiality, integrity, and availability are denoted by $c(o_\beta)$, $i(o_\beta)$, and $a(o_\beta)$ respectively. Any of these values may be 0 if the attack cannot effect the relevant consequence. However, all three values associated with a single object cannot be 0, since in that case $o_\beta \in A(t_\alpha)$ would not hold. Thus, the consequence of a threat t_α is $C(t_\alpha) = \sum_{o_\beta \in A(t_\alpha)} c(o_\beta) + i(o_\beta) + a(o_\beta)$.

By removing an asset from the system, the consequences it faces can be *curtailed* [13]. In the case of data availability, replication serves this purpose, while in the case of confidentiality and integrity, cryptographic operations can be used. For the purpose of estimating risk, a consequence *curtailment* effectively removes the asset from the analysis.

Vulnerabilities. A *vulnerability* is a weakness in the system. It results from an error in the design, implementation, or configuration of either the operating system or application software. The set of vulnerabilities present in the system is denoted by $W = \{w_1, w_2, \dots\}$. $W(t_\alpha) \subseteq W$ is the set of weaknesses exploited by the threat t_α to subvert the security policy.

Safeguards. A *safeguard* is a mechanism that controls the exposure of the system's assets. The reference monitor's set of permission checks $P = \{p_1, p_2, \dots\}$ serve as safeguards in an operating system. Since the reference monitor mediates access to all objects, a vulnerability's exposure can be limited by denying the relevant permissions. The set $P(w_\gamma) \subseteq P$ contains all the permissions that are requested in the process of exploiting vulnerability w_γ .

The static configuration of a conventional reference monitor either grants or denies access to a permission p_λ . This *exposure* is denoted by $v(p_\lambda)$, with the value being either 0 or 1. An *active reference monitor* [11][12] allows each permission to be associated with an independent set of constraints that are verified at runtime before granting the permission. By limiting the circumstances under which the permission will be granted, the exposure of the resource being protected is reduced by a predetermined fraction.

The active reference monitor can therefore reduce the exposure of a statically granted permission to $v'(p_\lambda)$, a value in the range $[0, 1]$. This reflects the nuance that results from evaluating predicates as *auxiliary safeguards*. Thus, if all auxiliary safeguards are used, the total exposure to a threat t_α is $\mathcal{V}(t_\alpha) = \sum_{p_\lambda \in \hat{P}(t_\alpha)} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|}$ where $\hat{P}(t_\alpha) = \bigcup_{w_\gamma \in W(t_\alpha)} P(w_\gamma)$.

In practice, since the set of threats cannot be altered by the response apparatus, we can merge the first partition, which contains the threats, into the second by scaling each permission's weight (which represents its probability of being granted) with the sum of the threat likelihoods that have incident edges on the permission.

2.2 Risk Management

The risk to the host is the sum of the risks that result from each of the threats that it faces. The risk from a single threat is the product of the chance that the attack will occur, the exposure of the system to the attack, and the cost of the consequences of the attack succeeding [18]. Thus, the cumulative risk faced by the system is $\mathcal{R} = \sum_{t_\alpha \in T} \mathcal{I}(t_\alpha) \times \mathcal{V}(t_\alpha) \times \mathcal{C}(t_\alpha)$.

If the risk posed to the system is to be managed, the current level must be continuously monitored. When the risk rises past the threshold that the host can tolerate, the system's security must be tightened. Similarly, when the risk decreases, the restrictions can be relaxed to improve performance and usability.

The system's risk can be reduced either by reducing the exposure of vulnerabilities or by limiting the consequences to the data in the event of a successful attack. The former is effected through the use of auxiliary safeguards before granting a permission. The latter is realized by cryptographically protecting and remotely replicating threatened files. Both approaches may also be used simultaneously.

The set of permissions P is kept partitioned into two disjoint sets, $\Psi(P)$ and $\Omega(P)$, that is, $\Psi(P) \cap \Omega(P) = \phi$ and $\Psi(P) \cup \Omega(P) = P$. The set $\Psi(P) \subseteq P$ contains the permissions for which auxiliary safeguards are currently active. The remaining permissions $\Omega(P) \subseteq P$ are handled conventionally by the reference monitor, using only static lookups rather than evaluating associated predicates before granting these permissions. Similarly, the set of files O is kept partitioned into two disjoint sets, $\Psi(O)$ and $\Omega(O)$, where $\Psi(O) \cap \Omega(O) = \phi$ and $\Psi(O) \cup \Omega(O) = O$. The set $\Psi(O) \subseteq O$ contains the files that are currently inaccessible and unmodifiable due to their cryptographic encapsulation. The remaining files $\Omega(O) \subseteq O$ are transparently accessible and modifiable.

At any given point, when safeguards $\Psi(P)$ and curtailments $\Psi(O)$ are in use, the current risk \mathcal{R}' is calculated with $\mathcal{R}' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}'(t_\alpha) \times \mathcal{C}'(t_\alpha)$ where

$$\mathcal{V}'(t_\alpha) = \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Omega(P)} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} + \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Psi(P)} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|}$$

and

$$\mathcal{C}'(t_\alpha) = \sum_{o_\beta \in A(t_\alpha) \cap \Omega(O)} c(o_\beta) + i(o_\beta) + a(o_\beta).$$

2.3 Response Selection

The risk level *after* an event occurs is denoted by \mathcal{R}_a . If this increases past the threshold of risk tolerance \mathcal{R}_0 , the goal of the response engine is to reduce the risk by $\delta_g \geq \mathcal{R}_a - \mathcal{R}_0$ to a level below the threshold. To do this, it must select a subset of permissions $\rho(\Omega(P)) \subseteq \Omega(P)$ and a subset of objects $\rho(\Omega(O)) \subseteq \Omega(O)$, such that adding safeguards and curtailments respectively to the two sets will reduce the risk to the desired level. The resulting risk level is reduced to $\mathcal{R}'' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}''(t_\alpha) \times \mathcal{C}''(t_\alpha)$ where the new vulnerability measure is

$$\mathcal{V}''(t_\alpha) = \sum_{p_\lambda \in (\hat{P}(t_\alpha) \cap \Omega(P) - \rho(\Omega(P)))} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} + \sum_{p_\lambda \in (\hat{P}(t_\alpha) \cap \Psi(P) \cup \rho(\Omega(P)))} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|}$$

and the new consequence measure is

$$\mathcal{C}''(t_\alpha) = \sum_{o_\beta \in (A(t_\alpha) \cap \Omega(O) - \rho(\Omega(O)))} c(o_\beta) + i(o_\beta) + a(o_\beta).$$

2.4 Performance Sensitivity

The choice of safeguards and curtailments also impacts the performance of the system. Evaluating predicates before granting permissions introduces latency in system calls. Cryptographically protecting objects decreases usability. Hence, the choice of subsets $\rho(\Omega(P))$ and $\rho(\Omega(O))$ or subsets $\rho(\Psi(P))$ and $\rho(\Psi(O))$ is subject to the secondary goal of minimizing the overhead introduced.

The adverse impact of a safeguard or curtailment is proportional to the frequency with which it is used in the system's workload. Given a typical workload, we can count the frequency $f(p_\lambda)$ with which permission p_λ is requested in the workload. Similarly, we can count the frequency $f(o_\beta)$ with which file o_β is accessed in the workload. This can be done for all permissions and files. The cost of using subsets $\rho(\Omega(P))$ and $\rho(\Omega(O))$ for risk reduction can then be calculated with

$$\zeta(\rho(\Omega(P)), \rho(\Omega(O))) = \sum_{p_\lambda \in \rho(\Omega(P))} f(p_\lambda) + \sum_{o_\beta \in \rho(\Omega(O))} f(o_\beta).$$

2.5 Abstracting the Problem

The ideal choice of safeguards and curtailments minimizes the safeguards' and curtailments' impact on performance, while simultaneously ensuring that the risk remains below the threshold of tolerance. Thus, for risk reduction we wish to find:

minimize: $\zeta(\rho(\Omega(P)), \rho(\Omega(O)))$ subject to: $\mathcal{R}'' \leq \mathcal{R}_0$

Risk management can be viewed as an exercise in picking vertices from the second and third partitions of Figure 2, that need to be protected. Since the set of threats and their likelihoods cannot be altered by the response apparatus, we can merge the first partition, which contains the threats, into the second by scaling each vulnerability's weight with the sum of the threat likelihoods that have incident edges on it. We note that the semantics of risk management require that at each step, the risk must be reduced below the threshold of tolerance. This precludes optimization strategies such as minimizing a weighted sum of risk and runtime performance.

3 On the Hardness of Risk Management

We describe results that provide insights into the algorithmic hardness of the risk management problem.

3.1 Integral Costs and Benefits

When performance-sensitive runtime risk management is viewed as a 0/1 integer nonlinear programming problem with a linear objective function and a quadratic

constraint, it gives rise to a range of related graph problems. For example, consider the problem of selecting a set of responses such that the total cost of effecting them is below a threshold T_1 , and simultaneously ensuring that the residual risk is below T_2 when the costs and benefits are integers. Since the costs correspond to the frequency with which a resource is accessed in the workload, the costs are positive integers. In scenarios where the risk associated with each edge is derived by counting the frequency with which the asset (associated with one vertex that the edge is incident upon) is accessed through the permission (associated with the other vertex that the edge is incident upon), the edge weights are also positive integers. This can be defined as the following problem \mathbf{P}_1 :

Problem 1 (\mathbf{P}_1). Given a graph $G = \langle V, E, \mathbf{p}, \mathbf{w} \rangle$ with V denoting the set of vertices, E denoting the set of edges, $\mathbf{w} : V \rightarrow Z$ denoting a weighting function from the vertices to the set of positive integers, and $\mathbf{p} : E \rightarrow Z$ denoting a weighting function from the set of edges to the set of positive integers, a vertex threshold T_1 and an edge threshold T_2 , is there a subset of vertices V' such that $\sum_{v \in V'} w(v) \leq T_1$ and $\sum_{e=(u,v); u,v \notin V'} p(e) \leq T_2$?

Alternatively, the risk management algorithm could attempt to select a set of responses that would impose a cost less than the threshold T_1 but subject to the constraint that the resulting risk reduction would exceed threshold T_2 (where any response primitive chosen would eliminate all risk contributions that depended on access to the targeted permission or asset). This can be formulated as the problem \mathbf{P}_2 :

Problem 2 (\mathbf{P}_2). Given a graph $G = \langle V, E, \mathbf{p}, \mathbf{w} \rangle$ with V denoting the set of vertices, E denoting the set of edges, $\mathbf{w} : V \rightarrow Z$ denoting a weighting function from the vertices to the set of positive integers, and $\mathbf{p} : E \rightarrow Z$ denoting a weighting function from the set of edges to the set of positive integers, a vertex threshold T_1 and an edge threshold T_2 , is there a subset of vertices V' such that $\sum_{v \in V'} w(v) \leq T_1$ and $\sum_{e=(u,v); u \in V' \text{ or } v \in V'} p(e) \geq T_2$?

The two problems \mathbf{P}_1 and \mathbf{P}_2 can be seen to be identical. Implementing a solution for one therefore immediately provides a mechanism to address the other. The equivalence can be seen since an instance of \mathbf{P}_1 can be represented as an instance of \mathbf{P}_2 by replacing T_2 with $\sum_{e \in E} w(e) - T_2$ and vice versa. An important point to note about \mathbf{P}_2 is that if a vertex in $V - V'$ does not have any incident edges, then it is automatically included in V' .

3.2 Independent Vulnerabilities and Consequences

In our initial investigation, we found that even simplifications of the performance-sensitive runtime risk management problem are algorithmically hard to solve. For

example, consider the case where every attack relies on a single vulnerability and affects a single asset. The corresponding graph is a *matching*. Optimal response selection in this scenario is algorithmically expensive as shown below:

Theorem 1. \mathbf{P}_2 is NP-complete even if G is a matching.

Proof. We reduce the 0/1 knapsack problem to \mathbf{P}_2 . The knapsack problem is known to be NP-complete [8].

An instance of the knapsack problem is characterized by n objects $O = \{o_1, o_2, \dots, o_n\}$ with respective profits $\{p_1, p_2, \dots, p_n\}$ and respective integer weights $\{w_1, w_2, \dots, w_n\}$, a knapsack capacity W and a profit target T . The goal is to pack objects into the knapsack so as to obtain a profit of at least T , while ensuring that the sum of the weights of the objects is at most W . Without loss of generality, we can assume that the weights are even integers.

Given the knapsack instance, we construct the following instance of \mathbf{P}_2 . Corresponding to object O_i , create two vertices v_i and v_{n+i} and an edge connecting them with weight p_i . The two vertices are given weight $\frac{w_i}{2}$ each. The vertex threshold is set at W and the edge threshold is set at T .

We claim that the knapsack instance is a “yes” instance if and only if the \mathbf{P}_2 instance is.

Assume that the given knapsack instance is a “yes” instance, i.e., there is a set of objects $O' \subseteq O$, such that $\sum_{y:y \in O'} w(y) \leq W$ and $\sum_{y:y \in O'} p(y) \geq K$. Pick the vertices in the \mathbf{P}_2 instance that correspond to these objects. As per the construction, the vertex threshold of these vertices is at most W and the edge threshold is at least T .

Now assume that the \mathbf{P}_2 instance is a “yes” instance, i.e., there is a collection of vertices whose combined weight is at most W and the sum of the weights of the edges connected to these vertices is at least K . As per the construction of the \mathbf{P}_2 instance, if vertex v_i is picked, then so is vertex v_{n+i} . Further, the contribution of these two vertices to the vertex threshold is w_{n+i} and to the edge threshold is p_i . Consider the objects corresponding to the picked vertex pairs. As per the construction, their weights sum to at most W and their profits sum to at least K . □

3.3 Qualitative Exposures and Consequences

Instead of considering the case when each vulnerability affects a different asset in the system, we extended the scope of the problem to consider the result when each vulnerability could affect multiple assets and each asset could be affected by multiple vulnerabilities. We restrict the problem to the case where only qualitative knowledge about the vulnerabilities and consequences in the system is available, with the result that a vertex exists for each vulnerability and asset in the system, but it is unweighted.

Since only their absence or presence is known, an unweighted edge between the permission guarding a vulnerability and the object affected by the consequence is inserted only when the vulnerability and consequence are both present. To ensure that the risk remains below a predefined threshold, vertices can be removed by deactivating the corresponding permissions or curtailing the relevant consequences. The result is that an edge incident on any of the removed vertices would itself be removed from the graph, reducing the risk. This is formulated as problem \mathbf{P}_3 :

Problem 3 (\mathbf{P}_3). Given a bipartite graph with unweighted vertices and unweighted edges, find the smallest set of vertices, subject to the constraint that the number of edges remaining after the vertices are removed is below a predefined threshold.

3.4 Known Workloads

The formulation of \mathbf{P}_3 did not account for the frequency with which each response primitive occurs in the workload. In practice, the frequency with which the safeguard or data protection primitive is invoked affects its impact on performance. Picking primitives with lower frequencies is therefore preferable. When a workload is known in advance, the problem can be formulated as \mathbf{P}_4 :

Problem 4 (\mathbf{P}_4). Given a bipartite graph with weighted vertices and unweighted edges, find the set of vertices with the lowest sum of vertex weights, subject to the constraint that the number of edges remaining after the vertices are removed is below a predefined threshold.

3.5 Dynamic Application Workloads

We can generalize the risk model from the case where exposure and consequences are considered only qualitatively – that is, only their presence or absence is known, to the case where an estimate of their degree is known. If the degree is estimated with an integer, then the risk contributed by the presence of each exposure and consequence pair is also an integer (since it is the product of two integers). Therefore the edges in the bipartite graph constructed to represent the risk has integer weights.

In general, if the target application workload is known *a priori*, information gleaned from it can be used to optimize the choice of risk management responses. The approach comes with the caveat that predicting a target workload may be nontrivial. In particular, past workloads may not be available and even if they are, they may not be representative of future tasks. Additionally, if the target workload has high variance – that is, if it dynamically and significantly changes its characteristics, then the use of average frequencies for vertex weights

can result in distorted tradeoffs between cost and benefit estimates of selecting specific responses. In such a situation, we can factor out performance sensitivity by using unweighted vertices. The corresponding formulation is \mathbf{P}_5 :

Problem 5 (\mathbf{P}_5). Given a bipartite graph with unweighted vertices and weighted edges, find the set of vertices with the lowest sum of vertex weights, subject to the constraint that the number of edges remaining after the vertices are removed is below a predefined threshold.

We show \mathbf{P}_5 is *NP-complete* by reducing the *vertex cover problem* to it. Recall that this determines whether it is possible to construct a *cover* of a specified size, where a cover is a subset of vertices with the property that every edge in the graph has at least one end incident upon one of the vertices in the cover.

Theorem 2. \mathbf{P}_5 is NP-complete.

Proof. Assume we wish to check whether a cover of size k exists for graph G . We construct a bipartite graph B with disjoint partitions π_1 and π_2 as follows.

For each vertex v_i in G , we add four vertices, $v_{i,1}$, $v_{i,2}$, $v_{i,3}$, and $v_{i,4}$ to B . $v_{i,1}$ and $v_{i,3}$ are inserted in π_1 while $v_{i,2}$ and $v_{i,4}$ are inserted in π_2 . An edge between $v_{i,3}$ and $v_{i,4}$ is added to B and given weight 1. A second edge, between $v_{i,1}$ and $v_{i,4}$, and a third edge, between $v_{i,2}$ and $v_{i,3}$, are also added to B . The second and third edges are each given weight $k + 1$.

For each edge (v_i, v_j) in G , we add two edges with weight $k + 1$ to B . The first is between $v_{i,1}$ and $v_{j,2}$ and the second is between $v_{i,2}$ and $v_{j,1}$. This is illustrated in Figure 3.

The target number of vertices to be removed is set to $\rho = \frac{|\pi_1| + |\pi_2|}{2}$, that is, half the total number of vertices in B . We run our algorithm for \mathbf{P}_5 on B with threshold k . If we can remove ρ vertices subject to the constraint that the total weight of the remaining edges is below the threshold k , then there exists a vertex cover of size k for G .

The reason the reduction holds is as follows. Since the edges connecting $v_{i,1}$ to $v_{i,4}$ and $v_{i,2}$ to $v_{i,3}$ have weights that exceed the threshold, either $v_{i,1}$ or $v_{i,4}$ and either $v_{i,2}$ or $v_{i,3}$ must be removed for the total weight of the remaining edges to be below the threshold. Since half of the vertices can be removed from the graph, exactly one vertex is removed from each of the pairs. To remain below the threshold, it is necessary to remove all the edges in B that were added in correspondence to the edges in G . Specifically, if v_i is a vertex in the cover of G , then $v_{i,1}$ and $v_{i,2}$ must be removed from B for the threshold constraint to be maintained. Since only one of the vertices $v_{i,1}$ and $v_{i,4}$ and only one of $v_{i,2}$ and $v_{i,3}$ can be removed, if $v_{i,1}$ and $v_{i,2}$ are selected for removal, then $v_{i,3}$ and $v_{i,4}$ must remain in B along with the edge between the two. Conversely, any group

of four vertices ($v_{i,1}, v_{i,2}, v_{i,3}$, and $v_{i,4}$) in B that corresponds to a vertex v_i in G that is not in the cover can have either $v_{i,2}$ or $v_{i,3}$ removed without increasing the total weight of the edges. Thus, the only way half the vertices of B can be removed while the total weight of the edges remains below the threshold k is if the corresponding vertices in G form a cover of size k . \square

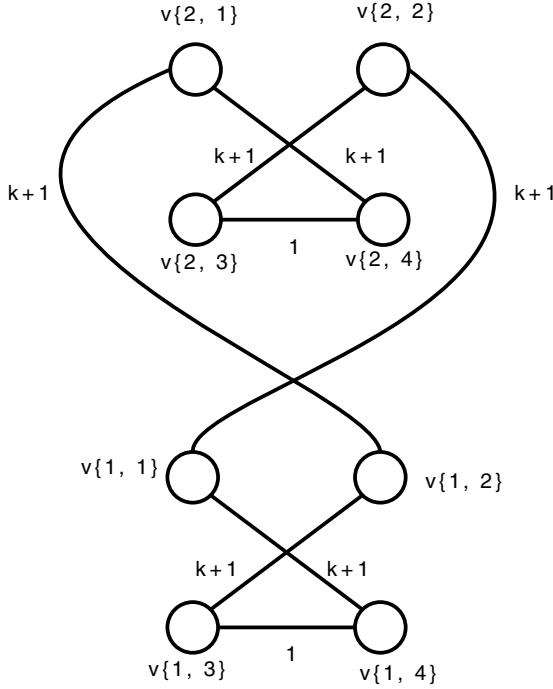


Fig. 3. Given a graph with two vertices, v_1 and v_2 , and an edge between them, this is the corresponding bipartite graph

4 Open Questions

Although problems \mathbf{P}_1 , \mathbf{P}_2 , and \mathbf{P}_3 are the minimally and maximally constrained versions of \mathbf{P}_4 and \mathbf{P}_5 , the complexity of \mathbf{P}_1 , \mathbf{P}_2 , and \mathbf{P}_3 remains to be analyzed. In addition, efficient solutions (and approximation algorithms, as needed) must be designed for these problems. Further, variants of the problems where edges (representing risk) can have real values instead of integers when an active monitor (modeled in Section 2) is used to dynamically limit the exposure of the system, or the consequences are estimated with fractional values.

Cascading Dependencies

Our experience developing a prototype risk manager for the operating system paradigm uncovered the problem of *cascading dependencies*. If the semantics

of risk reduction through edge removal requires both vertices that the edge is incident upon to also be removed, then the effect may be a cascade of edge removal. The total risk reduction that results by selecting a set of responses is not just the risk reduction corresponding to the sum of the edge weights of the induced subgraph. Instead, it also includes the risk from the sum of the edges that have a single end incident on any vertex in the set of selected responses. This significantly complicates the problem, since it may potentially introduce a cascading set of dependencies, all of which must be examined to determine the optimal choice of edges. Figure 4 illustrates the issue.

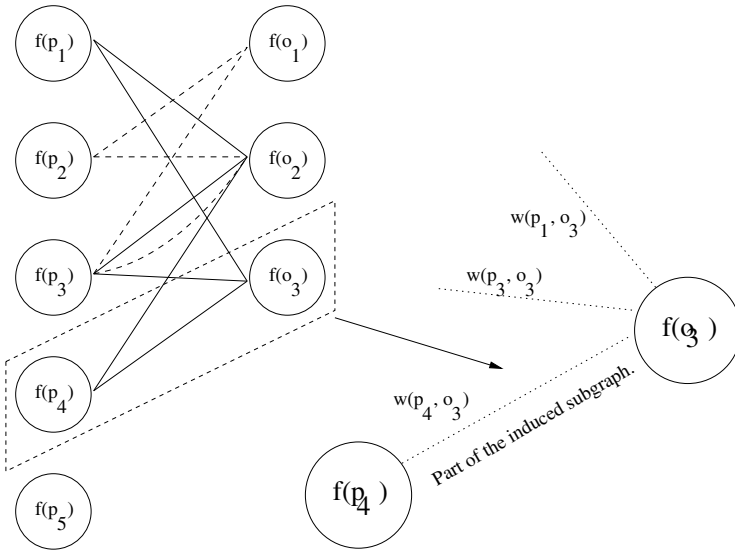


Fig. 4. When edge (p_4, o_3) is removed, if the semantics require that both vertices that it is incident upon must be inactivated, then the dependency cascade would leave only p_5 operational

5 Related Work

The effort to manage the risk of information systems can be traced to the use of the Annual Loss Expectancy (ALE) metric [6][7] by large data processing centers. The use of the ALE paradigm by commercial tools [19], coupled with a focused research effort [2][3][4][5], resulted in improvements in risk modeling. Although risk analysis has been used to manage the security of systems for several decades [6], its use has been limited to offline risk computation and manual response. SooHoo [20] proposed a general model using decision analysis to estimate computer security risk and automatically update input estimates. Bilar [1] used reliability modeling to analyze the risk of a distributed system. Risk is calculated as a function of the probability of faults being present in the system's

constituent components. Risk management is framed as an integer linear programming problem, aiming to find an alternate system configuration, subject to constraints such as acceptable risk level and maximum cost for reconfiguration.

Problems \mathbf{P}_1 through \mathbf{P}_5 are being proposed for the first time, although variants have been studied and described earlier. The 0/1 1-dimensional knapsack problem is called a *weakly NP-complete* problem, since it admits algorithms whose running times are polynomial if the problem parameters are represented in unary. Such algorithms are called pseudo-polynomial algorithms. Ibarra and Kim presented a pseudo-polynomial time algorithm [14] for the knapsack problem that has been used as the basis for the design of an efficient fully polynomial approximation scheme for the knapsack problem. Efficient approaches to unidimensional and multidimensional variants of the knapsack have also been developed [17][16]. This paper shows that problems \mathbf{P}_1 through \mathbf{P}_5 are NP-complete.

In the prototype implemented [9], a heuristic was used to guarantee that the risk is maintained below a threshold. Although approximation algorithms exist [15], they were not employed since the choices had to be made in real time. The heuristic used is based on the greedy algorithm for the *0-1 Knapsack Problem* that yields a solution that is always within a factor of 2 of the optimal choice [8]. When the risk needs to be reduced, the heuristic uses the greedy strategy of picking the response primitive with the highest benefit-to-cost ratio repeatedly until the constraint is satisfied. By maintaining the choices in a *heap* data structure keyed on the benefit-to-cost ratio, each primitive in the response set can be chosen in $O(1)$ time. This is significant, since implementing a single response primitive is often sufficient to disrupt an attack in progress. A separate *heap* is used to maintain the active safeguards keyed by the cost-to-benefit ratio instead. When the risk needs to be relaxed, the active safeguards with the highest cost-to-benefit ratios can be selected, since these yield the best improvement to system performance. A future avenue of research is empirical comparison of the approximation and NP-complete algorithms to the heuristic.

6 Conclusion

We note that the semantics of risk management require that the risk be reduced below the threshold of tolerance each time it is found to exceed it. Risk can be reduced by denying access to a resource that contains a vulnerability or by activating data protection measures. This is modeled as the removal of edges representing risk in the aforementioned graph. Depending on whether the risk estimates are integers or reals, whether the vulnerabilities and consequences are independent or conditional, whether the application workload is known in advance, whether the workload is stable or changing rapidly, and depending on the semantics of response selection, there are different underlying graph problems. We analyzed some of the problems that form the algorithmic underpinnings of optimal risk management.

References

1. Bilar, D.: Quantitative Risk Analysis of Computer Networks, Ph.D. Thesis, Dartmouth College (2003)
2. 1st Computer Security Risk Management Model Builders Workshop, Martin Marietta, Denver, Colorado, National Bureau of Standards (May 1988)
3. 2nd Computer Security Risk Management Model Builders Workshop, AIT Corporation, Ottawa, Canada, National Institute of Standards and Technology (June 1989)
4. 3rd International Computer Security Risk Management Model Builders Workshop, Los Alamos National Laboratory, Santa Fe, New Mexico, National Institute of Standards and Technology (August 1990)
5. 4th International Computer Security Risk Management Model Builders Workshop, University of Maryland, College Park, Maryland, National Institute of Standards and Technology (August 1991)
6. Guidelines for Automatic Data Processing Physical Security and Risk Management, National Bureau of Standards (1974)
7. Guidelines for Automatic Data Processing Risk Analysis, National Bureau of Standards (1979)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco (1979)
9. Gehani, A.: Support for Automated Passive Host-based Intrusion Response, PhD thesis, Duke University (2003)
10. Gehani, A.: Performance-sensitive Real-time Risk Management is NP-Hard. In: Workshop on Foundations of Computer Security affiliated with the 19th IEEE Symposium on Logic in Computer Science (2004)
11. Gehani, A., Kedem, G.: RheoStat: Real-time Risk Management. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 296–314. Springer, Heidelberg (2004)
12. Gehani, A., Kedem, G.: Real-time Access Control Reconfiguration. In: International Infrastructure Survivability Workshop affiliated with the 25th IEEE International Real-Time Systems Symposium (2004)
13. Gehani, A., Chandra, S., Kedem, G.: Augmenting Storage with an Intrusion Response Primitive to Ensure the Security of Critical Data. In: 1st ACM Symposium on Information, Computer and Communications Security (2006)
14. Ibarra, O., Kim, C.: Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM* 22(4) (1975)
15. Kellerer, H., Pferschy, U.: A new fully polynomial approximation scheme for the knapsack problem. In: Jansen, K., Rolim, J.D.P. (eds.) APPROX 1998. LNCS, vol. 1444, pp. 123–134. Springer, Heidelberg (1998)
16. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Heidelberg (2004)
17. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley and Sons, New York (1990)
18. Guidelines for Automatic Data Processing Physical Security and Risk Management, National Institute of Standards and Technology (1996)
19. Description of Automated Risk Management Packages that NIST/NCSC Risk Management Research Laboratory Has Examined, National Institute of Standards and Technology (1991)
20. Hoo, K.S.: Guidelines for Automatic Data Processing Physical Security and Risk Management, Ph.D. Thesis, Stanford University (2002)

Parameterized Metareasoning in Membership Equational Logic^{*}

Manuel Clavel, Narciso Martí-Oliet, and Miguel Palomino

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
{clavel,narciso,miguelpt}@sip.ucm.es

Abstract. Basin, Clavel, and Meseguer showed in [1] that membership equational logic is a good metalogical framework because of its initial models and support of reflective reasoning. A development and an application of those ideas was presented later in [4]. Here we further extend the metalogical reasoning principles proposed there to consider classes of parameterized theories and apply this reflective methodology to the proof of different parameterized versions of the deduction theorem for minimal logic of implication.

1 Motivation

A *reflective* logic is a logic in which important aspects of its metalogic can be represented at the object level in a consistent way, so that the object-level representations correctly simulate the relevant metalogical aspects. As a consequence, in a reflective logic, metatheorems involving families of theories can be represented and logically proved as theorems about its universal theory. Basin, Clavel, and Meseguer showed in [1] that logical frameworks can be good metalogical frameworks when their theories always have initial models and they support reflective and parameterized reasoning; they also showed that membership equational logic is a particular logical framework that satisfies these requirements. In this paper, we extend their ideas and apply them to the (parameterized) deduction theorem.

Basin and Matthews have shown in [2] how metatheories based on inductive definitions can be used to formalize metatheorems that are *parameterized* with their scope of application. As a case study, they formalize different parameterized versions of the deduction theorem in the theory FS_0 [8]; we will use the same case study to motivate the developments of the following sections.

We can use membership equational logic (described in more detail in Section 2) to represent theoremhood in a logic as a sort in a theory. Conditional membership axioms then directly support the representation of rules as schemas, which is typically used in presenting logics and formal systems. Similarly, we can

^{*} Research supported by Spanish projects DESAFIOS10 TIN2009-14599-C03-01 and PROMETIDOS S2009/TIC-1465.

represent theoremhood in a parameterized family of logics as a sort in a parameterized theory. A sort in a parameterized membership equational theory can be used to represent theoremhood in a family of logics if and only if there is a correspondence between logics in the family and instances of the parameterized theory. Moreover, this correspondence has to be such that theoremhood in a logic in the family can be represented as membership in this sort in the corresponding instance of the parameterized theory.

We shall now illustrate the above idea using minimal logic (of implication) as a running example. Representing minimal logic in membership equational logic entails defining a theory T that conservatively represents minimal logic's theoremhood. The formulae of minimal logic correspond to members of the set built from the binary connective \rightarrow (written infix, associating to the right) and sentential constants. Theorems correspond to members of a second set, and are either instances of the standard Hilbert axiom schemas K ,

$$A \rightarrow B \rightarrow A,$$

and S ,

$$(A \rightarrow B) \rightarrow (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow C),$$

```
fth MINIMAL is
kind Symbol[].
kind Expression[SentConstant Formula Theorem].
***** kinds
*** ** Symbol
op <ASCII-identifiers> : -> Symbol .
*** ** Expression
op <integer> : -> Expression .
op [_,_,_] : Symbol Expression Expression -> Expression .

vars A B C : Expression .
***** sorts
*** ** SentConstant
mb <integers> : SentConstant .
*** ** Formula
cmb A : Formula if A : SentConstant .
cmb [->, A, B]: Formula if A : Formula /\ B : Formula .
*** ** ** Theorem
cmb [->, A, [->, B, A]] : Theorem
  if A : Formula /\ B : Formula .
cmb [->, [->, A, B], [->, [->, [A, [->, B, C]]], [->, A, C]] : Theorem
  if A : Formula /\ B : Formula /\ C : Formula .
cmb B : Theorem
  if A : Formula /\ B : Formula
  /\ A : Theorem /\ [->, A, B]: Theorem .
endfth
```

Fig. 1. The theory MINIMAL

or are generated by applying the *modus ponens* rule,

$$\frac{A \quad A \rightarrow B}{B}.$$

Then, the deduction theorem for minimal logic is a metatheorem that states that

$$\text{if } \vdash_A B \text{ then } \vdash A \rightarrow B,$$

where \vdash denotes that a formula can be deduced in minimal logic from the rules above and \vdash_A is provability when A is considered to be an additional axiom. Since A is arbitrary, this result is a statement about a family of logics (or theories); actually, the result is also parametric in another sentence since it holds for extensions of minimal logic with additional connectives, like the standard conjunction.

The theory **MINIMAL**—in short, **ML**—in Figure 1 represents minimal logic in membership equational logic using the above idea. The lines starting with **kind** declare the kinds and their associated sorts; for the time being, kinds can be safely ignored. The sort **Formula** represents the well-formed formulae in minimal logic, in the sense that any formula in minimal logic can be represented as a term of this sort and vice versa. For example, if A, B are sentential constants represented respectively by 1 and 2, then $(A \rightarrow B)$ is represented by the term $[-\rightarrow, 1, 2]$ of sort **Formula**. Similarly, the sort **Theorem** represents the theorems in minimal logic, so that any theorem in minimal logic can be represented as a term of this sort, and vice versa.

Consider now the task of representing not just minimal logic, but the family of logics that includes any extension of minimal logic with respect to its language—connectives and syntactic rules—and proof system—axioms and inference rules. A solution to this is given by the parameter theory **EXTENDED-MINIMAL**—in short, **EML**—in Figure 2. The parametric sort **@NewSynRule** allows us to capture the extensions of minimal logic’s language with new binary connectives. For example, the extension of minimal logic’s language with the \wedge -operator corresponds to the instantiation of **EML** with the following membership axiom $Ax(@NewSynRule)$ associated to **@NewSynRule**:

```
mb [[/\, A, B], A, B]: @NewSynRule .
```

Similarly, the parametric sorts **@NewAxiom** and **@NewInfRule** allow us to capture the extensions of minimal logic’s proof system with new axioms and/or new inference rules of two premises. For example, the extension of minimal logic’s proof system with the axiom schemas for the binary connective \wedge corresponds to the instantiations of **EML** with the following membership axioms associated to **@NewAxiom**:

```
mb [-\rightarrow, A, [-\rightarrow, B, [/\, A, B]]]: @NewAxiom .
mb [-\rightarrow, [/\, A, B], A]: @NewAxiom .
mb [-\rightarrow, [/\, A, B], B]: @NewAxiom .
```

```

fth EXTENDED-MINIMAL is
including MINIMAL .
kind Expression[@NewAxiom] .
kind Rule[@NewSynRule @NewInfRule] .
***** kinds
*** ** Rule
op [_ , _ , _] : Expression Expression Expression -> Rule .

vars A B C : Expression .
***** sorts
*** ** Formula
cmb A : Formula
  if [A, B, C] : @NewSynRule
  /\ B : Formula /\ C : Formula .
*** ** Theorem
cmb A : Theorem if A : @NewAxiom /\ A : Formula .
cmb A : Theorem
  if [A, B, C] : @NewInfRule
  /\ A : Formula /\ B : Formula /\ C : Formula
  /\ B : Theorem /\ C : Theorem .
***** parameters
op @A : -> Expression .
mb @A : Formula .
endfth

```

Fig. 2. The theory EXTENDED-MINIMAL

```

fth EXTENDED-MINIMAL-DT[EXTENDED-MINIMAL] is
including EXTENDED-MINIMAL .
mb @A : Theorem .
endfth

```

Fig. 3. The theory EXTENDED-MINIMAL-DT[EXTENDED-MINIMAL]

Now, let $@A$ be the parametric constant that appears (as a subscript of \vdash) in the deduction theorem. The parameterized theory in Figure 3—in short, $DT[EML]$ —can be used to represent any extension of minimal logic with respect to its language and proof system.

With this example in mind, our objectives in this paper move at two different levels. First, we want to design a metareasoning principle over parameterized theories in membership equational logic; a concrete application of this principle would be a proof of the fact that the deduction theorem holds for every possible instantiation of $DT[EML]$. Secondly, and foremost, we intend to reify both parameterized theories and the metareasoning principle in the universal theory U_{MEL} of membership equational logic [6]; that is, our goal is to define representation functions to reify parameterized theories as terms in U_{MEL} and the

metareasoning principle as a formula over U_{MEL} . As a concrete application, we will show that the parameterized deduction theorem can be proved by showing that a certain formula holds in U_{MEL} .

2 Membership Equational Logic

Membership equational logic is an expressive version of equational logic. A full account of the syntax and semantics of membership equational logic can be found in [3,10]. Here we define the basic notions needed in this paper.

A *signature* in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, Σ a K -kinded signature $\Sigma = \{\Sigma_{k_1 \dots k_n, k}\}_{(k_1 \dots k_n, k) \in K^* \times K}$, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K -kinded family of sets. We call S_k the set of *sorts* of kind k and write $[s]$ for the kind of a sort s . The pair (K, Σ) is what is usually called a many-sorted signature of function symbols; however we call the elements of K *kinds* because each kind k now has a set S_k of associated *sorts*, which in the models will be interpreted as subsets of the carrier for the kind.

The atomic formulae of membership equational logic are *equations* $t = t'$, where t and t' are Σ -terms of the same kind, and *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. Sentences are Horn clauses on these atomic formulae, i.e., sentences of the form

$$\forall(x_1, \dots, x_m). A_0 \text{ if } A_1 \wedge \dots \wedge A_n$$

where each A_i is either an equation or a membership assertion, and each x_j is a K -kinded variable. A theory in membership equational logic is a pair (Ω, E) , where E is a finite set of sentences in membership equational logic over the signature Ω . We write $(\Omega, E) \vdash \phi$ to denote that (Ω, E) entails the sentence ϕ .

We employ standard semantics concepts from many-sorted logic. Given a signature $\Omega = (K, \Sigma, S)$, an Ω -*algebra* A is a many-kinded Σ -algebra (that is, a K -indexed-set $A = \{A_k\}_{k \in K}$ together with a collection of appropriately kinded functions interpreting the operators in Σ) and an assignment that associates to each sort $s \in S_k$ a subset $A_s \subseteq A_k$. As usual, we denote by T_Ω the K -kinded algebra of ground (K, Σ) -terms, and by $T_\Omega(X)$ the algebra of (K, Σ) -terms on the K -kinded set of variables X . An algebra A and a valuation σ , assigning to variables of kind k values in A_k , satisfy an equation $(\forall X) t = t'$ iff $\sigma(t) = \sigma(t')$, where we overload notation by identifying σ with its unique homomorphic extension to terms. We write $A, \sigma \models (\forall X) t = t'$ to denote such a satisfaction. Similarly, $A, \sigma \models (\forall X) t : s$ holds iff $\sigma(t) \in A_s$.

Note that an Ω -algebra is a K -kinded first-order model with function symbols Σ and a kinded alphabet of unary predicates $\{S_k\}_{k \in K}$. We can then extend the satisfaction relation to Horn and first-order formulae ϕ over the atomic formulae in the standard way. We write $A \models \phi$ when the formula ϕ is satisfied for all valuations σ , and then say that A is a model of ϕ . As usual, we write $(\Omega, E) \models \phi$ when all the models of the set E of sentences are also models of ϕ .

Theories in membership equational logic have initial models [10]. This provides the basis for reasoning by induction. In the initial model of a membership

equational theory, sorts are interpreted as the smallest sets satisfying the axioms in the theory, and equality is interpreted as the smallest congruence satisfying those axioms. Given a theory (Ω, E) , we denote its initial model by $T_{\Omega/E}$. In particular, when $E = \emptyset$ we obtain the term algebra T_{Ω} . We write $(\Omega, E) \models \phi$ to denote that the initial model of the membership equational theory (Ω, E) is also a model of ϕ , that is, that the satisfaction relation $T_{\Omega/E} \models \phi$ holds.

2.1 Reflection in Membership Equational Logic

A reflective logic is a logic in which important aspects of its metalogic can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metalogical aspects. More concretely, a logic is reflective when there exists a *universal* theory in which we can represent and reason about all finitely presentable theories in the logic, including the universal theory itself [5]. As a consequence, in a reflective logic, metatheorems involving families of theories can be represented and proved as theorems about its universal theory [1]. A universal theory U_{MEL} for membership equational logic is described in [6], along with a representation function $(\overline{_} \vdash \overline{_})$ that encodes pairs, consisting of a finitely presentable membership equational theory with nonempty kinds and a sentence in it, as sentences in U_{MEL} . The signature of U_{MEL} contains constructors to represent operators, variables, terms, kinds, sorts, signatures, axioms, and theories. In particular, the signature of U_{MEL} includes the kinds `[Op]`, `[Var]`, `[Term]`, `[TermList]`, `[Kind]`, `[Sort]`, and `[Theory]` for terms representing, respectively, operators, variables, terms, lists of terms, kinds, sorts, and theories. In addition, it contains three Boolean operators¹

```
op _::_in_ : [Term] [Kind] [Theory] -> [Bool] .
op _::_in_ : [Term] [Sort] [Theory] -> [Bool] .
op _=_in_ : [Term] [Term] [Theory] -> [Bool] .
```

to represent, respectively, that a term is a ground term of a given kind in a membership equational theory, and that a membership assertion or an equation holds in a membership equational theory.

The representation function $(\overline{_} \vdash \overline{_})$ is defined in [6] as follows: for all finitely presentable membership equational theories with nonempty kinds R , and atomic formulae ϕ over the signature of R ,

$$\overline{R} \vdash \phi \triangleq \begin{cases} (\overline{t} : \overline{s} \text{ in } \overline{R}) = \text{true} & \text{if } \phi = (t : s) \\ (\overline{t} = \overline{t'} \text{ in } \overline{R}) = \text{true} & \text{if } \phi = (t = t') \end{cases}$$

where $(\overline{_})$ is a representation function defined recursively over theories, signatures, axioms, and so on. In particular, to represent terms the signature of U_{MEL} contains the constructors

¹ The operator declarations have been changed slightly from those in [6] to better match their use in this work.

```

op _[_] : [Op] [TermList] -> [Term] .
op nil : -> [TermList] .
op _,_ : [TermList] [TermList] -> [TermList] .

```

and the representation function $\overline{(_)}$ is defined as follows:

$$\bar{t} \triangleq \begin{cases} \bar{c} & \text{if } t = c \text{ is a constant} \\ \bar{x} & \text{if } t = x \text{ is a variable} \\ \bar{f}[\bar{t}_1, \dots, \bar{t}_n] & \text{if } t = f(t_1, \dots, t_n). \end{cases} \quad (1)$$

For example, the term $s(0)$ of kind Num is represented in U_{MEL} as the term $\bar{s}[\bar{0}]$ of kind $[Term]$. Constants, operators, variables, kinds, and sorts are represented using strings of ASCII characters preceded by a quote. For example, $s(0)$ can be represented in U_{MEL} as the term `'s['0]`. It is convenient to represent variables along with their kinds using a binary constructor

```

op _|_ : [Var] [Kind] -> [Term] .

```

For example, $s(N)$ is represented in U_{MEL} as the term `'s['N'|'Num]`.

The following results state the main properties of U_{MEL} as a universal theory and are proved in [6]. We assume a finitely presentable membership equational theory $R = (\Omega, E)$ with nonempty kinds, and with $\Omega = (K, \Sigma, S)$.

Proposition 1. *For all terms t in T_Ω , and kinds k in K ,*

$$t \in (T_\Omega)_k \iff U_{MEL} \vdash (\bar{t} :: \bar{k} \text{ in } \bar{R}) = \mathbf{true}.$$

Furthermore, for all ground terms u of kind $[Term]$, if

$$U_{MEL} \vdash (u :: \bar{k} \text{ in } \bar{R}) = \mathbf{true},$$

then there is a term $t \in (T_\Omega)_k$ such that $\bar{t} = u$.

Proposition 2. *For all terms t, t' in $(T_\Omega)_k$ and sorts s in S_k ,*

$$\begin{aligned} R \vdash t : s &\iff U_{MEL} \vdash (\bar{t} : \bar{s} \text{ in } \bar{R}) = \mathbf{true} \\ R \vdash t = t' &\iff U_{MEL} \vdash (\bar{t} = \bar{t}' \text{ in } \bar{R}) = \mathbf{true}. \end{aligned}$$

Note that this proposition says that there exists a *logical* proof of $t : s$ (resp. of $t = t'$) in a membership equational theory R if and only if there exists also a *logical* proof of $(\bar{t} : \bar{s} \text{ in } \bar{R}) = \mathbf{true}$ (resp. of $(\bar{t} = \bar{t}' \text{ in } \bar{R}) = \mathbf{true}$) in the universal membership equational theory U_{MEL} .

Finally, not only can the theory U_{MEL} represent and reason about the entailment relation of any other theory but also about their own structure. In particular, we can define an operator

```

op _spec_in_ : [AxiomSet] [Sort] [Signature] -> [Bool] .

```

that distinguishes those axioms that specify a sort in a signature, in the following sense:

Proposition 3. *For any membership equational signature $\Omega = (K, \Sigma, S)$, any set of sentences Ax , and any sort s in some S_k , the following are equivalent:*

- $U_{\text{MEL}} \vdash (\overline{Ax} \text{ spec } \overline{s} \text{ in } \overline{\Omega}) = \text{true}$.
- Ax is a set of sentences over Ω that specify the sort s .

Proposition 4. *For any ground terms u, z , and M in U_{MEL} , if*

$$U_{\text{MEL}} \vdash (u \text{ spec } z \text{ in } M) = \text{true},$$

then there is a membership equational signature Ω , a sort s over Ω , and a set of sentences Ax in Ω specifying s , such that $\overline{\Omega} = M$, $\overline{Ax} = u$, and $\overline{s} = z$.

The proofs for these results would follow easily by mimicking the techniques for Propositions 1 and 2.

2.2 Reflecting an Inductive Principle

We need to introduce here some additional notation. For all terms $t \in T_\Omega(X)$, we denote by $\overline{t}^{[X]}$ the reflective representation of t defined in (1), except that now variables $x \in X$ are replaced by variables $\overline{x}^{[X]} = x$ of the kind $[\text{Term}]$, and we denote by $\overline{X}^{[X]}$ the set $\overline{X}^{[X]} \triangleq \{\overline{x}^{[X]} \mid x \in X\}$. The key difference between \overline{t} and $\overline{t}^{[X]}$ is that \overline{t} is a *ground term*, whereas $\overline{t}^{[X]}$ is a term of kind $[\text{Term}]$ with variables of the kind $[\text{Term}]$.

In addition, for all membership assertions $t:s$, with t in $T_\Omega(X)$ and s in some S_k ,

$$\overline{t:s}^{[R,X]} \triangleq (\overline{t}^{[X]} : \overline{s} \text{ in } \overline{R}) = \text{true},$$

and, similarly, for all equations $t = t'$, with t, t' in $T_\Omega(X)$,

$$\overline{t = t'}^{[R,X]} \triangleq (\overline{t}^{[X]} = \overline{t'}^{[X]} \text{ in } \overline{R}) = \text{true}.$$

Now we can define a representation function for metalogical statements that satisfies the expected property. Let $\{R_1, \dots, R_p\}$ be a set of membership equational theories, $\{k_1, \dots, k_n\}$ a finite multiset of kinds in $\{R_1, \dots, R_p\}$, $\mathbf{x} = \{x_1, \dots, x_n\}$ a finite set of variables, with each x_i of kind k_i , and τ a metalogical statement of the form

$$\forall t_1 \in (T_{\Omega_1})_{k_1} \dots \forall t_n \in (T_{\Omega_n})_{k_n} . \text{bexp}(R_1 \vdash \phi_1(\mathbf{t}), \dots, R_p \vdash \phi_p(\mathbf{t})), \quad (2)$$

where each $\phi_l(\mathbf{x})$ is an atomic Ω_l -formula with free variables in \mathbf{x} and bexp is a Boolean expression. Then,

$$\begin{aligned} \overline{\tau} &\triangleq \forall x_1. \dots \forall x_n. (((x_1 :: \overline{k}_1 \text{ in } \overline{R}_1) = \text{true} \wedge \dots \wedge (x_n :: \overline{k}_n \text{ in } \overline{R}_n) = \text{true}) \\ &\implies \text{bexp}(\overline{\phi_1(\mathbf{x})}^{[R_1, \mathbf{x}]}, \dots, \overline{\phi_p(\mathbf{x})}^{[R_p, \mathbf{x}]}) \end{aligned}$$

where $\{x_1, \dots, x_n\}$ are now variables of the kind $[\text{Term}]$. Now, the main result in [4] was:

Theorem 1. *Let τ be a metalogical statement of the form (2). Then, τ holds iff $U_{\text{MEL}} \models \overline{\tau}$.*

3 Parameterization

In the previous section we have recalled an inductive principle to reason about terms in a family of theories, which constitutes both an application of the ideas introduced in [1] as well as a generalization.² In this section we turn our attention to parameterization, which was already studied in [1] using the deduction theorem as a case study. Here we consider a generalization of the parameter theories and of the corresponding inductive principle, and we use them to formalize two versions of the deduction theorem not expressible in the formalisms presented in [1,4].

3.1 (Some) Parameterized Membership Equational Theories

As pointed out by Goguen and Burstall [9], a *parameterized theory* can be defined for logics in general as a pair of theories: the *parameter* P and the *body* T , that are related by a theory map $J : P \rightarrow T$ which is typically a theory inclusion. To *instantiate* such a parameterized theory, the key data needed is a theory morphism $H : P \rightarrow Q$ from the parameter theory to another theory Q . The *instantiation* by H is then defined as the pushout commutative diagram

$$\begin{array}{ccc}
 T & \xrightarrow{H^T} & T[H] \\
 J \uparrow & & \uparrow J^Q \\
 P & \xrightarrow{H} & Q
 \end{array}$$

in the category Th of theories and theory maps [9], when such a pushout exists.

Now we employ an instance of the previous construction to define, for each appropriate parameter theory P , a class \mathcal{P}_P of membership equational theories parameterized by P and a class \mathcal{V}_P of theory morphisms that instantiate parameterized theories in \mathcal{P}_P . For that, given two membership equational signatures Ω and Ω' , we will write $\Omega \cup \Omega'$ for the signature whose set of kinds is the set-theoretic union of those of Ω and Ω' , and whose operators and sorts are those of Ω and Ω' .

Then, we consider parameter theories P of the form

$$P = (\Omega \cup V \cup Z, E \cup Mb(V));$$

that is, P 's signature is built from

- a finite signature $\Omega = (K, \Sigma, S)$,
- a finite signature of parameters $V = (K, \{V_{\lambda,k}\}_{k \in K}, \emptyset)$, consisting of a pairwise disjoint K -kinded family of constants which satisfies that, for all $k \in K$, $\Sigma_{\lambda,k} \cap V_{\lambda,k} = \emptyset$, and

² The result proved in [4] also allowed to reason about equivalence classes of terms, which were not considered in [1].

- a finite signature of parameters $Z = (K, \emptyset, \{Z_k\}_{k \in K})$, consisting of a pairwise disjoint K -kinded family of sets which satisfies that, for all $k \in K$, $S_k \cap Z_k = \emptyset$;

and P 's axioms consist of

- a finite set of sentences E on terms in $T_\Omega(X)$, and
- a finite set of membership assertions $Mb(V)$ that specify a sort (possibly in Z) for each v in V .

Moreover, we consider theory maps $P \longrightarrow T$ which are *theory inclusions* and, for this reason, we usually denote parameterized theories by $T[P]$. Specifically, we define \mathcal{P}_P as the class of parameterized theories $T[P]$ of the form

$$T[P] = (\Omega' \cup V \cup Z, E \cup G \cup Mb(V)),$$

where $\Omega \subseteq \Omega'$ and G is a finite set of additional axioms (which extend P 's axioms). Note that for all parameter theories P there is a trivial extension $P[P]$ of P , namely, $P[P] = P$.

Now, let $Inst(P)$ be the class of theories

$$Q = (\Omega \cup V \cup Z, E \cup Eq(V) \cup Ax(Z)),$$

where

- $Eq(V)$ is a finite set of equations of the form

$$v = t \quad (v \in V),$$

assigning to each constant $v \in V$ a *ground* term $t \in T_\Omega$ such that $Q \vdash t : s$, where s is the sort assigned to v in $Mb(V)$, and

- $Ax(Z)$ is a finite set of membership axioms of the form

$$\forall (x_1, \dots, x_m). t : z \text{ if } A_1 \wedge \dots \wedge A_n,$$

where $z \in Z_k$ for some kind $k \in K$, t is a term over the signature Ω , and A_i is an atomic formula over the same signature, for $i = 1, \dots, n$. We collect all the axioms specifying a sort $z \in Z_k$ in a set $Ax(z)$.

We define \mathcal{V}_P as the class of theory morphisms $\beta : P \longrightarrow Q$ such that $Q \in Inst(P)$ and β is the identity signature morphism. Note that the set \mathcal{V}_P is in bijective correspondence with the set $Inst(P)$.

The above defines a notion of instantiation for parameterized theories that, for any $T[P] \in \mathcal{P}_P$ and $\beta \in \mathcal{V}_P$, specializes the pushout construction to

$$\begin{array}{ccc} T[P] & \longrightarrow & T[\beta] \\ \uparrow & & \uparrow \\ P & \xrightarrow{\beta} & Q \end{array}$$

where $T[\beta] = (\Omega' \cup V \cup Z, E \cup G \cup Eq(V) \cup Ax(Z))$.

One of the key ideas behind our use of theory morphisms is the following. Although β is the identity morphism on signatures, it identifies terms in Q , and hence in $T[\beta]$, by adding equations of the form $v = t$. This has an effect equivalent to mapping constants to terms. More formally, suppose $T[P] \in \mathcal{P}_P$ and $\beta \in \mathcal{V}_P$. For all terms $t \in T_{\Omega \cup V}(X)$, we denote by t_β the term in $T_\Omega(X)$ that results from replacing all parameters v in t by their instantiations in $Eq(V)$. We can extend this notion of term replacement to atomic formulae in the standard way: $(t : s)_\beta \triangleq t_\beta : s$ and $(t = t')_\beta \triangleq t_\beta = t'_\beta$. Note then that for all atomic formulae ϕ over the signature of $T[\beta]$, and due to the equations in $Eq(V)$, it holds that

$$T[\beta] \vdash \phi \iff T[\beta] \vdash \phi_\beta. \tag{3}$$

4 Induction Principles for Parameterized Theories

We next introduce an inductive metareasoning principle over parameterized theories. First, we need the following definition.

Definition 1. Let $P = (\Omega \cup V \cup Z, E \cup Mb(V))$ be a parameter theory with $\Omega = (K, \Sigma, S)$, let $\mathcal{P} = \{R_1[P], \dots, R_p[P]\}$ be a finite multiset of parameterized theories in \mathcal{P}_P , and $e \in [1..p]$. We say that \mathcal{P} is coherent modulo $R_e[P]$ if

- 1-a. every term t of kind $k \in K$ in $R_e[P]$ is also a term of kind k in $R_l[P]$ for $1 \leq l \leq p$, and
- 1-b. for all theory morphisms $\beta : P \rightarrow Q$ in \mathcal{V}_P , all terms t and t' of kind $k \in K$ in $R_e[P]$, and all $1 \leq l \leq p$, it holds that

$$R_e[\beta] \vdash t = t' \implies R_l[\beta] \vdash t = t'.$$

That is, we assume that among the parameterized theories in \mathcal{P} there is one that is “equationally generic” in the sense that, if an equation holds in any of its instances, then it also holds in the corresponding instance of any of the rest of the parameterized theories in \mathcal{P} . We can then use this distinguished theory to reason inductively about the whole family.

Proposition 5. Let $\mathcal{P} = \{R_1[P], \dots, R_p[P]\}$ be a finite multiset of parameterized theories in \mathcal{P}_P that is coherent modulo $R_e[P]$. Let $R_e[P] = (\Omega'_e \cup V \cup Z, E \cup G_e \cup Mb(V))$, let s be a sort in some S_k , and let $C_{[R_e[P],s]} = \{C_1, \dots, C_n\}$ be those sentences in $E \cup G_e$ that specify the sort s , i.e., those C_i of the form

$$\forall(x_1, \dots, x_{r_i}). A_0 \text{ if } A_1 \wedge \dots \wedge A_{q_i},$$

where, for some term w of kind k , A_0 is $w : s$.

Then, for all finite multisets of atomic formulae $\{\phi_l(x)\}_{l \in [1..p]}$ with free variable x of kind k , and Boolean expressions be_{xp} , the following metalogical statement holds:

$$\begin{aligned} &\forall \beta \in \mathcal{V}_P. (\psi_1 \wedge \dots \wedge \psi_n) \\ &\implies \\ &\forall \beta \in \mathcal{V}_P. (\forall t \in T_\Omega. (R_e[\beta] \vdash t : s \implies be_{xp}(R_1[\beta] \vdash \phi_1(t)_\beta, \dots, R_p[\beta] \vdash \phi_p(t)_\beta))) \end{aligned}$$

where, for $1 \leq i \leq n$ and C_i in $C_{[R_e[P],s]}$, ψ_i is

$$\forall t_1 \in (T_\Omega)_{k_{i_1}} \dots \forall t_{r_i} \in (T_\Omega)_{k_{i_{r_i}}} . [A_1]^\# \wedge \dots \wedge [A_{q_i}]^\# \implies [A_0]^\#$$

and, for $0 \leq j \leq q_i$,

$$[A_j]^\# \triangleq \begin{cases} \text{bexp}(R_1[\beta] \vdash \phi_1(u(\mathbf{t})), \dots, R_p[\beta] \vdash \phi_p(u(\mathbf{t}))) & \text{if } A_j = u:s \\ R_e[\beta] \vdash A_j(\mathbf{t}) & \text{otherwise.} \end{cases}$$

Actually, this proposition is a particular case of the following, more general one, that will be needed for the deduction theorem.

Proposition 6. *Let $\mathcal{P} = \{R_1[P], \dots, R_p[P]\}$ be a finite multiset of parameterized theories in \mathcal{P}_P that is coherent modulo $R_e[P]$. Let $R_e[P] = (\Omega'_e \cup V \cup Z, E \cup G_e \cup Mb(V))$, let s be a sort in some S_k , and let $C_{[R_e[P],s]} = \{C_1, \dots, C_n\}$ be those sentences in $E \cup G_e$ that specify the sort s , i.e., those C_i of the form*

$$\forall(x_1, \dots, x_{r_i}). A_0 \text{ if } A_1 \wedge \dots \wedge A_{q_i},$$

where, for some term w of kind k , A_0 is $w:s$.

Then, for all finite multisets of atomic formulae $\{\phi_l(x)\}_{l \in [1..p]}$ with free variable x of kind k , and Boolean expressions bexp , the following metalogical statement holds:

$$\begin{aligned} & \forall \beta \in \mathcal{V}_P. (U_{\text{MEL}} \models \bar{\beta}(\gamma) \implies \psi_1) \wedge \dots \wedge \forall \beta \in \mathcal{V}_P. (U_{\text{MEL}} \models \bar{\beta}(\gamma) \implies \psi_n) \\ & \implies \\ & \forall \beta \in \mathcal{V}_P. (U_{\text{MEL}} \models \bar{\beta}(\gamma) \implies \\ & \quad \forall t \in T_\Omega. (R_e[\beta] \vdash t : s \implies \text{bexp}(R_1[\beta] \vdash \phi_1(t)_\beta, \dots, R_p[\beta] \vdash \phi_p(t)_\beta))) \end{aligned}$$

where, for $1 \leq i \leq n$ and C_i in $C_{[R_e[P],s]}$, ψ_i is

$$\forall t_1 \in (T_\Omega)_{k_{i_1}} \dots \forall t_{r_i} \in (T_\Omega)_{k_{i_{r_i}}} . [A_1]^\# \wedge \dots \wedge [A_{q_i}]^\# \implies [A_0]^\#$$

and, for $0 \leq j \leq q_i$,

$$[A_j]^\# \triangleq \begin{cases} \text{bexp}(R_1[\beta] \vdash \phi_1(u(\mathbf{t})), \dots, R_p[\beta] \vdash \phi_p(u(\mathbf{t}))) & \text{if } A_j = u:s \\ R_e[\beta] \vdash A_j(\mathbf{t}) & \text{otherwise.} \end{cases}$$

Proof. Let $\beta \in \mathcal{V}_P$ be such that $U_{\text{MEL}} \models \bar{\beta}(\gamma)$ and let t be such that $R_e[\beta] \vdash t : s$. Then, we have to show that $\text{bexp}(R_1[\beta] \vdash \phi_1(t)_\beta, \dots, R_p[\beta] \vdash \phi_p(t)_\beta)$ is true.

We proceed by structural induction on the derivation of $R_e[\beta] \vdash t : s$. There exists a sentence C_i in $C_{[R_e[P],s]}$ and a substitution $\sigma : \{x_1, \dots, x_{r_i}\} \longrightarrow T_{\Omega_e}$, such that

- $R_e[\beta] \vdash t = \sigma(w)$, and
- $R_e[\beta] \vdash \sigma(A_j)$, for $1 \leq j \leq q_i$.

By hypothesis, $U_{\text{MEL}} \models \overline{\beta}(\gamma) \implies \psi_i$, and since we are assuming $U_{\text{MEL}} \models \overline{\beta}(\gamma)$, ψ_i must hold. But then, in particular, it also holds $[A_1]_{\sigma}^{\sharp} \wedge \dots \wedge [A_{q_i}]_{\sigma}^{\sharp} \implies [A_0]_{\sigma}^{\sharp}$, where, for $0 \leq j \leq q_i$,

$$[A_j]_{\sigma}^{\sharp} \triangleq \begin{cases} \text{bexp}(R_1[\beta] \vdash \phi_1([\sigma(u)]_{R_e}), \dots, R_p[\beta] \vdash \phi_p([\sigma(u)]_{R_e})) & \text{if } A_j = u:s \\ R_e \vdash \sigma(A_j) & \text{otherwise.} \end{cases}$$

Note now that, for $1 \leq j \leq q_i$,

- If $A_j = (u:s)$, then $[A_j]_{\sigma}^{\sharp}$ holds by induction hypothesis, since $R_e[\beta] \vdash \sigma(u) : s$.
- If $A_j \neq (u:s)$, then $[A_j]_{\sigma}^{\sharp}$ holds by assumption.

Hence, $[A_0]_{\sigma}^{\sharp}$, that is, $\text{bexp}(R_1[\beta] \vdash \phi_1(\sigma(w)), \dots, R_p[\beta] \vdash \phi_p(\sigma(w)))$, also holds. Finally, since $R_e[\beta] \vdash t = \sigma(w)$ and \mathcal{P} is coherent modulo $R_e[P]$, we have that $\text{bexp}(R_1[\beta] \vdash \phi_1(t), \dots, R_p[\beta] \vdash \phi_p(t))$ as required. \square

We will be mainly interested in those γ such that $U_{\text{MEL}} \models \overline{\beta}(\gamma)$ is equivalent to imposing some restrictions on the instances β at the object level. This will be illustrated in Section 6.

5 Reflected Parameterized Induction

In this section we explain how the inductive principle for reasoning about parameterized theories introduced in Section 4 can be reflected. To accomplish this, the key ideas are the following.

- Parameterization is reflected as quantification over (meta)variables representing the parameters. In particular, parameterized atomic formulae are represented as atomic formulae which contain free (meta)variables representing the parameters.
- Instantiation requirements are reflected as a formula (γ), which contains also free (meta)variables representing the parameters. The idea is that all substitutions of the (meta)variables representing the parameters must satisfy this formula.

5.1 Representing Parameterized Theories

We first need to further extend the notation introduced in Section 2.2 to deal with parameters. Let $P = (\Omega \cup V \cup Z, E \cup Mb(V))$ be a parameter theory with $\Omega = (K, \Sigma, S)$. For all terms $t \in T_{\Omega \cup V}(X)$, we will denote by $\overline{t}^{[V, X]}$ its reflective $(\overline{\quad})$ -representation except that now parameters $v \in V$ and variables $x \in X$ are replaced by (meta)variables v and x of the kind **[Term]**. For t a ground term, we shall simply write $\overline{t}^{[V]}$. Similarly, if $t \in T_{\Omega \cup Z}(X)$ we shall write $\overline{t}^{[X]}$ as we did in Section 2.2. Also, for any sort z in Z_k , $k \in K$, we will denote by $\overline{z}^{[Z]}$ a (meta)variable of the kind **[AxiomSet]**. In addition, we will denote by $\overline{V}^{[V]}$ the

set $\overline{V}^{[V]} \triangleq \{\overline{v}^{[v]} \mid v \in V\}$, and by $\overline{Z}^{[Z]}$ the set $\overline{Z}^{[Z]} \triangleq \{\overline{z}^{[z]} \mid z \in Z_k, k \in K\}$, and assume that they are disjoint.

Finally, for any theory morphism $\beta : P \longrightarrow Q$ in \mathcal{V}_P , with $Q = (\Omega \cup V \cup Z, E \cup Eq(V) \cup Ax(Z))$, we will denote by $\overline{\beta}$ the ground substitution $\overline{\beta} : \overline{V}^{[V]} \cup \overline{Z}^{[Z]} \longrightarrow T_{\text{MEL}}$, defined as follows: $\overline{\beta}(\overline{v}^{[v]}) \triangleq \overline{t}$, if $(v = t) \in Eq(V)$, and $\overline{\beta}(\overline{z}^{[z]}) \triangleq \overline{Ax(z)}$, if $z \in Z_k$.

Proposition 7. *For all theory morphisms $\beta : P \longrightarrow Q$ in \mathcal{V}_P and all terms $t \in T_{\Omega \cup V}(X)$,*

$$\overline{\beta}(\overline{t}^{[v,x]}) = \overline{t}_\beta^{[x]}.$$

Proof. By structural induction on t . □

We now define a *generic* representation function $\overline{(_)}^P$ for parameterized membership equational theories. Let $P = (\Omega \cup V \cup Z, E \cup Mb(V))$ be a parameter theory with $\Omega = (K, \Sigma, S)$, $K = \{k_1, \dots, k_m\}$ and $Mb(V) = \{v_1 : s_1, \dots, v_n : s_n\}$. Then, for any parameterized theory $T[P] = (\Omega' \cup V \cup Z, E \cup G \cup Mb(V))$ in \mathcal{P}_P ,

$$\overline{T[P]}^P \triangleq (\overline{\Omega' \cup V \cup Z}, \overline{E \ G \ Mb(V)}^P \ \overline{Z}^P),$$

where

– $\overline{Mb(V)}^P$ is the term

$$\overline{Mb(V)}^P \triangleq (\text{eq } \overline{v_1} = \overline{v_1}^{[v_1]} \dots \text{eq } \overline{v_n} = \overline{v_n}^{[v_n]} .),$$

and

– \overline{Z}^P is the term

$$\overline{Z}^P \triangleq (\overline{Z_{k_1}}^P \dots \overline{Z_{k_m}}^P)$$

where, for any $k_i \in K$, if $Z_{k_i} = \{z_{i1}, \dots, z_{iq_{k_i}}\}$, then $\overline{Z_{k_i}}^P$ is the term

$$\overline{Z_{k_i}}^P \triangleq (\overline{z_{i1}}^{[z]} \dots \overline{z_{iq_{k_i}}}^{[z]}).$$

Intuitively, \overline{Z}^P is a term representing all possible instantiations of the set of axioms defining the sorts in Z .

Proposition 8. *For any parameterized membership equational theory $T[P] \in \mathcal{P}_P$, $T[P] = (\Omega' \cup V \cup Z, E \cup G \cup Mb(V))$, and any theory morphism $\beta : P \longrightarrow Q$ in \mathcal{V}_P , it holds that*

$$\overline{\beta}(\overline{T[P]}^P) = \overline{T[\beta]}.$$

Proof. By definition of substitution application and $\overline{\beta}$ we have

$$\begin{aligned} \overline{\beta}(\overline{T[P]}^P) &= (\overline{\Omega' \cup V \cup Z}, \overline{E \ G \ \beta(Mb(V))}^P \ \overline{\beta(Z}^P)) \\ &= (\overline{\Omega' \cup V \cup Z}, \overline{E \ G} \ (\text{eq } \overline{v_1} = \overline{t_1} \dots \text{eq } \overline{v_n} = \overline{t_n} .) \\ &\quad (\overline{Ax(z_{11})} \dots \overline{Ax(z_{mq_{k_m}})})) \end{aligned}$$

which, by the definition of $T[\beta]$, yields the desired result. □

5.2 Representing Parameterized Atomic Formulae

We now define a *generic* representation function $\overline{(_)}^{[-]}$ for atomic formulae over parameterized membership equational theories. Note that we use the same notation as in Section 2.2.

For $P = (\Omega \cup V \cup Z, E \cup Mb(V))$ with $\Omega = (K, \Sigma, S)$, any parameterized theory $T[P] \in \mathcal{P}_P$, and any membership assertion $t:s$,

$$\overline{t:s}^{[T[P],X]} \triangleq (\overline{t}^{[V,X]} : \overline{s} \text{ in } \overline{T[P]}^P) = \mathbf{true}.$$

Similarly, for any equation $t = t'$,

$$\overline{t = t'}^{[T[P],X]} \triangleq (\overline{t}^{[V,X]} = \overline{t'}^{[V,X]} \text{ in } \overline{T[P]}^P) = \mathbf{true}.$$

Proposition 9. *For all ground atomic formulae ϕ over the signature of the parameterized theory $T[P]$ and all theory morphisms $\beta : P \rightarrow Q$ in \mathcal{V}_P ,*

$$U_{\text{MEL}} \models \overline{\beta}(\overline{\phi}^{[T[P],\emptyset]}) \iff T[\beta] \vdash \phi_\beta.$$

Proof. Let $\phi = t:s$ (the proof is analogous for $\phi = (t = t')$). Notice that by the definition of substitution application and Propositions 7 and 8,

$$\begin{aligned} \overline{\beta}(\overline{t:s}^{[T[P],\emptyset]}) &= (\overline{\beta}(\overline{t}^{[V]}) : \overline{s} \text{ in } \overline{T[P]}^P) = \mathbf{true}) \\ &= (\overline{\beta}(\overline{t}^{[V]}) : \overline{s} \text{ in } \overline{\beta}(\overline{T[P]}^P) = \mathbf{true}) \\ &= (\overline{t}_\beta : \overline{s} \text{ in } \overline{T[\beta]} = \mathbf{true}). \end{aligned}$$

Thus, since $(\overline{t}_\beta : \overline{s} \text{ in } \overline{T[\beta]} = \mathbf{true})$ is a ground atomic formula, due to the soundness and completeness of membership equational logic we can reduce the problem to proving that

$$U_{\text{MEL}} \vdash (\overline{t}_\beta : \overline{s} \text{ in } \overline{T[\beta]}) = \mathbf{true} \iff T[\beta] \vdash \phi_\beta,$$

which holds by Proposition 2. □

Corollary 1. *For P a parameter theory with $Mb(V) = \{v_1:s_1, \dots, v_n:s_n\}$, and $\beta : P \rightarrow Q$ in \mathcal{V}_P ,*

$$U_{\text{MEL}} \models \overline{\beta}(\overline{v_i:s_i}^{[P,\emptyset]}) \quad (1 \leq i \leq n).$$

Proof. Notice that in this case the parameterized theory $T[P]$ is $P[P] = P$, and hence $T[\beta]$ is Q . Then, by Proposition 9,

$$U_{\text{MEL}} \models \overline{\beta}(\overline{v_i:s_i}^{[P,\emptyset]}) \iff Q \vdash (v_i:s_i)_\beta, \quad (1 \leq i \leq n)$$

and the righthand side entailments hold by definition of Q . □

5.3 Representing Requirements

We will need to impose, at the metalevel, that the parameters in the theory P are correctly instantiated. For that, if $Mb(V) = \{v_1 : s_1, \dots, v_n : s_n\}$, we define

$$\overline{Mb(V)}^{c(P)} \triangleq ((\overline{v_1}^{[V]} :: \overline{k_1} \text{ in } \overline{P}) = \text{true} \wedge \dots \wedge (\overline{v_n}^{[V]} :: \overline{k_n} \text{ in } \overline{P}) = \text{true} \wedge \overline{v_1 : s_1}^{[P, \emptyset]} \wedge \dots \wedge \overline{v_n : s_n}^{[P, \emptyset]}),$$

where k_i is the kind of s_i for $i = 1, \dots, n$. It immediately follows from Proposition 1 and Corollary 1 that

Proposition 10. *For $P = (\Omega \cup V \cup Z, E \cup Mb(V))$ a parameter theory, and $\beta : P \rightarrow Q$ in \mathcal{V}_P ,*

$$U_{\text{MEL}} \models \overline{\beta}(\overline{Mb(V)}^{c(P)}).$$

The formula $\overline{Mb(V)}^{c(P)}$ will be used to impose that the parameters in V are instantiated with ground terms of the appropriate sort. Analogously, we will also require that the variables in $\overline{Z}^{[Z]}$ are correctly instantiated (that is to say, with membership axioms specifying the sorts in Z), and for that we will use a new representation function $\overline{(_)}^{D(P)}$, defined over sorts in Z as follows:

$$\overline{z}^{D(P)} \triangleq (\overline{z}^{[Z]} \text{ spec } \overline{z} \text{ in } \overline{\Omega \cup Z}) = \text{true} \quad (z \in Z_k).$$

Proposition 11. *For $P = (\Omega \cup V \cup Z, E \cup Mb(V))$, any sort $z \in Z_k$ for some kind k , and $\beta : P \rightarrow Q$ in \mathcal{V}_P ,*

$$U_{\text{MEL}} \models \overline{\beta}(\overline{z}^{D(P)}).$$

Proof. By definition of substitution application and $\overline{\beta}$,

$$\begin{aligned} \overline{\beta}(\overline{z}^{D(P)}) &= (\overline{\beta}(\overline{z}^{[Z]}) \text{ spec } \overline{z} \text{ in } \overline{\Omega \cup Z} = \text{true}) \\ &= (\overline{Ax}(z) \text{ spec } \overline{z} \text{ in } \overline{\Omega \cup Z} = \text{true}), \end{aligned}$$

and hence the result follows from Proposition 3 by soundness of membership equational logic. \square

The representation function $\overline{(_)}^{D(P)}$ is extended to Z in the obvious way by

$$\overline{Z}^{D(P)} \triangleq \bigwedge_{z \in Z_k} \overline{z}^{D(P)}.$$

Corollary 2. *For $P = (\Omega \cup V \cup Z, E \cup Mb(V))$ a parameter theory and $\beta : P \rightarrow Q$ in \mathcal{V}_P ,*

$$U_{\text{MEL}} \models \overline{\beta}(\overline{Z}^{D(P)}).$$

5.4 Reflecting Parameterized Induction Principles

We now define a representation function for metalogical statements. Let $\mathcal{P} = \{R_1[P], \dots, R_p[P]\}$ be a finite multiset of parameterized theories in \mathcal{P}_P that is coherent modulo $R_e[P]$, $\{k_1, \dots, k_n\}$ a finite multiset of kinds, and τ a metalogical statement of the form

$$\begin{aligned} \forall \beta \in \mathcal{V}_P. (U_{\text{MEL}} \models \overline{\beta}(\gamma) \implies \\ \forall t_1 \in (T_{\Omega_1})_{k_1} \dots \forall t_n \in (T_{\Omega_n})_{k_n}. \text{bexp}(R_1[\beta] \vdash (\phi_1(\mathbf{t}))_\beta, \dots, R_p[\beta] \vdash (\phi_p(\mathbf{t}))_\beta)) \end{aligned}$$

where each $\phi_i(\mathbf{x})$ is an atomic formula with free variables in \mathbf{x} , each x_i of kind k_i . Then, $\overline{\tau}$ is defined as

$$\begin{aligned} \forall \overline{V}^{[V]}. \forall \overline{Z}^{[Z]}. ((\overline{Mb(V)})^{c(P)} \wedge \overline{Z}^{D(P)} \wedge \gamma) \implies \\ \forall x_1 \dots \forall x_n. ((x_1 :: \overline{k_1} \text{ in } \overline{R_1[P]}) = \mathbf{true} \wedge \dots \wedge (x_n :: \overline{k_n} \text{ in } \overline{R_n[P]}) = \mathbf{true}) \\ \implies \text{bexp}(\overline{\phi_1(\mathbf{x})}^{[R_1[P], \mathbf{x}]}, \dots, \overline{\phi_p(\mathbf{x})}^{[R_p[P], \mathbf{x}]}) \end{aligned}$$

where $\{x_1, \dots, x_n\}$ are now variables of the kind $[\text{Term}]$.

The following auxiliary result is needed in the proof of our main theorem.

Proposition 12. *For $P = (\Omega \cup V \cup Z, E \cup Mb(V))$ a parameter theory with $Mb(V) = \{v_1 : s_1, \dots, v_n : s_n\}$, and any ground substitution $h : \overline{V}^{[V]} \cup \overline{Z}^{[Z]} \longrightarrow TU_{\text{MEL}}$ such that*

$$U_{\text{MEL}} \models h(\overline{Mb(V)})^{c(P)} \wedge \overline{Z}^{D(P)},$$

there is a theory morphism $\beta : P \longrightarrow Q$ in \mathcal{V}_P , with $Q = (\Omega \cup V \cup Z, E \cup Eq(V) \cup Ax(Z))$, such that β is the ground substitution h .

Proof. By definition of substitution application, for $1 \leq i \leq n$,

$$h(\overline{v_i}^{[V]} :: \overline{k_i} \text{ in } \overline{P} = \mathbf{true}) = (h(\overline{v_i}^{[V]}) :: \overline{k_i} \text{ in } \overline{P} = \mathbf{true}).$$

The hypothesis implies $U_{\text{MEL}} \models h(\overline{v_i}^{[V]} :: \overline{k_i} \text{ in } \overline{P} = \mathbf{true})$ for each v_i , $1 \leq i \leq n$, and by Proposition 1, using the completeness of membership equational logic and the fact that $h(\overline{v_i}^{[V]} :: \overline{k_i} \text{ in } \overline{P} = \mathbf{true})$ is a ground atomic formula, it follows that there are ground terms $t_i \in (T_\Omega)_{k_i}$ such that $\overline{t_i} = h(\overline{v_i}^{[V]})$.

Similarly, from $U_{\text{MEL}} \models h(\overline{Z}^{D(P)})$, by Proposition 4, it follows that there are sets of axioms $Ax(z)$ specifying z in $\overline{\Omega \cup Z}$ for each $z \in Z_k$, $k \in K$, such that $h(\overline{z}^{[Z]}) = Ax(z)$.

Let $Q = (\Omega \cup V \cup Z, E \cup \{v_1 = t_1, \dots, v_n = t_n\} \cup \bigcup_{z \in Z_k} Ax(z))$. By definition of substitution application,

$$\begin{aligned} h(\overline{v_i : s_i}^{[P, \emptyset]}) \\ = (h(\overline{v_i}^{[V, \emptyset]} : \overline{s_i} \text{ in } \overline{P}^P) = \mathbf{true}) \\ = (h(\overline{v_i}^{[V]} : \overline{s_i} \text{ in } (\overline{\Omega \cup V \cup Z}, \overline{E} \text{ eq } \overline{v_1} = h(\overline{v_1}^{[V]}) \dots \text{eq } \overline{v_n} = h(\overline{v_n}^{[V]}) \\ h(\overline{z_{11}}^{[Z]}) \dots h(\overline{z_{mq_{k_m}}^{[Z]})}) = \mathbf{true}) \\ = (\overline{t_i} : \overline{s_i} \text{ in } \overline{Q} = \mathbf{true}). \end{aligned}$$

Since $h(\overline{v_i : s_i}^{[P, \emptyset]})$ is an atomic ground formula and $U_{\text{MEL}} \simeq h(\overline{v_i : s_i}^{[P, \emptyset]})$, by Proposition 2 and completeness of membership equational logic we have $Q \vdash t_i : s_i$, $1 \leq i \leq n$. Then, the identity signature morphism $\beta : P \rightarrow Q$ satisfies the requirements to be in \mathcal{V}_P . \square

Theorem 2. *Let τ be a metalogical statement of the above form. Then, τ holds iff $U_{\text{MEL}} \simeq \overline{\tau}$.*

Proof. Assume that τ holds and let $h : \overline{V}^{[V]} \cup \overline{Z}^{[Z]} \rightarrow T_{U_{\text{MEL}}}$ be such that $U_{\text{MEL}} \simeq h(\overline{Mb(V)}^{C(P)} \wedge \overline{Z}^{D(P)} \wedge \gamma)$. By Proposition 12 there is $\beta \in \mathcal{V}_P$ with $\overline{\beta} = h$, so our task is reduced to proving that

$$\begin{aligned} \forall x_1 \dots \forall x_n. ((x_1 :: \overline{k_1} \text{ in } \overline{R_1[P]}) = \text{true} \wedge \dots \wedge (x_n :: \overline{k_n} \text{ in } \overline{R_n[P]}) = \text{true}) \\ \implies \overline{\beta}(\overline{\text{bexp}(\phi_1(\mathbf{x}))}^{[R_1[P], \mathfrak{a}]}, \dots, \overline{\phi_p(\mathbf{x})}^{[R_p[P], \mathfrak{a}]}) \end{aligned}$$

holds in the initial model of U_{MEL} . So let $\sigma : \{x_1, \dots, x_n\} \rightarrow T_{U_{\text{MEL}}}$ be a substitution such that

$$(\sigma(x_1) :: \overline{k_1} \text{ in } \overline{R_1[P]}) = \text{true} \wedge \dots \wedge (\sigma(x_n) :: \overline{k_n} \text{ in } \overline{R_n[P]}) = \text{true}$$

holds in $T_{U_{\text{MEL}}}$; by Proposition 1 we know that, for $i = 1, \dots, n$, $\sigma(x_i) = \overline{w_i}$ for some $w_i \in (T_{\Omega_i})_{k_i}$. By the definition of substitution application and Propositions 7 and 8, for $1 \leq l \leq p$ and $\phi_l = (t_l : s_l)$ (similarly for $\phi_l = (t_l = t'_l)$),

$$\begin{aligned} \sigma(\overline{\beta}(\overline{\phi_l(\mathbf{x})}^{[R_l[P], \mathfrak{a}]}) &= \sigma(\overline{\beta}(t_l(\mathbf{x}) : s_l}^{[R_l[P], \mathfrak{a}]}) \\ &= \sigma(\overline{\beta}(t_l(\mathbf{x})}^{[V, \mathfrak{a}]} : \overline{s_l} \text{ in } \overline{R_l[P]}^P = \text{true})) \\ &= (\sigma(\overline{\beta}(t_l(\mathbf{x})}^{[V, \mathfrak{a}]})) : \overline{s_l} \text{ in } \overline{\beta}(R_l[P]^P) = \text{true}) \\ &= (\sigma(\overline{(t_l(\mathbf{x}))}_\beta}^{[\mathfrak{a}]})) : \overline{s_l} \text{ in } \overline{R_l[\beta]} = \text{true}) \\ &= (\overline{(t_l(\mathbf{x}))}_\beta) : \overline{s_l} \text{ in } \overline{R_l[\beta]} = \text{true}) \\ &= (\overline{(t_l(\sigma(\mathbf{x})))}_\beta) : \overline{s_l} \text{ in } \overline{R_l[\beta]} = \text{true}) \\ &= (\overline{\beta}(t_l(\mathbf{w}))}^{[V]}) : \overline{s_l} \text{ in } \overline{\beta}(R_l[P]^P) = \text{true}) \\ &= \overline{\beta}(t_l(\mathbf{w}) : s_l}^{[R_l[P], \emptyset]}) \\ &= \overline{\beta}(\overline{\phi_l(\mathbf{w})}^{[R_l[P], \emptyset]}). \end{aligned}$$

Hence, by Proposition 9, $U_{\text{MEL}} \simeq \sigma(\overline{\beta}(\overline{\phi_l(\mathbf{x})}^{[R_l[P], \mathfrak{a}]}))$ iff $R_l[\beta] \vdash \phi_l(\mathbf{w})_\beta$. But then, since τ holds and we are assuming that $U_{\text{MEL}} \simeq \overline{\beta}(\gamma)$, we have $\text{bexp}(R_1[\beta] \vdash (\phi_1(\mathbf{t}))_\beta, \dots, R_p[\beta] \vdash (\phi_p(\mathbf{t}))_\beta)$ for all \mathbf{t} , in particular for \mathbf{w} , and the result follows.

We have just shown the implication from left to right. A careful examination reveals that all the implications are in fact equivalences and hence this proves the theorem. \square

In particular, Theorem 2 can be applied to the inductive principle

$$\begin{aligned} & \forall \beta \in \mathcal{V}_P. (U_{\text{MEL}} \vDash \bar{\beta}(\gamma) \implies \psi_1) \wedge \cdots \wedge \forall \beta \in \mathcal{V}_P. (U_{\text{MEL}} \vDash \bar{\beta}(\gamma) \implies \psi_n) \\ & \implies \\ & \forall \beta \in \mathcal{V}_P. (U_{\text{MEL}} \vDash \bar{\beta}(\gamma) \implies \\ & \quad \forall t \in T_\Omega. (R_0[\beta] \vdash t : s \implies \text{bexp}(R_1[\beta] \vdash \phi_1(t)_\beta, \dots, R_p[\beta] \vdash \phi_p(t)_\beta))) \end{aligned}$$

by replacing each metalogical statement ϕ by its logical representation $\bar{\phi}$ to get an inductive principle for U_{MEL} .

6 The Deduction Theorem Revisited

6.1 Formalizing the Deduction Theorem

The parameterized versions of the deduction theorem can now be expressed as metatheoretic statements relating the initial models of all the different instantiations of $\text{DT}[\text{EML}]$ and EML that satisfy certain requirements. In its standard form, the deduction theorem can be formalized as follows:

$$\forall \beta \in \mathcal{V}_{\text{EML}}^1. \forall t \in T_{\text{EML}}. (\text{DT}[\beta] \vdash t : \text{Theorem} \implies \text{EML}[\beta] \vdash [-\rightarrow, @A, t]_\beta : \text{Theorem}), \quad (4)$$

where

$$\mathcal{V}_{\text{EML}}^1 = \{\beta \in \mathcal{V}_{\text{EML}} \mid Ax(@\text{NewAxiom}) \cup Ax(@\text{NewSynRule}) \cup Ax(@\text{NewInfRule}) = \emptyset\}.$$

Note that $\{\text{DT}[\text{EML}], \text{EML}[\text{EML}]\}$ is coherent module $\text{DT}[\text{EML}]$.

However, the deduction theorem also holds for all extensions of minimal logic's language and minimal logic's axioms, which can be formalized as follows:

$$\forall \beta \in \mathcal{V}_{\text{EML}}^2. \forall t \in T_{\text{EML}}. (\text{DT}[\beta] \vdash t : \text{Theorem} \implies \text{EML}[\beta] \vdash [-\rightarrow, @A, t]_\beta : \text{Theorem}), \quad (5)$$

where $\mathcal{V}_{\text{EML}}^2 = \{\beta \in \mathcal{V}_{\text{EML}} \mid Ax(@\text{NewInfRule}) = \emptyset\}$.

Furthermore, the deduction theorem can also be verified for all extensions of minimal logic's language, axioms, and two-premise rules (this can be generalized to finitely many assumptions), provided that all new rules of the form

$$\frac{B \quad C}{D}$$

are such that, for all formulae A , if $(A \rightarrow B)$ and $(A \rightarrow C)$ are theorems in the corresponding extension of minimal logic, then $(A \rightarrow D)$ is also a theorem [2]. This version of the deduction theorem can be formalized as follows:

$$\forall \beta \in \mathcal{V}_{\text{EML}}^3. \forall t \in T_{\text{EML}}. (\text{DT}[\beta] \vdash t : \text{Theorem} \implies \text{EML}[\beta] \vdash [-\rightarrow, @A, t]_\beta : \text{Theorem}), \quad (6)$$

where

$$\begin{aligned} \mathcal{V}_{\text{EML}}^3 = \{ & \beta \in \mathcal{V}_{\text{EML}} \mid \forall x_1. \forall x_2. \forall x_3. \forall x_4. (\text{DT}[\beta] \vdash [x_4, x_2, x_3] : @\text{NewInfRule} \\ & \implies \text{EML}[\beta] \vdash [-\rightarrow, x_1, x_2] : \text{Theorem} \implies \text{EML}[\beta] \vdash [-\rightarrow, x_1, x_3] : \text{Theorem} \\ & \implies \text{EML}[\beta] \vdash [-\rightarrow, x_1, x_4] : \text{Theorem})\}. \end{aligned}$$

6.2 Proving the Deduction Theorem

In what follows, we denote by $V\text{-EM}$ and $Z\text{-EM}$, respectively, the sets of parameters $\{\textcircled{A}\}$ and $\{\textcircled{\text{NewAxiom}}, \textcircled{\text{NewSynRule}}, \textcircled{\text{NewInfRule}}\}$, and by MB-V-EM the set of axioms $\{\text{mb } \textcircled{A} : \text{Formula } .\}$. Using the results of Section 5 we can formalize the different versions of the deduction theorem, (4), (5), and (6), as theorems about U_{MEL} . All these theorems have a common structure

$$\begin{aligned} & \forall \overline{V\text{-EM}}^{[V\text{-EM}]} . \forall \overline{Z\text{-EM}}^{[Z\text{-EM}]} . ((\overline{\text{MB-V-EM}}^{C(\text{EML})} \wedge \overline{Z\text{-EM}}^{D(\text{EML})} \wedge \gamma) \implies \\ & \quad \forall x. ((x :: \overline{\text{Expression in EML}} = \text{true}) \implies \\ & \quad \quad \overline{(x : \text{Theorem}}^{[\text{DT}[\text{EML}], x]} \implies \overline{[->, \textcircled{A}, x] : \text{Theorem}}^{[\text{EML}, x]}))), \end{aligned} \quad (7)$$

but differ in the definition of γ . Note that this is in direct correspondence with the fact that the metatheoretic statements (4), (5), and (6) only differ in the requirements imposed over the instantiations $\beta \in \mathcal{V}_{\text{EML}}$. Concretely, for (4), (5), and (6) the formula γ is defined, respectively, as:

$$\begin{aligned} \gamma^1 & \triangleq (\overline{\textcircled{\text{NewAxiom}}}^{[Z\text{-EM}]} = \text{none} \wedge \overline{\textcircled{\text{NewSynRule}}}^{[Z\text{-EM}]} = \text{none} \wedge \\ & \quad \overline{\textcircled{\text{NewInfRule}}}^{[Z\text{-EM}]} = \text{none}), \\ \gamma^2 & \triangleq (\overline{\textcircled{\text{NewInfRule}}}^{[Z\text{-EM}]} = \text{none}), \\ \gamma^3 & \triangleq (\forall x_1. \forall x_2. \forall x_3. \forall x_4. \overline{[x_4, x_2, x_3] : \textcircled{\text{NewInfRule}}}^{[\text{DT}[\text{EML}], \varpi]} \\ & \quad \implies \overline{[->, x_1, x_2] : \text{Theorem}}^{[\text{EML}, \varpi]} \implies \overline{[->, x_1, x_3] : \text{Theorem}}^{[\text{EML}, \varpi]} \\ & \quad \implies \overline{[->, x_1, x_4] : \text{Theorem}}^{[\text{EML}, \varpi]}). \end{aligned}$$

By Theorem 2, (7) implies, for each definition of γ , the corresponding parameterized version of the deduction theorem. The correctness of the above formalizations follows from the following remark: for all theory morphisms $\beta \in \mathcal{V}_{\text{EML}}$,

$$\beta \in \mathcal{V}_{\text{EML}}^i \iff U_{\text{MEL}} \models \overline{\beta}(\gamma^i) \quad i = 1, 2, 3.$$

Finally, to prove each version of (7) in U_{MEL} we apply the reflected version of the induction principle for the sort **Theorem** in the parameterized theory $\text{DT}[\text{EML}]$. The proofs mirror the standard proof of the deduction theorem: we show $A \rightarrow B$ by induction on the structure of possible derivations of B when A is assumed as an axiom. Note, however, that to prove the deduction theorem for all extensions of minimal logic's language and minimal logic's axioms we have to consider as an additional base case of the inductive proof when B is one of the new axioms. Moreover, to prove the deduction theorem for all extensions of minimal logic's language, axioms, and two-premise rules satisfying the above mentioned requirement, we also have to consider as an additional step case of the inductive proof when B follows by an application of one of the new rules. By using the reflected version of the induction principle for the sort **Theorem** in the parameterized theory $\text{DT}[\text{EML}]$, all these considerations are appropriately mirrored in our proofs.

7 Conclusion

Based on the ideas introduced in [1] by Basin, Clavel, and Meseguer about reflective metalogical frameworks, and about membership equational logic as one of them, we have further explored the capabilities of membership equational logic as a logic to reason *about* logics and about relationships *between* logics.

In this paper we have extended the notion of parameterized membership equational theories and of reflected parameterized induction introduced in [1]. By doing this, we are able to formalize and prove a wider class of metatheorems: for example, the parameterized versions (5) and (6) of the deduction theorem cannot be formalized in [1,4]. Our experiments show that one can prove metatheorems similar to those provable in logical frameworks based on parameterized inductive definitions [2]. In essence, we can do this because the requirements that such metatheorems pose on the metatheory—namely, that one can build families of sets using parameterized inductive definitions and that one can reason about their elements by induction—are realizable in membership equational logic using parameterization and reflection.

This work can be extended in a number of directions, both theoretical and practical. From the theoretical side, a research line would be to investigate how to reflect induction principles other than structural induction, e.g., induction over an arbitrary, user-definable well-founded order; also, our notion of parameterized membership equational theories and of their instantiations could be further generalized. From the practical side, the obvious application would be to extend the ITP theorem prover [7] with reflected parameterized induction principles so as to carry out inductive proofs of metatheorems; however, the development of the tool has changed hands and gone undercover, so it is not clear how it will evolve.

Acknowledgments. We thank David Basin and José Meseguer for many discussions on using reflection for metareasoning.

References

1. Basin, D., Clavel, M., Meseguer, J.: Reflective metalogical frameworks. *ACM Transactions on Computational Logic* 5(3), 528–576 (2004)
2. Basin, D., Matthews, S.: Structuring metatheory on inductive definitions. *Information and Computation* 162(1/2), 80–95 (2000)
3. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. *Theoretical Computer Science* 236, 35–132 (2000)
4. Clavel, M., Martí-Oliet, N., Palomino, M.: Formalizing and proving semantic relations between specifications by reflection. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) *AMAST 2004*. LNCS, vol. 3116, pp. 72–86. Springer, Heidelberg (2004)
5. Clavel, M., Meseguer, J.: Axiomatizing reflective logics and languages. In: Kiczales, G. (ed.) *Proceedings of Reflection 1996*, San Francisco, California, pp. 263–288 (April 1996)
6. Clavel, M., Meseguer, J., Palomino, M.: Reflection in membership equational logic, many-sorted equational logic, Horn-logic with equality, and rewriting logic. *Theoretical Computer Science* 373(1-2), 70–91 (2007)

7. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: A tutorial. *Journal of Universal Computer Science* 12(11), 1618–1650 (2006); Special issue with extended versions of selected papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages
8. Feferman, S.: Finitary inductively presented logics. In: Ferro, R., Bonotto, C., Valentini, S., Zanardo, A. (eds.) *Logic Colloquium 1988*, pp. 191–220. North-Holland (1989)
9. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery* 39(1), 95–146 (1992)
10. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) *WADT 1997*. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)

Fast Sort Computations for Order-Sorted Matching and Unification

Steven Eker

Computer Science Laboratory, SRI International
Menlo Park, CA 94025, USA
eker@csl.sri.com

Abstract. Given a preregular order-sorted signature, we consider two closely related problems. The first arises in matching where we need to compute the least sort of a ground term in order to decide whether it is less or equal to the sort of a variable to which we wish to bind it. The second arises in unification where we have computed an unsorted unifier and we want to compute any corresponding order-sorted unifiers by finding order-sorted renamings of the unsorted free variables occurring in the unifier such that for each bound variable, the least sort of the term to which it is bound becomes less than or equal to its own sort.

We present a fast solution to the first problem, based on compiling the overloaded declarations for each operation in to a decision diagram. We then show how this method can be lifted to the variable case using a BDD encoding to represent computations with unknown sorts in order to solve the second problem. We also discuss some extensions of the techniques.

1 Introduction

Order-sorted term algebra is a useful extension of classical unsorted term algebra and many sorted term algebra. It formed the semantic underpinnings of the OBJ3 language [10], while the membership equational logic fragment of the Maude language [7] is implemented as an extension of order-sorted term rewriting.

Two of the most fundamental computations that are performed with terms are matching, which is a key step in term rewriting, and unification which a key step in completion and narrowing [8]. In this paper we describe the algorithms used for the sort computation part of order-sorted matching and unification in the Maude 2 interpreter. Concrete examples will be presented in syntax of Maude.

1.1 Matching

With matching we have a distinct pattern term and a distinct subject term where the subject term is not allowed to contain variables. Order-sorted matching can be done in a manner similar to unsorted matching, except for the binding of variables. When a variable x of sort s is to be bound to a ground term t , the least sort, s' , of t must be calculated and compared to the sort s . If $s' \leq s$ then

the variable can be bound and the matching process proceeds as normal. If not, this branch of the matching process fails.

Thus for matching we essentially need two pieces of functionality: computing the sort of a ground term and comparing the two sorts. The former can be done bottom-up, so at each subterm the task becomes to compute the least sort of $f(t_1, \dots, t_n)$ given the least sorts of t_1, \dots, t_n . The latter can be done using any convenient representation of the partial order on sorts. For example we can index the sorts and for each sort s keep an array of Booleans encoding those sorts that are less or equal to s . Then checking whether some sort s' is less or equal to s can be done with a simple array lookup.

1.2 Unification

Unification is rather more difficult. Although sort information can often be usefully incorporated into an existing unsorted unification algorithm to prune branches that must fail due to sort considerations, the design, and more importantly the combination, of order-sorted unification algorithms appears difficult, as constraints on the sort of a fresh variable can arise in multiple unification subproblems. Little has been written on the subject, where by contrast unsorted unification algorithms and their combination have a rich literature, for example: [12,6,16,4,2,3].

The approach developed in the literature is to use an unsorted unification algorithm (or in the case of unification modulo equational theories, a combination unsorted unification algorithms) to generate a complete set of unifiers, (or possibly a reduced set of unifiers, where some unifiers that could not give rise to an order-sorted unifier are opportunistically culled), where each unifier is expressed in terms of free unsorted variables. Then a second phase takes each unsorted unifier and generates zero or more order-sorted unifiers by finding injective renamings ρ that map the free unsorted variables in the unifier to fresh sorted variables, such that for each bound variable $x \leftarrow t$ in the unsorted unifier, the least sort of $t\rho$ is less or equal to the sort of x . Under certain criteria, Meseguer et al. [13] show that this approach can generate a complete set of order-sorted unifiers. There are problems with certain natural order-sorted signatures that do not satisfy these criteria, however Hendrix and Meseguer [11] show how at least one important case can be handled by translating the unification problem into an extended signature that does satisfy the criteria.

In this paper we ignore problematic special cases, assume that the criteria for completeness and correctness have been met and focus on algorithms for performing the sort computations that are fast in practice.

2 Preliminaries

Our notation closely follows that of Hendrix and Meseguer [11]. Formally our order-sorted computations will take place with respect to an order-sorted signature $\Sigma = \langle S, F, \leq \rangle$ where S is a set of sorts, $F = \{F_{w,s} \mid (w, s) \in S^* \times S\}$ is a

family of sets of operators and \leq is a partial order on S . Here an operator f may appear in several different sets $F_{w,s}$, as long as all the w have the same length, in which case it is said to be *overloaded*.

Let $X = \{X_s \mid s \in S\}$ be a family of disjoint sets of variables. The Σ -terms with variables X form a family $T_\Sigma(X) = \bigcup_{s \in S} T_\Sigma(X)_s$ where $T_\Sigma(X)_s$ is defined inductively: $X_s \cup F_{\lambda,s} \subseteq T_\Sigma(X)_s$ and $f(t_1, \dots, t_n) \in T_\Sigma(X)_s$ if there exists $(s_1 \dots s_n), (s'_1 \dots s'_n)$ and $f \in F_{s_1 \dots s_n, s'}$ such that $s' \leq s$ and for each $i \in \{1, \dots, n\}$, $s'_i \leq s_i$, and $t_i \in T_\Sigma(X)_{s'_i}$. The set of variables occurring in $t \in T_\Sigma(X)$ are denoted $vars(t)$.

Given $t \in T_\Sigma(X)$, the sorts of t form a set $sorts(t) = \{s \mid t \in T_\Sigma(X)_s\}$. If this set has a unique least element l , we say t has a least sort, $ls(t) = l$. For the purposes of this paper we are only interested in signatures Σ where each $t \in T_\Sigma(X)$ has a least sort. Such signatures are called *preregular* [9].

In computing on a partially ordered set of sorts, $\langle S, \leq \rangle$ it is convenient to have a top sort, \top such that $s \leq \top$ for all $s \in S$. We will assume that such a sort exists or has been added.

For $s, s' \in S$, we write $s < s'$ as an abbreviation for $s \leq s'$ and $s \neq s'$. For each natural number $n \geq 2$ we extend the ordering \leq pointwise to S^n ; i.e. $u_1 \dots u_n \leq v_1 \dots v_n$ iff for all $i \in \{1, \dots, n\}, u_i \leq v_i$.

For an operator f , we call the collection of pairs $D_f = \{(w, s) \mid f \in F_{w,s}\}$ the *declarations* of f . For ease of reading we will write a declaration (w, s) using the more suggestive notation $w \rightarrow s$ where w is called the *domain* and s is called the *range*. The declarations of f induce a function

$$ls_f : S^n \rightarrow S$$

where $ls_f(u)$ is the the least sort s such for some $w \in S^n$, with $u \leq w$, $(w \rightarrow s) \in D_f$ or \top if there is no such sort. The prerogularity of the signature ensures if such sorts exists, there is a unique least such sort and thus ls_f is well defined.

It is easy to see that ls_f is monotonic, i.e. if $u \leq v$ then $ls_f(u) \leq ls_f(v)$. It is this monotonicity that allows us to compute the least sort of a term $f(t_1, \dots, t_n)$ bottom-up, by evaluating ls_f on the least sorts of t_1, \dots, t_n .

2.1 Connected Components

The partially ordered set of sorts, $\langle S, \leq \rangle$, can be view as a digraph G with nodes labeled by sorts and an arc from the node labeled by s_1 to the node labeled by s_2 iff $s_2 \leq s_1$. For order-sorted signatures used in programming and specification, it is typically the case that G consists of multiple connected components, representing unrelated hierarchies of sorts. Consider the following example:

```
fmod NUMBERS-AND-SHAPES is
  sorts NzNat Nat NzInt Int .
  subsorts NzNat < Nat NzInt < Int .
  sorts Square Rectangle Rhombus Parallelogram Quadrilateral Triangle Polygon .
  subsorts Square < Rectangle Rhombus < Parallelogram < Quadrilateral .
  subsorts Quadrilateral Triangle < Polygon .
endfm
```

Here there are two connected components of sorts, one representing a hierarchy of integers and one representing a hierarchy of polygons.

If S can be partitioned into multiple, mutually disjoint connected components K_1, \dots, K_m we can gain some efficiency by computing with the partially ordered sets $\langle K_i, \leq \rangle$ rather than $\langle S, \leq \rangle$. In this case we add a top sort \top_K to each connected component K . In Maude these connected components are called *kinds* and the top sorts are called *error sorts*, though in practice an error sort \top_K is often understood as a special sort that represents its kind K in sort computations.

Normally if $f \in F_{u_1 \dots u_n, s_1}$ and $f \in F_{v_1 \dots v_n, s_2}$, we expect that for $i \in \{1, \dots, n\}$, u_i and v_i belong to the same connected component. If this property holds, preregularity implies that s_1 and s_2 are in the same connected component. If this property does not hold then we can treat f as multiple distinct operators that just happen to have the same name. We call this kind of overloaded *ad hoc overloading*.

For the purposes of exposition, we will mostly ignore the partition of S into its connected components, however a practical implementation of the algorithms we present will work with the connected components rather than S and it is the size of these connected components rather than the size of S that is important for measuring time and space requirements.

3 Computing the Least Sort of a Term

As mentioned in the previous section, the least sort of a term $f(t_1, \dots, t_n)$ can be computed by applying ls_f to the least sorts of t_1, \dots, t_n , so the problem reduces to evaluating ls_f on some $u \in S^n$ given the declarations D_f .

A naïve method of computing $ls_f(u)$ is to preprocess D_f by sorting it into a list $(w_1 \rightarrow s_1), \dots, (w_d \rightarrow s_d)$ such that $s_i < s_j$ implies $i < j$. To compute $ls_f(u)$ we find the first i such that $u \leq w_i$ and return s_i or \top if there is no such i . This method requires only modest preprocessing and consumes no additional space over that required to store the declarations of f , however each comparison $u \leq w_i$ requires $O(n)$ evaluations of the partial ordering on sorts, and we may need to examine all d declarations for a running time of $O(n.d)$. In practice this approach is quite slow however it is important to note that both the preprocessing and evaluation phases are polynomial time.

We assume that the signature is preregular, but for a practical implementation it is useful to be able to check that this property holds. For the naïve algorithm we can check the preregularity of ls_f on a single input vector u the following method. If we find a first i such that $u \leq w_i$, we check for all $j \in \{i+1, d\}$ that $u \leq w_j$ implies $s_i \leq s_j$ otherwise we have found a counterexample to the preregularity condition.

A method that is faster at runtime is to use the naïve method to fill out an n -dimensional lookup table. In filling out the table for each operator f we can check the preregularity of ls_f over all input vectors and thus the preregularity of the entire signature Σ . Using a lookup table requires $O(n)$ time for each lookup. However the lookup table for f requires $O(|S|^n)$ storage and $O(n.d \cdot |S|^n)$ initialization time.

This method was in fact used in an early implementation of Maude. However as Maude specifications became more complex, operators with arity greater than

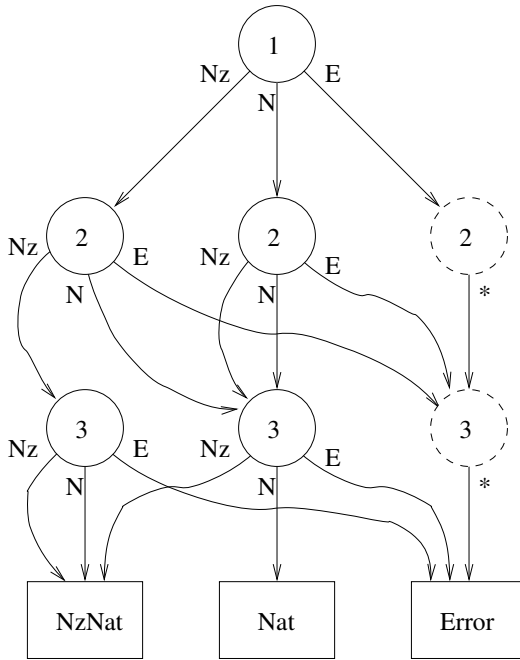


Fig. 1. Decision diagram for ls_{madd}

6, and sort hierarchies with more than 20 sorts became common, and the space and initialization time requirements of this method became impractical.

Given the way ls_f is defined it will have a very regular structure and it is natural to consider representing it as a decision diagram. We illustrate the idea by way of a concrete example. Consider the signature:

```
fmod MULTIPLY-AND-ADD is
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op madd : NzNat NzNat Nat -> NzNat .
  op madd : Nat Nat NzNat -> NzNat .
  op madd : Nat Nat Nat -> Nat .
endfm
```

Here `madd` is the operation that multiplies its first two arguments and adds the result to a third, and the declarations for it capture its behavior with respect to sorts on natural and nonzero natural numbers. We assume a top sort is added, which for convenience we call `Error` and abbreviate the sort names `NzNat`, `Nat` and `Error`, to `Nz`, `N` and `E` where necessary for sake of space. The decision diagram we construct for ls_{madd} is shown in Figure 1. Our decision diagram consists of a directed acyclic graph with root node at the top and the remaining nodes organized into n layers. General decision diagrams label each non-leaf node with a test, each arc with a test result and each leaf node with an outcome [14]. For our purpose, each non-leaf node is labeled with an argument to test, each

arc with a possible value that the argument may have and each leaf node with the value of $ls_{madd}(u)$ where u is obtained by concatenating the arc labels on any path to that leaf node.

Given a decision diagram of this form, the $ls_f(s_1, \dots, s_n)$ is evaluated by starting at the root node and iterating the following step: from a node with label j , follow the outgoing arc labeled by the sort s_j . When a leaf is reached the label of the leaf is the value of $ls_f(s_1, \dots, s_n)$.

Though in principle there is no need to consider the arguments in left to right order, or even in the same order along each path from root to leaf, in practice, it simplifies the evaluation of ls_f . Notice that in Figure 1, two nodes are shown in dashed outline with all of their notional outgoing arcs represented by a single arc labeled “*”. Strictly speaking these nodes are redundant since the argument they test will have no effect on the value of ls_{madd} . However including them makes for a uniform decision diagram where each path from root to leaf is exactly n arcs, which again make the evaluation simpler at runtime. These two decisions trade potentially larger decision diagram size against the speed of the evaluation code in an interpreter, where the same executable code handles sort computations for different operators and the decision diagram is considered data. For the purposes of compiling order-sorted term rewriting systems to executable code, dedicated code to evaluate ls_f for each operator f might be generated from the decision diagram for ls_f , and then minimizing the most likely path length becomes important.

3.1 Efficient Sort Decision Diagram Construction

We will only consider the construction of uniform decision diagrams when the arguments of ls_f are considered left to right. Removing redundant nodes is a trivial transformation, while the effect of alternative traversal sequences for arguments is a more subtle issue which we do not consider here.

In constructing our decision diagrams we would like to check the preregularity of each n -ary operator on all $|S|^n$ possible input vectors however we want to avoid explicitly considering those vectors. The key idea is that each node N is associated with the set of non-redundant declarations that is “still alive” along all paths to N . This bounds the number of nodes in each layer by $2^{|D_f|}$. For Maude programs composed in a structured way, the size of the S (or in practice the size of the connected components) tends to grow large as more sorts are added to existing connected components by later modules, whereas the number n of arguments per operator seldom rises above 15-20, and the number of declarations for each operator is typically small. Thus our algorithm is adequate in practice.

We say a declaration $s_1 \dots s_n \rightarrow s_0$ subsumes a declaration $s'_1 \dots s'_n \rightarrow s'_0$ from i onwards if $s'_i \dots s'_n \leq s_i \dots s_n$ and $s_0 \leq s'_0$. The intuition is that if we have dealt with all the arguments prior to the i th then declarations that are subsumed from i onwards are in some sense redundant.

We want to define a family of functions min_i on sets of declarations to remove declarations that are considered redundant during the decision diagram construction. To deal the issue of mutual subsumption between declarations we

index the declarations and use the index to break ties. Given a set of declarations $D = \{d_j \mid j \in \{1, \dots, m\}\}$ we define $simplify_i(D)$ to contain all $d_j \in D$ such that for all $d_k \in D$ at least one of the following properties holds:

1. d_k does not subsume d_j from i onwards; or
2. $j \leq k$ and d_j subsumes d_k from i onwards.

Note that (2) is needed to handle mutual subsumption including self subsumption.

The basic algorithm constructs the decision diagram for ls_f top-down as a sequence of $n + 1$ layers starting with the first layer that just contains the root node, and completing each layer before starting the next. The root node is labeled “1” and is associated with the full set of declarations, D_f . For each node N in the i th layer associated with the set of declarations D_N , and each sort s we compute a set of non-redundant *live* declarations as follows: starting with D_N we discard any declarations $s_1 \dots s_n \rightarrow s_0$ such that $s \not\leq s_i$ to get a new set $D_{N,s}$ of *live* declarations. We then apply $simplify_{i+1}$ to form a set of non-redundant *live* declarations $\widehat{D}_{N,s} = simplify_{i+1}(D_{N,s})$.

We now check to see if we have already constructed node N' in the $(i + 1)$ th layer whose associated set of declarations is $\widehat{D}_{N',s}$. If so we just add an arc labeled s from N to N' . Otherwise we create a new node N' labeled $i + 1$ in the $(i + 1)$ th layer whose associated set of declarations is $\widehat{D}_{N',s}$, and then add the arc.

When we reach the $(n + 1)$ th layer each node should be associated with set of declarations that is either singleton or empty. If the singleton case, the range sort of the single declaration provides the label for the leaf node, while in the empty case the label is \top .

If there is a node in the final layer with multiple declarations, these declarations must have incomparable range sorts, and thus we have a witness to non-preregularity. Conversely, if we have minimal witness to non-preregularity, i.e. some term $f(t_1, \dots, t_n)$ with at two incomparable least sorts, s_1 and s_2 , while for $i \in \{1, \dots, n\}$, each t_i has a unique least sort, we can trace the sorts of the t_i 's through the decision diagram and arrive at a leaf node where declarations for both s_1 and s_2 must still be alive. Thus by constructing the decision diagram for each operator f , we also test the signature for preregularity.

4 Computing Order-Sorted Unifiers from Unsorted Unifiers

An *order-sorted substitution* is a map $\theta : Y \rightarrow T_{\Sigma}(X)$ where Y is a finite subset of X such that for $x \in Y \cap X_s$, $ls(\theta(x)) \leq s$. The *range variables* of θ are defined by $rvars(\theta) = \bigcup_{x \in Y} rvars(\theta(x))$.

Given $t, t' \in T_{\Sigma}(X)$, an order-sorted unifier of t and t' is an order-sorted substitution $\theta : rvars(t) \cup rvars(t') \rightarrow T_{\Sigma}(X)$ such that $t\theta = t'\theta$ and $rvars(\theta)$ is disjoint from $rvars(t) \cup rvars(t')$. This disjointness condition prevents cycles and implicitly enforces the so called “occurs check”.

Regular unsorted unification can be solved in linear time [15] and yields a single most general unifier. Given that the free theory meets technical conditions of [9] for extracting order-sorted unifiers from unsorted unifiers to be a correct and complete method for obtaining the order-sorted unifiers, if there were an polynomial time algorithm even for testing whether an order-sorted unifier could be extracted from an unsorted unifier we would expect to be able to decide the existence of order-sorted unifiers in polynomial time in the free theory. We now show that order-sorted unification in the free theory is NP-complete, and thus a polynomial time algorithm for extracting order-sorted unifiers from an unsorted unifier is unlikely.

4.1 NP-Completeness

Given a two free-theory terms t_1, t_2 from a preregular order-sorted signature, deciding where they are unifiable, $t_1 =_? t_2$, is NP-complete.

To see that the problem is in NP we note that given a putative order-sorted unifier σ :

1. $t_1\sigma = t_2\sigma$ can be checked in linear time by instantiation.
2. The least sort of a term t can be computed in polynomial time by proceeding bottom-up and at each operator occurrence, using the naïve algorithm from the previous section.
3. Thus for each assignment $x \leftarrow t$ in σ we can check in polynomial time that $ls(t)$ is less or equal to the sort of x .

To prove NP harness we reduce the classical NP-complete problem of Boolean satisfiability to order-sorted unification by encoding the truth tables of common Boolean operators by subsort overloaded operators in the following signature.

```
fmod ENCODE is
  sorts True False Value .
  subsort True False < Value .
  op true : -> True .
  op false : -> False .
  op not : True -> False .
  op not : False -> True .
  op and : True True -> True .
  op and : True False -> False .
  op and : False True -> False .
  op and : False False -> False .
  op or : True True -> True .
  op or : True False -> True .
  op or : False True -> True .
  op or : False False -> False .
endfm
```

Let ϕ be any propositional formula over propositions p_1, \dots, p_n . We translate ϕ into a term t over the signature ENCODE using the following rules:

- Proposition p_i is replaced by a variable x_i of sort **Value**.
- Each propositional connective is replaced by the corresponding operator.

Let y be a variable of sort **True**. Then the order-sorted unification problem $y =_? t$ has a solution iff ϕ is satisfiable, since y and t are unifiable iff there is an assignment of sorts to x_1, \dots, x_n such that t has sort **True** and the sort declarations exactly mirror the semantics of the propositional connectives.

4.2 Order-Sorted Unifier Extraction

In order to conveniently represent unsorted unifiers in our order-sorted framework we assume that the poset of sorts $\langle S, \leq \rangle$ is enriched with a top element \top and the family of sets of variables X is enriched with a set X_\top . Each operator $f \in \Sigma_{s_1 \dots s_n, s}$ is lifted to $f \in \Sigma_{\top^n, \top}$ so that every syntactically well-formed term belongs at least to $T_\Sigma(X)$.

We use variables from X_\top to represent unsorted variables. An unsorted unifier will be a substitution $v : Y \rightarrow T_\Sigma(X_\top)$ such that $rvars(v)$ is disjoint from Y .

An *order-sorted renaming* of v is an injective map $\rho : rvars(v) \rightarrow X \setminus Y$ such that for each $x \in Y$, $ls(tv\rho) \leq ls(x)$. Intuitively ρ provides sorts for the variables in the unsorted unifier, allowing the composition $\rho \circ v$ to meet the requirements of an order-sorted unifier.

In fact we are not interested in all such order-sorted renamings. Given two order-sorted renamings $\rho, \rho' : Z \rightarrow X$ for some finite $Z \subset X_\top$, ρ is more general than ρ' , denoted $\rho' \preceq \rho$ iff for all $x \in Z$, $ls(\rho'(x)) \leq ls(\rho(x))$. What we really want is a set R of order-sorted renamings for v such that for every order-sorted renaming ρ of v , there is some $\rho' \in R$ such that ρ' is more general than ρ .

The method of computing such sets of renamings proposed in [11] is essentially top-down constraint propagation, where the sort of each $x \in Y$ provides a constraint on the sort of each $v(x)$ and these constraints are percolated down into the subterms using the declarations for each operator until a set of constraints is found for the sort of each variable in $rvars(v)$. The set of constraints on the sort of each variable x determine if there is a valid sort that can be used in the construction of an order-sorted renaming. Where an operator has multiple declarations or when greatest lower bound of a pair of sorts is non-unique, multiple paths may have to be followed, leading to multiple order-sorted renamings.

4.3 Bottom-up Symbolic Sort Computation

Although the problem is expressed in terms of finding order-sorted renamings, the names of the new variables are unimportant and the core issue is to find mappings from $rvars(v)$ to S that give suitable sorts to the range variables of an unsorted unifier. Since $rvars(v)$ is finite, each such mapping can be represented as a vector of sorts. We treat these vectors of sorts as unknowns that have to be computed.

Our approach is to lift our bottom-up scheme for computing the sorts of ground terms to a bottom-up scheme for computing sort functions on open terms, where the unknown sorts of the variables are represented symbolically. We then build symbolic constraints that assert that for each bound variable $x \in Y$, $ls(v(x)) \leq ls(x)$. We then form the conjunction of these constraints and compute the solutions of this conjunction that are maximal on the assignment of sorts to free variables.

The set of sorts S and the sort functions ls_f that operate on it can be considered a finite data type and we need a symbolic representation of this data type on which we can perform the above manipulations.

4.4 Boolean Decision Diagrams

Boolean Decision Diagrams (BDDs) can be viewed as compressed truth tables. For random Boolean functions they are less compact than truth tables. However Boolean functions of interest often have a lot of structure and in practice BDDs can be much more compact than truth tables — perhaps even exponentially so if we are lucky. Many useful operations such as combination by an arbitrary Boolean connective, composition, remapping of variables and variable elimination by instantiation, universal quantification or existential quantification can be performed efficiently [5]. BDDs are a standard tool in hardware design and verification and several implementations are available as open source libraries.

We encode each sort $s \in S$ (where S is assumed to include a top sort \top) by a binary code of length $l = \lceil \log_2(|S|) \rceil$ and unknown sorts by vectors of BDD variables of length l . Since remapping BDD variables, and composing BDDs are standard BDD operations, for the purposes of exposition we will treat BDDs as functions and operations on BDDs as functors that return new functions, with the technical details of remapping BDD variables and composing BDDs largely hidden by function notation.

4.5 Precomputations

We precompute a number of BDDs based on the signature. We start with two families of structures that are constructed directly from primitive BDD operations. For each sort $s \in S$ we construct:

1. The vector of constant BDDs, bddvec_s , corresponding to the binary code of s .
2. A BDD $\text{isSort}_s : \mathbf{B}^l \rightarrow \mathbf{B}$, where $\text{isSort}_s(\mathbf{b})$ is true if \mathbf{b} is the binary code of s and false otherwise.

The remaining BDD structures we need are constructed by applying standard BDD operations to these initial structures.

For each sort $s \in S$, we compute a BDD $\text{leq}_s : \mathbf{B}^l \rightarrow \mathbf{B}$,

$$\text{leq}_s(\mathbf{b}) = \bigvee_{s' \leq s} \text{isSort}_{s'}(\mathbf{b})$$

by a succession of disjunction operations. Intuitively this captures the notion that \mathbf{b} encodes a valid sort s' and that $s' \leq s$.

We compute a BDD $\text{gt} : \mathbf{B}^l \times \mathbf{B}^l \rightarrow \mathbf{B}$,

$$\text{gt}(\mathbf{b}, \mathbf{b}') = \bigvee_{s \in S} (\text{isSort}_s(\mathbf{b}) \wedge \neg \text{isSort}_s(\mathbf{b}') \wedge \text{leq}_s(\mathbf{b}'))$$

by a succession of disjunctions of conjunctions. Intuitively this captures the notion that \mathbf{b} and \mathbf{b}' encode valid sorts s and s' such that $s > s'$.

In order to compute a BDD representation of ls_f we need to generate a vector of BDDs which we view as a function $\text{ls}_f : (\mathbf{B}^l)^n \rightarrow \mathbf{B}^l$. If f has m declarations we construct ls_f by a series of approximations, A_0, A_1, \dots, A_m , starting with

$A_0 = \text{bddvec}_\top$. For the j th declaration of f , $(s_1 \dots s_n) \rightarrow s$, we compute a BDD $\text{applicable}_j : (\mathbf{B}^l)^n \rightarrow \mathbf{B}$,

$$\text{applicable}_j(\mathbf{b}_1, \dots, \mathbf{b}_n) = \bigwedge_{i \in 1, \dots, n} \text{leq}_{s_i}(\mathbf{b}_i)$$

by a succession of conjunctions. This captures the notion of sorts encoded by $\mathbf{b}_1, \dots, \mathbf{b}_n$ being less or equal to $(s_1 \dots s_n)$. We also compute a BDD $\text{notGreater}_j : (\mathbf{B}^l)^n \rightarrow \mathbf{B}$,

$$\text{notGreater}_j = \neg(\text{leq}_s \circ A_{j-1})$$

using vector composition and negation. This captures the notion that s is not greater than the sort that would be computed by the previous approximation A_{j-1} . Finally we compute the new approximation $A_j : (\mathbf{B}^l)^n \rightarrow \mathbf{B}^l$ using the BDD *ite* (or “if then else”) operator which is applied pointwise to the vectors bddvec_s and A_{j-1} ,

$$A_j = \text{ite}(\text{applicable} \wedge \text{notGreater}, \text{bddvec}_s, A_{j-1})$$

When all m declarations for f have been processed, we have $\text{ls}_f = A_m$.

It is important to realize that in these computations we are applying functors to Boolean functions and thus (symbolically) performing computations on all possible arguments to those functions. In particular the test $\text{applicable}_j \wedge \text{notGreater}_j$ is neither true nor false but is a function dependent on its inputs.

4.6 Unification-Time Computations

We start with an unsorted unifier $v : Y \rightarrow T_\Sigma(X_\top)$ such that $\text{rvars}(v) = \{x_1, \dots, x_m\}$ is disjoint from Y . We want to find most general order-sorted renamings $\rho : \text{rvars}(v) \rightarrow X$ such that for each $y \in Y$, $\text{ls}(yv\rho) \leq \text{ls}(y)$.

For each $i \in \{1, \dots, m\}$ we allocate a vector \mathbf{z}_i of l BDD variables, that will encode the new sort that x_i will receive when it is renamed by a generic renaming $\rho : \text{rvars}(v) \rightarrow X$. In order to ensure that the value of the BDD vector always corresponds to a valid sort (since $|S|$ might not be a power of 2, there may be some unused codes) we need to enforce the constraint that $\text{leq}_\top \circ \mathbf{z}_i$ is true.

For each $y \in Y \cap X_s$ we perform a bottom-up computation on the term yv to generate a vector \mathbf{L}_y of BDDs which encodes $\text{ls}(yv\rho)$. In the base case we have either a constant $c \in F_{\lambda, s'}$ in which case the required vector is bddvec'_s or a variable x_i in which case the required vector is \mathbf{z}_i . In the induction case if we have a term $t = f(t_1, \dots, t_n)$ and we have computed vectors of BDDs, $\mathbf{q}_1, \dots, \mathbf{q}_n$ encoding $\text{ls}(t_1\rho), \dots, \text{ls}(t_n\rho)$, we generated the vector of BDDs for t by the BDD vector composition

$$\text{ls}_f \circ (\mathbf{q}_1, \dots, \mathbf{q}_n)$$

In order to ensure that $\text{ls}(yv\rho) \leq s$ we enforce a constraint that $\text{leq}_s \circ \mathbf{L}_y$ is true.

Forming a conjunction of of both kinds of constraints we end up with a BDD $\text{osr} : (\mathbf{B}^l)^m \rightarrow \mathbf{B}$,

$$\text{osr} = \left(\bigwedge_{i \in \{1, \dots, m\}} \text{leq}_\top \circ z_i \right) \wedge \left(\bigwedge_{y \in Y \cap X_s} \text{leq}_s \circ L_y \right)$$

This BDD is true exactly on those valuations of the vectors of BDD variables z_i that correspond to order-sorted renamings. However the number of these valuations will in most cases be very large; what we really want is another BDD mostGeneral which is true on exactly on those valuations of the vectors of BDD variables z_i that correspond to most general order-sorted renamings.

Suppose some order-sorted renaming ρ is not a most general order-sorted renaming. Then there must be some order-sorted renaming ρ' that is strictly more general. Because of the monotonicity of the sort functions ls_f , there must exist a order-sorted renaming ρ'' such that exactly one variable, x_i , has $ls(\rho''(x_i)) > ls(\rho(x_i))$. We capture this notion with the BDD $\text{moreGeneral}_i : (\mathbf{B}^l)^m \rightarrow \mathbf{B}$,

$$\text{moreGeneral}_i(z_1, \dots, z_m) = \exists \mathbf{q}. [\text{gt}(\mathbf{q}, z_i) \wedge \text{osr}(z_1, \dots, z_{i-1}, \mathbf{q}, z_{i+1}, \dots, z_m)]$$

This construction is done by introducing a new vector \mathbf{q} of BDD variables to compute the function inside the existential quantification via BDD variable remapping and then using the BDD operation of variable elimination by quantification to eliminate these new BDD variables.

Finally we capture the notion of a most general order-sorted renaming as a order-sorted renaming such that there is not a strictly more general order-sorted renaming by conjuncting osr with the constraint that moreGeneral_i be false for all $i \in \{1, \dots, m\}$. This yields the BDD we want, $\text{mostGeneral} : (\mathbf{B}^l)^m \rightarrow \mathbf{B}$,

$$\text{mostGeneral} = \text{osr} \wedge \bigwedge_{i \in \{1, \dots, m\}} \neg \text{moreGeneral}_i$$

by a succession of conjunctions.

Having computed the BDD mostGeneral we now have a representation of all most general order-sorted renamings. The renamings themselves are recovered by finding the valuations of the BDD variables $\{z_i \mid i \in \{1, \dots, m\}\}$ that make mostGeneral evaluate to true. Given such a valuation, the value of each vector z_i corresponds to the binary code of a sort s_i and the corresponding renaming should map x_i to a fresh variable of that sort.

The extraction of such valuations from a BDD depends on the technical details of BDDs but is straightforward:

- We trace the from the root of the BDD to the true terminal.
- Each such path corresponds to one or valuations, given by the labels on arcs taken.
- Where some BDD variable is not mentioned on a path, both the true and false values must be used, giving rise to multiple valuations.

Each valuation is extracted using time that is linear in the number of BDD variables.

5 Extensions

We now briefly describe some useful algorithms based on simple extensions of the algorithms presented so far.

5.1 Congruence Class Checks

Congruence class matching, unification and rewriting are much studied topics [8,1]. Notionally rather than computing with terms t these techniques compute with congruence classes of terms, $[t]_E$, under some equational theory E . In practice, since the congruence classes can be large or even infinite, implementations work with a chosen representatives rather than the actual congruence classes.

While they are beyond the scope of this paper, the order-sorted extensions of these techniques typically require some restrictions on the sort structure of the order-sorted signature. In this section we consider how such checking such restrictions can be sped up using sort decision diagrams.

Permutative theories are those where every equational axiom is permutative. An equation is permutative if the multiset of variables and functions symbols occurring in the left-hand side is equal to the multiset of variables and functions symbols occurring in the right-hand side. In practice the two most important permutative axioms are commutativity, $f(x, y) = f(y, x)$ and associativity $f(f(x, y), z) = f(x, f(y, z))$.

For ordered sorted computations on congruence classes it is convenient if all members of a congruence class have the same least sort, so that the selection of congruence class representatives is independent of order-sortedness. This can be ensured by requiring the sort function ls_f for each function f with axioms to satisfy those axioms.

For commutativity, this can be ensured by *commutative completion* on the signature before a sort decision diagram is constructed; for each $f \in F_{s_1 s_2, s}$ we add f to $F_{s_2 s_1, s}$.

For associativity, we need to check the associativity of ls_f . For an operation f that is associative-commutative we do the commutative completion first and check the ls_f corresponding to the completed signature. The naïve solution is just to compare $ls_f(ls_f(u, v), w)$ and $ls_f(u, ls_f(v, w))$ for each triple of sorts (u, v, w) using the sort decision diagram for each evaluation of ls_f . This requires $O(|S|^3)$ time.

Let L be the set of nodes in the second layer of the sort decision diagram for ls_f . Clearly $|L| \leq |S|$ and in practice L is often significantly smaller than S , especially for large S . A more sophisticated algorithm performs the associativity check in $O(|L| \cdot |S|^2)$ time.

The key idea is that each element of $l \in L$ represents a partial evaluation of ls_f ; that is $ls_l(s) = ls_f(c, s)$ for some sort c . To check $ls_f(ls_f(u, v), w) = ls_f(u, ls_f(v, w))$ for each triple of sorts (u, v, w) it suffices to check $ls_f(ls_l(v), w) = ls_l(ls_f(v, w))$ for each $l \in L$ and each pair of sorts (v, w) . Furthermore by keeping a map which takes each $l \in L$ to some sort s such there is a arc from the root node to l labeled by s , if an inequality is discovered, an example triple can be

exhibited. Also by keeping a map which takes each $l \in L$ to a count of the number of arcs from the root node to l , each time an inequality is discovered, the relevant count can be added to a total so that a final total of the number of triples on which ls_f fails the associativity test is computed.

5.2 Towers of Function Symbols

The standard way to represent natural numbers in an algebraic specification is with successor notation. This is convenient for writing rewrite rules, however for large numbers the terms become unwieldy. Maude provides a special internal representation for huge stacks of unary symbols where the height of the stack is represented by a arbitrary precision number, and special matching and unification algorithms maintain the illusion that the virtual stack of unary operators is really there. However this means that matching and unification sort computations must be percolate through these huge virtual stacks of unary symbols efficiently.

The key observation is that the set of sorts is finite, and typically quite small, and thus when we look a stack $f^n(t)$ of n unary function symbols f above some term t for increasing values of n , there must be some point at which a sort repeats (if n is large enough) and there after cycles with some period p . So we merely have to keep track of the sorts up to and including the first cycle.

For each sort s we compute and store a vector of sorts s_1, s_2, \dots, s_k where $s_i = ls(f^i(x))$ for $x \in X_s$, and i is the smallest integer such that $ls(f^{i+1}(x)) = s_r$ for some $r \leq i$. Thereafter the sorts will cycle with period $i + 1 - r$.

To compute $ls(f^n(t))$ for $n \leq i$ we simply get the vector of sorts for $s = ls(t)$ and look up the entry s_n ; for $n > i$ we look up the entry s_c where $c = r + ((n - r) \bmod (i + 1 - r))$.

For unification we need to lift this computation to $ls(f^n(t))$ where $ls(t)$ is given by a vector of L of BDDs over some vector of BDD variables z . We want to compute a vector M of BDDs over the same vector of BDD variables such that for each valuation of z where L evaluates to the binary code of a valid sort s , M evaluates to the binary code of $ls(f^n(x))$ where $x \in X_s$. Since we do not know n in advance we cannot precompute a vector of BDDs representing the sort mapping induced by f^n and perform a vector composition with L . Also since n can be arbitrarily large it is not practical to precompute ls_f as we did in §4.5 and perform an iterative vector composition.

Instead we expect $|S|$ to be relatively small (at least compared to n) and build up M by considering each sort s in turn, at runtime. For each $s \in S$ we compute $s' = ls(f^n(x))$ where $x \in X_s$, using the method given above and construct a vector of BDDs

$$\text{mapSort}_s = \text{ite}(\text{isSort}_s \circ L, \text{bddvec}_{s'}, \text{false}^l)$$

The intuition is that mapSort_s returns the binary code for s' exactly when the L evaluates to the binary code of s , and returns a vector of false BDDs otherwise.

We then form M as a disjunction

$$M = \bigvee_{s \in S} \text{mapSort}_s$$

so that all sorts $s \in S$ are handled.

6 Final Remarks

We have described an algorithm for compiling the declarations of an operator f into a decision diagram encoding the least sort function ls_f for that operation that allows ls_f to be evaluated in linear time. We have shown how ls_f and relations on sorts can be encoded using BDDs so that given an unsorted unifier, a Boolean function that evaluates to true on exactly those valuations that correspond to most general order-sorted renamings of the free variables can be constructed, and how the most general order-sorted renamings can be extracted. We have also shown how the decision diagram can be used to accelerate associativity checks on ls_f and how both the least sort computation and the BDD based computation of most general order-sorted renamings can be efficiently extended to towers of unary operators where the tower height is given by an arbitrary precision number.

References

1. Baader, F., Snyder, W.: Unification theory. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, ch. 8, pp. 445–532. Springer, Berlin (2001)
2. Boudet, A.: Unification in a combination of equational theories: an efficient algorithm. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 292–307. Springer, Heidelberg (1990)
3. Boudet, A.: Competing for the AC-unification race. Journal of Automated Reasoning 11, 185–212 (1993)
4. Boudet, A., Contejean, E., Devie, H.: A new AC-unification algorithm with a new algorithm for solving diophantine equations. In: Proceedings of the 5th IEEE Symposium on Logic in Computer Science, pp. 289–299. IEEE Computer Society Press (1990)
5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35, 677–691 (1986)
6. Bürckert, H.-J., Herold, A., Kapur, D., Siekmann, J.H., Stickel, M.E., Tepp, M., Zhang, H.: Opening the AC-unification race. Journal of Automated Reasoning 4, 465–474 (1988)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 243–320. MIT Press, Cambridge (1990)

9. Goguen, J.A.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105, 217–273 (1992)
10. Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P.: Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International (March 1992)
11. Hendrix, J., Meseguer, J.: Order-sorted equational unification revisited. In: *Proceedings of RULE 2008*. *Electronic Notes in Theoretical Computer Science*, Elsevier Science (2008)
12. Herold, A.: Combination of unification algorithms. In: Siekmann, J.H. (ed.) *CADE 1986*. LNCS, vol. 230, pp. 450–469. Springer, Heidelberg (1986)
13. Meseguer, J., Goguen, J.A., Smolka, G.: Order-sorted unification. *Journal Symbolic Computation* 8, 383–413 (1989)
14. Moret, B.M.E.: Decision trees and diagrams. *ACM Computing Surveys* 14, 593–623 (1982)
15. Paterson, M.S., Wegman, M.N.: Linear unification. *Journal of Computer and System Sciences* 16(2), 158–167 (1978)
16. Schmidt-Schauß, M.: Unification in a combination of arbitrary disjoint equational theories. *Journal of Symbolic Computation* 8(1-2), 51–99 (1989)

Solving the First Verified Software Competition Problems Using PVS*

Sam Owre and Natarajan Shankar

Computer Science Laboratory,
SRI International, Menlo Park CA 94025 USA
shankar@csl.sri.com
<http://www.csl.sri.com/~shankar/>

For Carolyn, inspiring teacher, mentor, and colleague

Abstract. The first Verified Software Competition (VSComp) was held in August 2010 as part of the third conference on *Verified Software: Theories, Tools, and Experiments*. The competition consisted of five problems of varying difficulty. These problems have been useful for illustrating the strengths and weaknesses of different verification methods. We present solutions to these problems using the SRI's Prototype Verification System (PVS). We also discuss how certain features of PVS were exploited in these exercises.

1 Introduction

Verification technology has reached a level of maturity where we can expect programs to be developed along with their specification and proofs of correctness. The Verified Software Grand Challenge initiated by Tony Hoare [Hoa03] has as its goal the widespread adoption of formal verification technology in the development of large-scale software systems. The conference *Verified Software: Theories, Tools, and Experiments* (VSTTE) is a forum for addressing this grand challenge. As a step toward measuring progress, the Verified Software Competition (VSComp) [KMS⁺11] was initiated at the 2010 VSTTE conference in Edinburgh, Scotland. This competition was used to evaluate the effectiveness with which verification tools could be used to specify and verify software. Since the competition, several research groups have used the challenge problems from the competition to evaluate their verification systems. We report on such an evaluation for SRI's Prototype Verification System (PVS) [ORSvH95]. The experiments here benchmark the capabilities of PVS as a specification/verification environment and suggest avenues for improvement. Our presentation can also be seen as a tutorial on the application of PVS in verification.

Eleven teams participated in the first Verified Software Competition (VSComp). Each team could contain at most three members, but many teams had just one or two members. Five problems were posed together with pseudocode by the organizers Peter Müller and Natarajan Shankar (with help from Valentin Wüstholtz). After a thinking

* This research was supported by NSF Grants CSR-EHCS(CPS)-0834810 and CNS-0917375 and by NASA Cooperative Agreement NNX08AY53A.

period of four hours, the participants were given two hours to solve as many of these problems as well as they could. Twenty-one solutions (some partial) were presented and evaluated by the judges (Gary Leavens, Peter Müller, and Natarajan Shankar). After the competition, several research groups completed the verification of many of the problems. A report [KMS⁺11] summarizing the results from the VSComp competition appears in the Proceedings of the conference on Formal Methods (FM2011).

We describe PVS solutions to these five problems. None of the teams appearing in the competition used PVS, but four of the five problems were solved using PVS prior to the competition. Several of the teams in the competition used tools for assertional program verification where the proof obligations were discharged using simplifiers such as those based on solvers for satisfiability modulo theories (SMT) [dMDS07, BTSS09]. Other teams used interactive theorem provers such as HOL 4 [GM93, SN08] and Isabelle [NPW02]. PVS is closer to the latter style of formalization, but the automation in PVS exploits proof obligation generation where the proof obligations are proved using simplifiers based on SMT solvers. The PVS formalization highlights the interplay between the specification language and the inference mechanisms. We use the PVS solutions to discuss the opportunities and challenges for verification technology.

Section 2 gives a brief overview of PVS. In Sections 3 to 7, we present the PVS solutions for each of the five problems in the competition. In Sec 8, we use the results to discuss the strengths and weaknesses of PVS.

2 Prototype Verification System

PVS is an interactive verification system developed and maintained by the SRI International Computer Science Laboratory. It was first released in 1993, and PVS version 5.0 is the latest version of this system. In PVS, both specifications and programs are written in higher-order logic. The simple types in the system include tuples, records, and functions built from primitive types like the Booleans and numbers. More complex types can be defined as

1. Predicate subtypes, e.g., the type of even numbers or order-preserving maps
2. Structural subtypes, e.g., records with additional fields
3. Dependent function, tuple, and record types, e.g., a stack represented as a pair consisting of a *size* field and an *elements* field that is an array with indices ranging from 0 to *size* - 1
4. Recursive and co-recursive datatypes, e.g., lists, trees, and streams.

Typechecking expressions relative to predicate subtypes and dependent types can trigger the generation of proof obligations (called Type Correctness Conditions or TCCs). Checking the well-typedness of expressions relative to simple types is decidable, but since any formula can be employed in the predicate part of a predicate subtype, the resulting TCCs need not be in a decidable class.

PVS specifications are given in the form of theories that contain a list of constant and formula declarations. Theories can have parameters, so that one can, for example, define list operations over an unspecified element type. Theory interpretations [OS01] can be

used to interpret theories to either demonstrate the consistency of an axiomatic theory or to build a concrete instance of an abstract theory. The system has extensive libraries of theories covering algebra, analysis, trigonometry, measure theory, and probability.

Much of the PVS specification language is executable as a functional programming language. There are code generators that transform the specifications to code in Common Lisp [Ste90] and the Clean programming language [PvE99]. The code generator employs update analysis to identify safe, destructive updates so that the generated code is efficient in both time and space performance. PVSio [Muñ03] is a plug-in for using the generated code in evaluating ground expressions within proofs as well as in software development.

PVS features an interactive theorem prover that can be used to compose proofs by combining various forms of automated inference. A typical proof is constructed by a human user providing the induction, case analysis, and quantifier instantiations, and suggesting lemmas, while the automation takes care of the Boolean and arithmetic simplification using binary decision diagrams and SMT solvers, heuristic quantifier instantiation, and rewriting with definitions and rewrite rules. Users can also write proof strategies that capture typical patterns of inference. In addition to proving theorems, Interactive proofs in PVS are used for exploring mathematical concepts and conjectures.

3 Problem 1: Summation and Maximum

The first problem in VSComp-1 is to write a program that computes the sum and maximum of the elements in an array a of N elements. Given that $N \geq 0$ and $a[i] \geq 0$ for $0 \leq i < N$, the verification task is to prove termination and establish the post-condition that $\text{sum} \leq N * \text{max}$.

In PVS, this is formalized in the theory `SumMax`. This theory has a parameter N of type `nat`, the subtype consisting of non-negative integers. Arrays are just functions where the domain type is `below(N)`, the subtype of natural numbers below N . Note that the type `below(N)` could be empty in the case when the parameter N is zero. Types are not assumed to be nonempty. Two variables A and B are declared to range over this array type. The function `SumMax` is defined recursively. This operation takes two arguments: the array A and an index i in the subrange from 0 to N . The range type of the function is a dependent tuple. The first component of this tuple is a natural number representing the maximal element in the array segment for the indices below i . The second component represents the sum of the array elements $A(0)$ to $A(i)$. The predicate subtype `upto(i * max)` restricts the second component to be at most $i * \text{max}$, where `max` is the first component of the tuple. The `SumMax` function is defined as the tuple $(0, 0)$ when i is 0 since the maximum and summation are both 0 in this case. When i is greater than 0, the `LET` construct pattern-matches the result (m, s) from the recursive invocation of `SumMax`. The first component of the result is $A(i-1)$ if $A(i-1)$ is greater than m , and is m , otherwise. The second component of the result is $A(i) + s$. The recursive definition is given a termination measure, the parameter i , that is required to decrease with each recursive invocation.


```

SumMax [N: nat]          : THEORY

BEGIN

A, B: VAR [below(N) -> nat]

SumMax(A, (i: upto(N))): RECURSIVE
[ max: nat,
  upto(i*max) ] =
IF i = 0
THEN (0, 0)
ELSE (LET (m, s) = SumMax(A, i-1)
      IN ((IF m < A(i-1) THEN A(i-1) ELSE m ENDIF),
         A(i-1) + s))
ENDIF
MEASURE i

AvgMax: LEMMA
SumMax(A, N)^2 <= N * SumMax(A, N)^1

END SumMax

```

This definition generates six TCCs, two of which are subsumed by others. Of the four remaining TCCs, the first requires that $i - 1$ in the expression $A(i - 1)$ is in the expected subrange $\text{below}(N)$, i.e., that the array-bounds check is valid. The third TCC checks that the occurrence of $i-1$ in $\text{SumMax}(A, i-1)$ is in the expected subtype $\text{upto}(N)$. The fourth TCC checks that the termination measure applied to the recursive call, i.e., $i-1$, is smaller than the parameter i in the left-hand-side of the definition. The above TCCs are all proved by the default strategy. The second TCC checks that the result, the pair $((\text{IF } m < A(i-1) \text{ THEN } A(i-1) \text{ ELSE } m \text{ ENDIF}), A(i-1) + s)$, is in the dependent tuple type $[\text{max: nat}, \text{upto}(i*\text{max})]$. In this case, we have as an antecedent the information that the pair (m, s) is already of the required type. This way, a property that would normally require an explicit induction is being proved implicitly using proof obligations generated from the definition. This TCC is the only one that requires an interactive proof of three steps, one of which suggests a lemma from the nonlinear arithmetic library in the PVS prelude library.

The property of interest can then be proved from the typing of the function `SumMax` in the lemma `AvgMax`. This lemma is proved automatically with a single `grind` command which repeatedly applies a series of strategies that introduce Skolem constants for quantifiers of universal strength, instantiate existential strength quantifiers, and apply simplification using decision procedures and rewrite rules. The type information is used implicitly in the proof, since it is passed on to the decision procedures used by PVS.

The proof above illustrates the use of dependent/predicate subtyping in ensuring that any array indices are within bounds and for implicitly performing induction through type constraints.

4 Problem 2: Inverting an Array

The second problem is that of inverting an injective array A on N elements in the subrange from 0 to $N - 1$ so that the output array B is such that $B[A[i]] = i$ for $0 \leq i < N$. In the competition, it was okay to assume that an injective array from

$0..N - 1$ to $0..N - 1$ is also surjective. Bonus points were given for proving that the inverted array B is also injective.

The inversion operation is defined recursively by the function `inverse_rec` which takes an array f and an index i and returns an array where the first i elements of f have been inverted, i.e., an array g where $g(f(j)) = j$ for $0 \leq j < i$. The “post-condition” on the result g is captured by the predicate `inverse_below?` which is used in the range type of the function `invert_rec`. In the base case, this function returns an array where each element is 0. This raises a couple of questions. One, what happens when N is 0, and the array/function type `[below(N) -> below(N)]` has an empty domain and range? Types can be empty in PVS. A function with an empty domain cannot be applied to an argument, and hence the empty range does not pose a problem. Two, could the return value in the base case have been replaced with the argument f ? Indeed, it can be replaced and verified in the same way.

The definition of `invert_rec` generates five TCCs, of which only the implicit induction proof obligation triggered by the post-condition requires a short interactive proof of five steps. The remaining TCCs are proved by their respective default strategies.

```

inversion : THEORY

BEGIN

N : VAR nat

inverse_below?(N) (f, g: [below(N) -> below(N)] (i: upto(N)): bool =
(FORALL (j: below(i)): g(f(j)) = j)

invert_rec(N) (f: (injective?[below(N), below(N)]),
i: upto(N)): RECURSIVE
{g : [below(N) -> below(N)] | inverse_below?(N) (f, g) (i)} =
(IF i = 0
THEN (LAMBDA (j: below(N)): 0)
ELSE (LET g = invert_rec(N) (f, i-1)
IN g WITH [(f(i-1)) := i-1])
ENDIF)
MEASURE i

:
:

END inversion

```

The operation `invert` invokes `invert_rec` with N for i . The claim that `invert(N) (A)` is injective is established by the lemma `injective_invert`. This theorem requires a fairly interactive proof with about seventeen steps. This proof uses a corollary to the pigeonhole principle which asserts that any injection on `below(N)` must be a surjection. Once we know that A is a surjection, then to any x and y in `below(N)`, there are x' and y' such that $A(x') = x$ and $A(y') = y$. By the type constraint on `invert_rec`, we have that `invert(N) (A) (x) = x'` and `invert(N) (A) (y) = y'`, so if $x' = y'$, then $x = A(x') = A(y') = y$ must hold.

```

invert(N) (A: (injective?[below(N), below(N)])) (i: below(N)): below(N) =
  invert_rec(N) (A, N) (i)

IMPORTING pigeon

injective_invert: LEMMA
(FORALL (A: (injective?[below(N), below(N)]))):
  injective?(invert(N) (A))

```

5 Problem 3: Searching in a List

The third problem is to write a function that finds the index of the first element in a linked list representation of a list of integers that is equal to 0. If there is no 0 in the list, then the function returns the length of the list. The task is to show that when the list does not contain 0, the function returns the length of the list, and when it contains 0, the function returns the index of the first such occurrence. The PVS formalization below solves the general problem of finding the index of the first occurrence of an element in a list.

PVS has a facility for defining recursive datatypes like lists, trees, and ordinals in terms of constructors, accessors, and recognizers. These definitions generate theories with predefined operations. The PVS prelude contains further definitions of list operations such as `length` and `append`. In particular, it contains that operation `nth` such that `nth(LL, i)` is the element at position `i` in the list starting from position 0. The theory `list_index` takes the element type `T` as a parameter. The function `find` is defined recursively to find the index of an element `l` in the list `LL`. The range type of this function is dependent on the input values `LL` and `l` with a type constraint that ensures that the result is an index of value at most the `length(LL)` such that all the elements preceding the element at the result index in list `LL` are different from `l`, and unless the result is the length of the list, the element at the result index is `l`. The definition generates five TCCs, and the implicit induction proof obligation generated by the type constraint on the range requires seven interactions.

```

list_index [T: TYPE ]
           : THEORY
BEGIN
  l, m, n: VAR T
  LL, MM, NN: VAR list[T]

  find(LL, l) : RECURSIVE
    {i: upto(length(LL)) | (FORALL (j: below(i)): nth(LL, j) /= l)
      AND (i < length(LL) => nth(LL, i) = l)}
  = (CASES LL OF
    null: 0,
    cons(m, MM): IF m = l THEN 0 ELSE 1 + find(MM, l) ENDIF
    ENDCASES)
    MEASURE length(LL)

END list_index

```

The lemmas below are all proved with a single strategy that first simplifies the goal using the `grind` strategy, then explicitly introduces type constraints on subterms into the sequent before applying `grind` once more.

```

find_length: LEMMA
  (FORALL (j: below(length(LL))): nth(LL, j) /= 1)
  IMPLIES find(LL, 1) = length(LL)

find_first: LEMMA
  (FORALL (j: below(find(LL, 1))): nth(LL, j) /= 1)

find_finds: LEMMA
  find(LL, 1) < length(LL) => nth(LL, find(LL, 1)) = 1

```

6 Problem 4: N-Queens

The fourth problem in the competition is to write (and verify) a program to place N queens on an $N \times N$ chess board so that no queen can capture another one with a legal move. For a given N , the algorithm must return a valid solution if there is one, and a flag if there is no such solution. This was, by quite a margin, the most interesting and challenging of the problems in the competition. None of the teams attempted it during the competition. The solution in PVS again exploits the combination of higher-order types and predicate subtypes. Here, we solve the more general problem of finding an assignment satisfying any given predicate, and not just the N-Queens condition.

The theory `nqueens` takes a parameter N representing the number of rows/columns in the chess board. The board is represented as an array from `below(N)` to itself so that a board `queen(i)` indicates that row number of the queen position on the i 'th column. The predicate `extends` holds of an index i and two boards A and $queen$ iff these boards agree on their first i elements.

```

nqueens    [N: nat ]
           : THEORY

BEGIN

  board : TYPE = [below(N)->below(N)]
  A, B, queen, new_queen: VAR board

  i, j, k: VAR upto(N)

  extends(i, A, queen): bool =
    (FORALL (j: below(i)): A(j) = queen(j))

  p: VAR [board -> bool]

  :
  :

END nqueens

```

The type `lift[T]` for any type T is a datatype with two constructors: `bottom`, which represents a failure flag or an undefined value, and `up(X)` for an expression X of type T . A higher-order predicate `qlift?` is defined to apply to a predicate p on boards and a lifted board x so that when input x is `bottom`, it must be the case that there is no board satisfying the predicate p , and when the input is of the form `up(queen)`, then $p(queen)$ holds. A board B is said to be a good extension of a board A with respect to index i and predicate p if $p(B)$ holds and A and B agree on the first i entries.

```

qlift?(p)(x : lift[board]): bool =
  CASES x OF
    bottom: (FORALL queen: NOT p(queen)),
    up(queen): p(queen)
  ENDCASES

%A good extension is one that satisfies p

good_extension?(i, A, p)(B): bool =
  (p(B) AND extends(i, A, B))

```

The assignment of queens is performed by two functions `search` and `scan` that search for an assignment extending a partial assignment of queens to columns. The `scan` operation iterates over columns. In each column `i`, it invokes the `search` operation to iterate over the row positions to find a row index `j` that can be extended to a valid N-Queens assignment. Both operations return a lifted board such that when the board is defined, it is a good extension of the given board with respect to index `i`. The invariant captured by the dependent type for `search` is critical to the correctness arguments. It asserts that given a column index `i`, a partially assigned board `A`, and the predicate `p`

1. The argument `j` representing the row index along column `i` must be such that for every row index `k` smaller than `j`, there is no good extension of board `A` with the queen for the `i`'th positioned at row `k`.
2. The operation `f` must be capable of extending a partially assigned board `B` to a good extension, if there is one. Thus, `search` updates `A` at column `i` to the row index `j` to obtain board `B`. The operation `f` is invoked on `B`, and if it fails to yield a good board, then `search` is invoked recursively with the parameter `j` incremented by one.
3. The use of the higher-order argument `f` allows the `scan` operation to invoke `search` with the recursive call `scan(i+1, p)`, the operation that continues the iteration over columns from column `i+1`, as the actual argument. This allows the mutual recursion between `scan` and `search` to be defined using the recursive invocation of `scan` to be passed as a higher-order parameter, similar to a continuation, to the invocation of `search` from `scan`.
4. The dependent type for `scan` can be used to ensure that this actual parameter matches the expected type for the formal parameter `f`.
5. The result returned by `search` is a good extension of `A` if there is one.

```

search((i: below(N)), A, p,
  (j | (FORALL (k: below(j)), B):
    NOT good_extension?(i+1, A WITH [i:= k], p)(B))),
  (f: [B: board -> (qlift?(good_extension?(i+1, B, p)))]): RECURSIVE
(qlift?(good_extension?(i, A, p))) =
(IF j = N THEN bottom
 ELSE LET B = A WITH [i := j]
  IN CASES f(B) OF
    bottom: search(i, A, p, j+1, f),
    up(C): up(C)
  ENDCASES
ENDIF)
MEASURE N - j

```

The proof obligations for the dependent type corresponding to the `scan` operation can be discharged using the declared type for `search`. The use of the higher-order parameter `f` in `search` allows the termination arguments for `search` and `scan` to be performed independently. It also allows the correctness property of the `scan` operation to be used as an “induction” hypothesis in verifying the `search` operation.

```
scan(i, p)(queen): RECURSIVE (qlift?(good_extension?(i, queen, p)))
=
(IF i = N THEN IF p(queen) THEN up(queen) ELSE bottom ENDIF
 ELSE search(i, queen, p, 0, scan(i+1, p))
 ENDIF)
MEASURE N - i
```

The `findboard` operation then returns a lifted board where the result is either `bottom` or of the form `up(X)`, where `X` is a board satisfying `p`. The `goodqueen?` predicate is used to check if a particular board is a valid `N`-Queens configuration so that the actual program is `findboard(goodqueen?)`.

```
findboard(p): (qlift?(p)) =
  scan(0, p)(LAMBDA (i: below(N)): 0)

goodqueen?(queen): bool =
  (FORALL (i, j: below(N)): i /= j IMPLIES
    (queen(i) /= queen(j) AND
     (i - j /= queen(i) - queen(j)) AND
     (j - i /= queen(i) - queen(j))))
```

The theory generates twenty TCCs of which four are subsumed by other TCCs, twelve are proved by the default strategies, and the remaining seven require proofs with a small number of interactions to make type constraints explicit and to manually instantiate quantifiers.

7 Problem 5: Amortized Queue Implementation

The fifth problem in VSComp 2010 is to implement an applicative queue with a good amortized complexity using linked lists. The queue structure must support the operations of enqueueing an element at the rear of the list, dequeueing an element from the front of the list, and returning the first element of the list. The queue itself must be implemented with two fields, the `head` and the `tail`, that are both linked lists. The implementation must maintain an invariant that the tail list must not be any longer than the head list, so that the first element is always on the head list. The implementation of the operations of enqueueing, dequeueing, and computing the first element of the queue must be shown to respect their contracts relative to an abstract version of the queue.

The theory `appqueue` contains the implementation of applicative queues. The `appqueue` type (with the name overloaded) is defined as a record with two list fields: `head` and `tail`, and two numeric fields `hdl` and `tll` containing the lengths of the head and tail lists, respectively. The constraints on the `hdl` and `tll` fields is captured by the dependent typing of the `appqueue` record type. The `hdl` field is constrained to contain the length of the list in the `head` field, and the `tll` field is constrained to contain the length of the `tail` field, which must be no more than the contents of the `hdl` field.

```

appqueue  [T: TYPE ]
          : THEORY

BEGIN

A, B, C: VAR list[T]
x, y, z: VAR T

appqueue: TYPE = [# head: list[T], tail: list[T],
                  hdl: {i: nat | i = length(head)},
                  tll: {i: nat | i = length(tail) AND i <= hdl} #]

Aq, Bq, Cq: VAR appqueue

.
.
.
END appqueue

```

We then define the contents of a queue Aq by the list given by contents (Aq) which is defined to be the result of concatenating the head field with the reversal of the tail field of Aq . We also define predicates `empty?` and `nonempty?` to indicate when a queue is empty or nonempty, respectively. By the invariant on `appqueue`, it is enough to check that the `hdl` field is 0 to determine if a queue is empty.

```

contents(Aq): list[T] = append(Aq`head, reverse(Aq`tail))

empty?(Aq): bool = (Aq`hdl = 0)

nonempty?(Aq): bool = NOT empty?(Aq)

```

The function `enq` captures the operation of enqueueing an element x to the back of queue Aq . If the `hdl` and `tll` fields of Aq are equal, then in order to preserve the invariant, the contents of `tail` along with the new element x are (reversed and) appended to the `head` list, the `tail` field is emptied, and the lengths are adjusted accordingly. Otherwise, the new element is added to the `tail` list and `tll` is incremented by one.

```

enq(x, Aq): appqueue =
  (IF Aq`hdl = Aq`tll
   THEN Aq WITH ['head := append(Aq`head, reverse(cons(x, Aq`tail))),
                'tail := null,
                'hdl := Aq`hdl + Aq`tll + 1,
                'tll := 0]
   ELSE Aq WITH ['tail := cons(x, Aq`tail),
                'tll := Aq`tll + 1]
  ENDIF)

```

The `deq` operation dequeues an element from a nonempty queue Aq by dropping the first element of the `head` list of the queue representation. As with `enq`, in order to preserve the invariant when the `head` and `tail` lists are of equal length, the contents of the `tail` list are reversed and appended to the modified `head` list. The `first` operation is defined to compute the first element of a nonempty queue.

```

deq((Aq | nonempty?(Aq)): appqueue =
  (IF Aq'hdl = Aq'tll
   THEN Aq WITH ['head := append(cdr(Aq'head), reverse(Aq'tail)),
                 'tail := null,
                 'hdl := Aq'hdl + Aq'tll - 1,
                 'tll := 0]
   ELSE Aq WITH ['head := cdr(Aq'head),
                 'hdl := Aq'hdl - 1]
   ENDIF)

first(Aq | nonempty?(Aq)): T =
  car(Aq'head)

```

The contractual properties of the three operations are stated in terms of the abstract queue given by the `contents` operation in the theorems `contents_enq`, `contents_deq`, and `contents_first` below. These operations are proved by a strategy similar to the one used with Problem 3 using two invocations of `grind` separated by the introduction of type predicates for the subexpressions, but with the additional invocation of rewrite rules for some of the list operations. The proof generates eleven TCCs all of which are discharged by the above proof strategy.

```

contents_enq: LEMMA
  contents(enq(x, Aq)) = append(contents(Aq), (cons(x, null)))

contents_deq: LEMMA
  nonempty?(Aq) IMPLIES
  contents(deq(Aq)) = cdr(contents(Aq))

contents_first: LEMMA
  nonempty?(Aq) IMPLIES first(Aq) = car(contents(Aq))

```

8 Discussion

The solutions that we have presented for the five VSComp problems can be used to discuss a number of different aspects of verification technology.

The Computational Sapir–Whorf Hypothesis. The Sapir–Whorf hypothesis states that language influences thought, and that the presence or absence of certain features in a language influence thought, or more strongly, determine thought. In linguistics, this is a controversial proposition, but in formal language, the language constraints certainly affect the felicity with which concepts can be expressed and manipulated. The language and inference mechanism in PVS are designed to operate in tandem. The type system allows arbitrary formulas to be used as constraints, so that type correctness requires proof obligations to be discharged. This makes the type system expressive, but verifying type correctness is undecidable since any formula could be generated as a proof obligation. We have also seen how the specification of an operation can be captured by its type. The proof obligations can typically be discharged automatically by a default proof strategy, or proved using a small number of interactions. PVS also contains a number of other features for automatically propagating type information and managing proof obligations. In the first four problems, type constraints on recursive operations were used to verify their correctness by implicit induction. In the fifth problem, the datatype invariants on the queue are encoded using subtypes. In PVS, *typing judgements* are used to

propagate type constraints through expressions and operations, to infer, for example, that the product of two even numbers is an even number. The PVS typechecker rapidly catches a large number of common specification errors so that definitions can be efficiently debugged as they are being written.

Automated versus Interactive Proof Construction. PVS admits both automated proof steps both through primitive proof steps as well as defined proof strategies. The primitive proof steps include those that invoke SAT/SMT solvers, rewriters, and model checkers. Defined proof strategies such as `grind` and `induct-and-simplify` offer robust automation for discharging proof obligations. These automated proof steps can also be used within an interactive development. The ability to invoke powerful automation within an interactive proof development is quite important. PVS is mainly used for exploring and experimenting with specifications and proofs. For such experimentation, it is useful to be able to vary the level of automation. For example, if a proof does not succumb to an automated strategy, then the user needs to either examine the resulting subgoals, or investigate why the strategy failed. The latter outcome could be because of incorrect or poorly stated definitions and conjectures. The interaction has to support such exploration. With PVS, there has been a concerted effort to develop interactive proof steps that approximate the level of detail in an informal proof. The proof steps are designed to either succeed or generate sensible subgoals. When a pattern of proof emerges, this pattern can be embedded within a strategy for future use. This way proofs that have been found with careful interactive exploration can be redone using proof strategies that are more robust to changes in the definitions and theorems. User-defined proof strategies are useful in building custom-built inference patterns for specific classes of problems by using existing proof strategies as building blocks. Since proofs using strategies can be rerun using only the primitive inference steps, only the latter steps need to be trusted.

Programs versus Specifications. The PVS specification language is based on higher-order logic to support the development of proofs. This language is designed to be convenient for expressing mathematical concepts, and PVS features well-developed formal libraries of mathematics and program semantics. Programs can be represented in a variety of ways within the PVS logic. The most direct way is to use a fragment of the higher-order logic as a functional programming language. This fragment contains nearly all of the logic except equality on higher types, which also includes quantification. Predicate subtypes ensure that well-typed programs can never fail, by ruling out division by zero, numeric overflows and underflows, out-of-bounds array indices, and other sources execution failure. Structural subtypes exploit the polymorphism of the update operation for records, tuples, and functions, in defining operations work uniformly over a subhierarchy of types. Update analysis is used to detect those updates that can be safely executed in-place, i.e., destructively [Sha02]. All of the programs in five exercises are executable as defined, but the update analysis is too weak to detect the possibility of executing the `invert` operation destructively. The N-Queens solution is also not executed destructively since the update operations occur in a higher-order setting within `scan`. Some of the VSComp solutions directly verified imperative programs for these problems with the added complexity of maintaining frame conditions.

Level of Effort. Obviously, the solutions described here were not obtained under competition conditions with a given time limit. We did not exactly measure the amount of time taken to obtain these solutions. Of these, only the N-Queens problem can be considered nontrivial. We estimate that the other problems could be solved by an expert PVS user in about half an hour to an hour. The N-Queens problem required some creativity in inventing the `qlift?` predicate and the corresponding subtype and in constructing the `good_extension?` predicate for capturing the invariant for the `search` and `scan` operations. The N-Queens exercise took less than two hours of effort with PVS, where a third of the time was spent in designing the approach, another third in fixing minor type errors and other bugs in the definition, and the final third in completing the correctness proof.

In all of the problems, the type system is very helpful in quickly identifying both trivial and deep programming errors mostly through the generation of unprovable TCCs. As programmers, we often wish we could do our programming in PVS so that we can specify preconditions, post-conditions, invariants, and proofs hand-in-hand with the definitions. However, PVS still needs many more features before it can be seen as a facile program development framework.

9 Conclusions

The first Verified Software Competition is a step toward measuring the progress in computer-aided software verification. The organizers of this competition have encouraged the submission of solutions to these problems by groups that did not participate in actual event. We have described one way of approaching these problems using the combination of expressiveness, automation, and interaction in PVS. The exercise has highlighted both the strengths and weaknesses of PVS. In particular, we have seen the need for better simplification with nonlinear arithmetic constraints, better quantifier instantiation, and more aggressive exploitation of type information.

The report [KMS⁺11] on the first Verified Software Competition summarizes the solutions presented by eleven groups. None of the groups finished all five problems during the competition, and none of the participants successfully completed the N-Queens problem. Subsequent to the competition, six groups completed all five problems. The solutions to these problems displayed a range of techniques from heavily interactive proof to automation based on SMT solving and first-order theorem proving. Nearly all of the participants were expert users of their chosen verification tools. It is clear that we need a better understanding of the balance between interaction and automation that can be employed most productively by expert and non-expert users.

Carolyn Talcott has been a mentor to both authors and her influence on this work is considerable. She pioneered operational approaches to program correctness, and the implicit induction techniques used here can be seen as an instance of this approach. PVS was developed to address problems of partial and total correctness developed in her book on *Lisp: Programming and Proving* [MT78] with John McCarthy. PVS itself is a Lisp program built using the principles of functional programming pioneered in the McCarthy/Talcott book.

We thank the anonymous referees for their helpful comments and Yannick Moy of AdaCore for his insightful feedback on an earlier draft.

References

- [BTSS09] Barrett, C., Tinelli, C., Sebastiani, R., Seshia, S.: Satisfiability modulo theories. IOS Press (2009)
- [dMDS07] de Moura, L., Dutertre, B., Shankar, N.: A tutorial on satisfiability modulo theories. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 20–36. Springer, Heidelberg (2007)
- [GM93] Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993), HOL home page <http://www.cl.cam.ac.uk/Research/HVG/HOL/>
- [Hoa03] Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. *Journal of the ACM* 50(1), 63–69 (2003)
- [KMS⁺11] Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholtz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011), materials www.vscmp.org
- [MT78] McCarthy, J., Talcott, C.: Lisp: Programming and proving (1978)
- [Muñ03] Muñoz, C.: Rapid Prototyping in PVS. National Institute of Aerospace, Hampton, VA (2003), <http://research.nianet.org/~munoz/PVSio/>
- [NPW02] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002), Isabelle home page <http://isabelle.in.tum.de/>
- [ORSvH95] Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering* 21(2), 107–125 (1995), PVS home page <http://pvs.csl.sri.com>
- [OS01] Owre, S., Shankar, N.: Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA (April 2001)
- [PvE99] Plasmeijer, R., van Eekelen, M.: Functional programming: Keep it CLEAN: A unique approach to functional programming. *ACM SIGPLAN Notices* 34(6), 23–31 (1999)
- [Sha02] Shankar, N.: Static analysis for safe destructive updates in a functional language. In: Pettorossi, A. (ed.) LOPSTR 2001. LNCS, vol. 2372, pp. 1–24. Springer, Heidelberg (2002)
- [SN08] Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
- [Ste90] Steele Jr., G.L.: Common Lisp: The Language, 2nd edn. Digital Press, Bedford (1990)

Towards a Maude Formal Environment

Francisco Durán¹, Camilo Rocha², and José María Álvarez¹

¹ Universidad de Málaga, Spain

² University of Illinois at Urbana-Champaign, IL, USA

Abstract. Maude is a declarative and reflective language based on rewriting logic in which computation corresponds to efficient deduction by rewriting. Because of its reflective capabilities, Maude has been useful as a *metatool* in the development of formal analysis tools for checking specific properties of Maude specifications. This includes tools for checking termination, confluence, and inductive properties of rewrite theories. Nevertheless, most of these tools have been designed to work in isolation, making it difficult, for instance, to exchange data between them and inconvenient to switch between their environments. This paper presents the Maude Formal Environment (MFE), an executable formal specification in Maude within which a user can interact with tools to mechanically verify properties of Maude specifications. One important aspect of this work is that the MFE has been designed to be easily extended with tools having highly heterogeneous designs whilst creating synergy among them. As a proof of concept, we report on the integration of five commonly used formal analysis tools for Maude specifications into MFE and illustrate their interoperability with an example.

1 Introduction

There is a great deal of interest today in developing multipurpose environments that combine declarative programming with specification languages and useful formal analysis tools (see, e.g., [23,16,18,1,2]). Maude [3,4] is a reflective declarative language and system based on rewriting logic in which computation corresponds to efficient deduction by rewriting. Maude has been successfully used as a *metatool* in the creation of tools for verifying properties of Maude specifications [5,6]. Nevertheless, these tools work in isolation, making it inconvenient to switch between their environments and difficult to exchange data between them. In this sense, despite its title, previous work presented in [6] does not conform to the notion of formal tool environment discussed here. In response to these limitations, we present the Maude Formal Environment (MFE), an executable and highly extensible software infrastructure within which a user can interact with several tools to mechanically verify properties of Maude specifications. In MFE, tool interoperability allows for discharging proof obligations of different nature without switching between different tool environments. The integration of different tools inside MFE's common environment presents the user with a consistent user interface, a mechanism to keep track of pending proof obligations, and allows the execution of several instances of each tool, among other features.

The purpose of MFE is to support interactive formal analysis of Maude specifications, and therefore the integration of tools within MFE revolves around Maude modules. MFE is naturally modeled in Maude as an object-based system in which the tools

are objects and their communication mechanism is message passing. User interaction is available through Full Maude [11,4], an extension of Maude that has become a common base on top of which tools can be built, offering a modular design for easily integrating other tools written in Maude (see, e.g., [15] for a guide on how to extend Full Maude).

One of the most interesting challenges was to make the MFE design highly extensible and amenable to tool interoperability. In MFE, there is no constraint on how each tool should model its particular domain or how it maintains its internal state. We implemented an object-based version of the *model-view-controller* pattern to separate the modeling of the domain for each tool, the presentation of information, and the actions based on user input into separate objects. This pattern isolates the application logic for the user from the user input and presentation, permitting independent development, testing and maintenance of each. We also followed good object-oriented design practices that kept the cohesion high and the coupling low among objects. In our experience, these good design practices proved key for the integration of tools in MFE.

We report here on the integration of five tools with highly different designs and implementations in MFE. Namely, the Maude Termination Tool (MTT) [8], the Church-Rosser Checker (CRC) [13,14], the Coherence Checker (ChC) [12,14], the Sufficient Completeness Checker (SCC) [20,21], and Maude's Inductive Theorem Prover (ITP) [7,19]. Despite their heterogeneity and isolated conception, these tools were integrated in MFE with very few code alterations, many of these due to renaming of sorts and operators. For tools which depend on external utilities not directly available from Maude such as MTT and SCC, we have extended the Maude system to a non-official distribution with *built-in* operators associated with appropriate C++ code that interacts with the external tools. A similar extension was already performed for SCC [19].

MFE, with these five tools, as well as some examples and some preliminary documentation, is available at <http://maude.lcc.uma.es/MFE>.

Outline of the Paper. Section 2 gives a summarized account of Maude's object-based programming and support for user interoperability. Section 3 discusses the main design aspects of MFE. Section 4 describes the tools available from the current version of MFE. Section 5 illustrates how to extend MFE with a concrete tool and Section 6 presents a case study in which a user interacts with several tools within MFE. Finally, Section 7 presents related and future work, and some concluding remarks.

2 Object-Based Programming and User Interfaces in Maude

We assume that the reader is familiar with the basics of rewriting logic and Maude, and refer to [4] for an introduction to these.

Maude can be used not only to define domain-specific languages or tools, but also to build environments for such languages and tools. In such applications, the predefined LOOP-MODE module can be used to handle the input/output and to maintain the persistent state of the language environment or tool. This section explains some basic background on how object-based systems, which naturally model distributed systems in Maude, and the LOOP-MODE module are used to define MFE's interactive infrastructure as an extension of Full Maude.

Object-based Programming. Maude supports the modeling of object-based systems in the predefined `CONFIGURATION` module that declares sorts representing the essential concepts of object, message, and configuration, along with notation for object syntax that serves as a common language for specific object-based systems. The basic sorts defined in `CONFIGURATION` are `Object`, `Msg`, and `Configuration`. A configuration is a multiset of objects and messages that represent a possible system state. Configurations are formed by multiset union, represented by empty syntax `__`, starting from singleton objects and messages. The empty configuration is represented by the constant `none`.

```
sort Configuration .
subsorts Object Message < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
  [ctor assoc comm id: none] .
```

In general, a rewrite rule for an object-based system has the form

```
cr1 [r] :
  Obj1 ... Objm Msg1 ... Msgn
=> Obj'1 ... Obj'j Objm+1 ... Objp Msg'1 ... Msg'q
if Cond .
```

where objects $Obj'_1 \dots Obj'_j$ are updated versions of objects $Obj_1 \dots Obj_m$, for $j \leq m$, $Obj_{m+1} \dots Obj_p$ are newly created objects, $Msg_1 \dots Msg_n$ are consumed messages, and $Msg'_1 \dots Msg'_q$ are new messages.

The user is free to define any object or message syntax that is convenient. However, for uniformity in identifying objects and message receivers, the adopted convention is that *the first argument of an object constructor should be its identifier*, and *the first argument of a message constructor should be the identifier of its addressee*. Module `CONFIGURATION` provides an object syntax that serves as a common notation that can be used by developers of object-based system specifications, as is the case in Full Maude. This module introduces sorts `Oid` for object identifiers, `Cid` for class identifiers, `Attribute` for named elements of an object's state, and `AttributeSet` for multisets of attributes. In this syntax, objects have the general form $\langle O : C \mid a_1 : v_1, \dots, a_n : v_n \rangle$ where O is an object identifier, C is a class identifier, and the $a_i : v_i$ are pairs of an attribute name a_i and a value v_i , for $1 \leq i \leq n$.

Full Maude provides convenient notation for object-oriented modules in which classes are declared with the syntax

```
class C | a1 : S1, ..., an : Sn .
```

where C is the name of the class, the a_i are attribute identifiers, and the S_i are the sorts of the corresponding attributes. Class inheritance is directly supported by Maude's ordered type structure. A subclass declaration

```
subclass C < C' .
```

is just a particular case of a subsort declaration $C < C'$. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses, together with the

newly defined attributes, messages, and rules of the subclass, characterize the structure and behavior of the objects in the subclass. In what follows, we use this convenient object-oriented notation for defining classes. See [4] for further details on this notation and on the transformation of object-oriented modules into system modules.

User Interfaces. Module `LOOP-MODE` specifies in Maude a general input/output facility by a read-eval-print loop using object-based concepts. A *loop object* is a term of the form `[In,St,Out]` where `In` is an input stream, `Out` is an output stream, and `St` is its state. One can think of the input and output events as implicit *rewrites* that transfer the input and output data between two objects, namely the loop object and the user (or terminal) object.

Loop objects are constructed with the operator

```
op [_,_,_] : QidList State QidList -> System [...] .
```

Besides having input and output streams, terms of sort `System` give a generic way for maintaining a state in its second component. In fact, sort `State` in `LOOP-MODE` does not have any constructors, giving complete flexibility for defining the terms we want to have for representing the state of the loop. In MFE, we represent state terms as configurations of objects and messages, by declaring sort `Configuration` as subsort of `State`.

Rewrite rules define the interaction of the state with the loop and the changes produced in the state by the actions requested by the user. In order to generate in Maude an *interface* for interacting with an application, the language for interaction needs to be defined by a data type for commands and other constructs. In this way, a rule can detect when a valid request has been introduced by the user, and if the state of the system allows it, passes it as the next action to be attempted. For the other direction of interaction, a rule detects when the state has a response to be output and, in that case, it places it in the output component of the loop object.

Full Maude. In Full Maude, the persistent state of the read-eval-print loop provided by module `LOOP-MODE` is given by a single object of class `DatabaseClass`. Objects of this class have: an attribute `db` of sort `Database` to keep the actual database where all the modules entered to the system are stored, an attribute `default` denoting the name of the current default module, and attributes `input` and `output` that simplify the communication between the read-eval-print loop and the database object. Using the above syntactic sugar for object-oriented modules, we can declare such a class as:

```
class DatabaseClass |
  db : Database, default : ModName, input : TermList, output : QidList .
```

Inputs from the user into Full Maude are parsed using the built-in `metaParse` function. For such parsing, Full Maude uses the `FULL-MAUDE-SIGN` module, in which we can find the declarations so that any valid input can be parsed. In particular, we find in these modules, among others, sorts `@Module@`, `@ModExp@`, and `@Command@`, of modules, module expressions, and commands, respectively, and syntax declarations such as:

```
op select_ . : @ModExp@ -> @Command@ .
op show module_ . : @ModExp@ -> @Command@ .
```

```
op mod_is_endm : @Interface@ @SDeclList@ -> @Module@ .
op omod_is_endom : @Interface@ @ODeclList@ -> @Module@ .
```

for commands `select` and `show module`, and for system and object-oriented modules.

The behavior of Full Maude upon the reception of new inputs from the user is specified by rewrite rules. For the different commands, different actions are accomplished.

3 The Design of MFE

The object-oriented model of MFE consists of three classes: the class `Proof` of proof objects that keep the state of specific proof requests, the class `Tool` of tool objects that manage proof objects, and a class `Controller` that inherits from the Full Maude's `DatabaseClass` and provides a centralized entry point for handling requests to the formal environment.

The `Controller` object orchestrates the behavior of the environment with the user and of the environment with its tools. The user interacts with the environment via commands that are encapsulated as messages in the object configuration. Each tool object and the controller object have a module defining the grammar of the commands it can handle. The controller handles any command it can parse; since this object extends Full Maude, it handles its own commands and Full Maude ones. If the controller receives a command it cannot parse, it will delegate it to the current *active* tool. If the active tool can parse the delegated command, then it notifies the controller and handles the command. Otherwise, it notifies the failure to the controller that in turn will notify the failure to the requester.

Classes `Proof` and `Tool` define some basic functionality for tools and, as we will see in Section 5 for a sample tool, are provided to simplify the task of incorporating new tools to the environment. However, tools can be added to the environment by defining the expected interaction with the controller object without using classes `Proof` and `Tool`. This was the case, for example, with the ITP tool that does not use in MFE the infrastructure provided by classes `Proof` and `Tool`.

In the following subsections we describe in more detail the object-based model of MFE and its interaction mechanism.

3.1 Proof Objects

Proof objects maintain the state of specific proof requests to a tool object. Every proof object maintains in its state the information of the module associated to the proof obligation and a set of object identifiers corresponding to the objects submitting the proof obligation.

```
class Proof | module : Module, requester : Set{Oid} .
```

The concept of a proof object representing the state of a proof requirement, is key for enabling tools in MFE with support for multiple proof requirements. Namely, handling a “new proof” command corresponds to instantiating a proof object with the appropriate attribute data. Commands that incrementally modify the status of a proof obligation

result in updates to the attributes of the proof object. For example, a CRC proof object will keep track of confluence and/or sort-decreasingness checks, and will be updated every time a new proof obligation is discharged; when all proof obligations have been proved, it will realize the check's completion and will inform all its requesters.

A subclass of `Proof` may be defined for each tool in the environment, adding the additional attributes and behavior required by the specific tool. Proof obligation objects, for instance, can be extended with additional attributes for keeping track of dependencies of subgoals that should be handled by other tools, timeouts, number of attempts, or any other required information.

3.2 Tool Objects

A tool object is responsible for maintaining the life cycle of its proof objects. When a tool object receives a request for a new proof, it tries to create a new proof object for it and, if successful, it sets the new proof object as active so that any command from the user or message from other tools in the environment are forwarded to it. There is exactly one tool object for each tool in the formal environment.

Every tool object has an attribute `grammar` that defines the grammar of user commands the tool can handle. Tools may rely on other tools, hence a tool object has attribute `tools` with a map from tool names to the object identifiers of the corresponding tools in the environment. If proof obligations are due, this attribute will be used to submit them to the appropriate tools. The references of proof objects associated to a tool object are maintained with a map from module names to object identifiers in attribute `reg`. In MFE, a tool may perform several analyses on a module, so that the information of such analyses is kept in the attributes of the corresponding proof object. A tool object also keeps in attribute `current` a reference to one of its proof objects, if any, which is referred by the tool as its *active* proof obligation.

```
class Tool | grammar : Module,
            tools : Map{ToolName, Oid},
            reg : Map{ModuleName, Oid},
            current : Oid,
            index : Nat .
```

Integration and interoperability of tools within MFE revolves around modules, and therefore, typically, the “new goal” commands have a module expression as parameter, although in general commands may have other parameters. For example, a check of the Church-Rosser property would take just the module to be checked as parameter. However, one can perform checks of coherence in the coherence checker with the option of checking coherence or ground coherence via an additional parameter. The decision of whether a new proof object is generated or not for a module when attempting different kinds of checks is up to the tool developer.

Despite this flexibility, in the general approach a tool object will create a new proof object for a module M whenever there is no record of M being previously handled by the tool, namely, if name of module M is not in the domain of the `reg` attribute. More precisely, when a proof is requested, the `reg` attribute is checked: if there is a proof for a module with such a name, then the module itself is compared with the current

one, to make sure that the module has not changed. In case there is some parameter, as for example an alternative transformation for a termination proof, if the proof has not succeeded before then the check is attempted: the existing proof object is replaced with the one corresponding to the new proof.

The different tools may perform tool specific checks on a parameter module M and if these checks succeed, then a new proof object is instantiated with a unique object identifier, with M as associated module (in attribute `module`), and the corresponding reference of the requesting object (in attribute `requester`). The remaining attributes of the proof object are set according to the purpose of the tool.

See Section 5 for an example of this behavior in the case of SCC.

3.3 The Controller Object

The task of the controller object is twofold: it provides a centralized entry point for handling user requests and it manages the tools that are available in the environment.

The `Controller` class inherits from Full Maude's `DatabaseClass`, and in addition to its module database and all the functionality for handling all Full Maude commands, the controller object stores information on the tools available in MFE. This is achieved by using the attribute `tools` that is a map from tool names into object identifiers. In the attribute `current-tool` the controller object maintains a reference to the active tool:

```
class Controller | current-tool : Oid, tools : Map{ToolName, Oid} .
subclass Controller < DatabaseClass .
```

The controller object is *the singleton* instance `mfe` of class `Controller`. To handle a command, the object `mfe` first tries using its grammar (which extends that of Full Maude). If the command can be parsed with its grammar, then `mfe` handles the request. Otherwise, it delegates the command to the active tool and waits for an answer. The user can select the active tool via a “select” command. The answer to the delegated request can be either affirmative or not, meaning that the tool can parse the command and will handle it, or that the given command does not conform to the grammar and therefore cannot be handled by the tool. Because of the way user and tool interaction has been designed and implemented in MFE, there is no need to enforce a policy of uniqueness of commands among its tools: if two tools share a command syntax, then such command will be handled by the controller object or else by the active tool. This simplifies the integration of existing tools, because most of their implementations can directly be used and because all proof scripts available are still usable after adding the appropriate selection and submission commands.

The `Controller` class adds the following commands to those available in Full Maude:

```
(select tool <tool-name> .) sets the tool <tool-name> as active tool.
(MFE help .) shows information on the commands available.
(show global state .) shows the state of the framework.
```

To illustrate the way in which the behavior of the controller object works, we present the `select-tool` rewrite rule that implements tool selection in MFE for the controller:

```

var X@Controller : Controller .   var Ct : Constant .
var TS : Map{ToolName, Oid} .   var QIL : QidList .
var Atts : AttributeSet .       var O : Oid .

crl [select-tool] :
  < mfe : X@Controller | input : ('select'tool_.[Ct]),
    output : QIL,
    current-tool : O,
    tools : TOOLS,
    Atts >
=> < mfe : X@Controller | input : nilTermList,
    output : QIL 'The getName(Ct) 'has 'been 'set 'as 'active 'tool.,
    current-tool : TOOLS[qid2tool(getName(Ct))],
    tools : TOOLS,
    Atts >
  if TOOLS[qid2tool(getName(Ct))] /= null .

```

When the result of parsing a “select tool” command in the grammar of the controller is placed in the `input` attribute of the `mfe` object, and it corresponds to a tool in the environment (see the condition in rule `select-tool`), then such a tool is set as the active one. Functions `getName` and `qid2tool` return the name of a given constant and transforms a quoted identifier into a tool name, respectively. A second rule (omitted here) handles the case in which the argument of the `select` command does not correspond to a tool in the environment; this rule reports on the situation by creating an error message.

3.4 User Interaction and Tool Interoperability

In the object configuration of MFE, user interaction is achieved via commands and tool interoperability via messages. Upon successful parsing, commands are converted into messages. With this approach, requests from users and from tools are handled in a uniform manner by just distinguishing the requester.

MFE internal messages identify their contents with “to-from” information and name the different operations offered by the tools and their answers. Using the following syntax for messages, each tool defines its corresponding message bodies.

```

sort MFEMsgBody .
op to_from_:_ : Oid Oid MFEMsgBody -> Msg [ctor] .

```

If a user command parses in the controller’s grammar, then the controller handles the command. If it fails, then the input is submitted to the active tool. The tool is expected to return an “input parsed” message indicating whether or not it is able to parse the command or not. Rewrite rule `input`, below, defines the behavior of a tool object that is able to parse the input command. Observe that when a tool object can parse a command, it sends two messages. Namely, it creates a copy of the original message but with a parsed version of the input command and it sends the requester an output message indicating that the input can be parsed.

```

var X@Tool : Tool .
var Atts : AttributeSet .
var G : Module .

vars O O' : Oid .
var QIL : QidList .

crl [input] :
  < O : X@Tool | grammar : G, Atts >
  (to O from O' : input(QIL))
=> < O : X@Tool | grammar : G, Atts >
  (to O from O' : getTerm(metaParse(G, QIL, '@Input@)))
  (to O' from O : input-parsed(QIL, true))
if RP:ResultPair := metaParse(G, QIL, '@Input@) .

```

If the tool cannot parse the input, then another rule (omitted here) sends the requester an input-parsed message with its second argument set to `false`. When the controller receives the input-parsed message, with `true` or `false`, it proceeds either by letting the corresponding tool resolve the command or by displaying an error message.

4 Tools in MFE

Five formal analysis tools are available in the current release of MFE. The Maude Termination Tool (MTT) can be used to prove termination of functional and system modules, the Church-Rosser Checker (CRC) can be used to check the Church-Rosser property of functional modules, the Coherence Checker (ChC) can be used to check the coherence of system modules, the Inductive Theorem Prover (ITP) can be used to verify inductive properties of functional modules, and the Sufficient Completeness Checker (SCC) can be used to check completeness of functional modules and deadlock freedom of system modules. One important aspect in the integration task is the interaction complexity due to the nontrivial dependencies among tools. Figure 1 depicts the tool-dependency graph for these five tools.

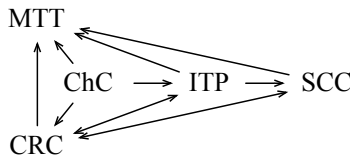


Fig. 1. Tool-dependency graph in MFE

In the following paragraphs we summarize the main features and dependencies of the five tools available in MFE. For further details on these tools, including user manuals, restrictions, and examples, we refer the reader to the given references and web sites.

4.1 The Maude Termination Tool

Maude has expressive features including advanced typing constructs with sorts, sub-sorts, kinds, and memberships; matching modulo axioms; evaluation strategies for both equations and rewrite rules; and very general conditional equations and rewrite rules. Proving termination of programs having such features is nontrivial. Furthermore, some of these features are not supported by standard termination methods and tools. Yet, the use of such features may be essential to ensure termination. MTT uses several non-termination preserving theory transformations [9,10] which are applied in a kind of pipeline to a module M , obtaining a module M' in such a way that a proof termination of M' witnesses the termination of M . For instance, by sequentially using four theory transformations, a conditional order-sorted context-sensitive system module can be transformed into an unconditional unsorted context-sensitive term rewrite system, which can be handled by the back-end tools.

The Maude Termination Tool (MTT) is a tool that checks the termination of (possibly conditional) order-sorted Maude functional or system modules. The current implementation takes a module as input and tries to prove its termination by applying theory transformations and then invoking back-end termination tools, such as MU-TERM [22] and AProVE [17], that can prove termination of (variants of) rewriting. For the current version of MFE, a new “hook” to a C++ library was included in non-official distribution of Core Maude for invoking these back-end termination tools. MTT is the only tool in the MFE that does not depend on any other tool in the environment.

A stand-alone version of MTT, with a graphical user interface, is available from <http://www.lcc.uma.es/~duran/MTT/>.

4.2 The Church-Rosser Checker

The Church-Rosser Checker (CRC) checks the Church-Rosser property of Maude functional (possibly conditional) order-sorted modules. For order-sorted modules, being Church-Rosser and terminating means not only confluence, but also a sort-decreasingness property: each normal form has the least possible sort among those of all equivalent terms. CRC depends on MTT for checking termination assumptions and on ITP for inductive theorem proving (see Section 4.5). Some of the proof obligations are not currently handled by any tool. We will see in Section 6 that although they are submitted to ITP, these proofs may be “trusted” by the user.

CRC can be used to check equational specifications $(\Sigma, E \cup Ax)$ with an initial semantics that have already been proved terminating and need to be checked (ground) Church-Rosser. The tool performs both a local confluence check by computing all (*conditional*) *critical pairs* of the equations E modulo the structural axioms Ax and a sort-decreasingness test in the form of membership assertions for each equation in E modulo Ax . If the (conditional) critical pairs or the sort decreasingness tests cannot be discharged by CRC, proof obligations are displayed as a guide to the user. In MFE, CRC can submit these proof obligations to ITP for inductive reasoning.

CRC, with its documentation and some examples, is available from <http://maude.lcc.uma.es/CRChC/>.

4.3 The Coherence Checker

(*Ground*) coherence allows reducing the problem of computing rewrites of the form $[t]_{E \cup Ax} \rightarrow [t']_{E \cup Ax}$ that in general is undecidable, to the much simpler and decidable problem of computing rewrites of the form $[t]_{Ax} \rightarrow [t']_{Ax}$, for t and t' any (ground) Σ -terms. Intuitively, coherence means that rewriting with R modulo $E \cup Ax$ can be achieved by adopting the strategy of first simplifying to canonical form with E modulo Ax and then applying a rule in R modulo Ax .

The Coherence Checker (ChC) is a tool that provides a (ground) coherence decision procedure for order-sorted system modules $\mathcal{R} = (\Sigma, E \cup Ax, R)$. The tool calculates the set of critical pairs between the equations E and the rewrite rules R modulo the structural axioms Ax , whose equational validity guarantees the (ground) coherence of \mathcal{R} . ChC depends on MTT for termination assumptions and on CRC for the equations E being Church-Rosser. Since this property is inductive, in some cases ITP can be used to prove some proof obligations.

ChC, with its documentation and some examples, is available from <http://maude.lcc.uma.es/CRChC/>.

4.4 The Sufficient Completeness Checker

Maude Sufficient Completeness Checker (SCC) is a tree automata based tool for checking *sufficient completeness* and *deadlock freedom* of terminating and sort-decreasing Maude modules. Both sufficient completeness and deadlock freedom are relative to *constructor* subsignatures. For $\mathcal{R} = (\Sigma, E, R)$, a signature pair (Y, Ω) , with $Y \subseteq \Omega \subseteq \Sigma$, is a pair of constructors for \mathcal{R} if for each sort s in Σ and each ground Σ -term with sort s , (i) there is a Ω -ground term u with sort s satisfying the equality $t = u$ in E , and (ii) there is a ground Y -term v with sort s satisfying the sequent $t \rightarrow v$ in R . Intuitively, sufficient completeness is the property that every operation in a specification is equationally defined for all inputs, and deadlock freedom is the property that every nondeterministic computation leads to a terminal state [24,19].

Sufficient completeness and deadlock freedom are important properties both for developers of specifications, to check that they have not missed a case in defining operations, and to inductive theorem provers, to check the soundness of a proposed induction scheme. In the case of equational specifications, SCC assumes the input specification is ground sort-decreasing and terminating; in the case of rewrite specifications, SCC assumes the input specification is ground sort-decreasing, terminating, Church-Rosser, and coherent.

The tool is designed for unparameterized, order-sorted, left-linear, and unconditional Maude specifications that are ground terminating and Church-Rosser. It is a decision procedure for this class of specifications when every associative symbol is also commutative. For associative symbols that are not commutative it uses machine learning techniques that work well in practice. If the specification is not sufficiently complete, SCC returns a counterexample to aid the user in identifying errors. The tool is not complete for specifications with non-linear or conditional axioms, but nevertheless has proven useful in identifying errors in such specifications. SCC accepts interactive commands

to check the sufficient completeness of a Maude module, and internally constructs a propositional tree automaton whose language is empty if and only if the Maude module is sufficiently complete. The emptiness check is performed by a C++ tree automata library named CETA.

The tool also supports several important completeness and freeness problems of context-sensitive specifications involving both equations and rewrite rules [24,19].

SCC is available from its website at <http://maude.cs.uiuc.edu/tools/scc>.

4.5 The Maude Inductive Theorem Prover

The Maude Inductive Theorem Prover (ITP) is an experimental proof assistant aiding in the task of proving inductive properties of the initial algebra associated to a membership equational theory. It is based on Membership Equational Logic, a good fit for reasoning inductively about functions and data structures involving partiality, subsorts, and conditions. ITP supports proofs by structural induction and complete induction, in which operations need not be completely specified. Goals are either conditional equations or conditional memberships, and inference steps are available through a series of user commands. ITP depends on MTT for checking termination assumptions, on SCC for checking sufficient completeness and freeness of equational constructors, and on CRC for checking the Church-Rosser property of the equations.

The ITP tool, documentation, and some examples are available from <http://maude.cs.uiuc.edu/tools/itp/>.

5 Extensibility by Example: The Integration of SCC

In this section we present a brief overview of the steps undergone to integrate SCC in MFE. In order to take as much advantage as possible of the infrastructure offered by MFE, some classes offered by MFE such as `Tool` and `Proof` are specialized with new attributes and behavior specific to SCC. In this way, SCC inherits the behavior defined already for these classes in MFE. Internal messages defining SCC's public interface are created and the rewrite rules defining SCC's behavior are updated so they fit into MFE's message passing interaction mechanism. Also, the controller object is modified to account for SCC in the object-based configuration of the formal environment.

5.1 SCC Proof Objects

An SCC proof object is an instance of class `SCCProof`, which is a subclass of `Proof`:

```
class SCCProof | sc-result : SCCResult,
                terminating : 3Bool,
                sort-dec : 3Bool,
                sound : Bool,
                complete : Bool,
                trusted : Bool .
subclass SCCProof < Proof .
```

Attribute `sc-result` registers the sufficient completeness result of sort `SCCResult`. If the emptiness test is successful, it holds the value `empty`; if it is unsuccessful, it holds a counter-example for sufficient completeness. A counter-example for sufficient completeness is a ground irreducible term that does not belong to the subsignature of constructors. Sort `Bool` is Maude’s predefined sort for Boolean values and operations, and sort `3Bool` is an extension of sort `Bool` with a third “undefined” value.

Ground termination and ground sort-decreasingness are assumed in SCC’s stand-alone version. An SCC proof object in MFE registers the ground termination and ground sort-decreasingness status of its associated module in the three-valued attributes `terminating` and `sort-dec`, respectively, so that it can be checked whether these assumptions hold (indicated by value `true`), do not hold (indicated by value `false`), or have not been submitted (indicated by value `maybe`). Thanks to the interoperability offered by MFE and as explained in Section 4, there are tools in MFE that can check for ground termination and ground sort-decreasingness of Maude specifications. Therefore, SCC proof objects can always submit these checks to the corresponding tools.

In some situations, some SCC’s requirements such as left-linearity of equations can be relaxed. For instance, if the emptiness check is successful when ignoring the non left-linear equations in a Maude module, then this module is sufficiently complete with all its equations. However, a counter-example to the sufficient completeness in the reduced module is not necessarily a sufficient completeness counter-example for the module with all its equations. SCC offers checks for these two properties and, in MFE, SCC proof object registers the information with Boolean attributes `sound` and `complete`. SCC proof objects define the Boolean valued attribute `trusted` for supporting a “trust” command for sufficient completeness proofs. This is useful for dealing with modules that are not supported by SCC or for sufficient completeness proofs that are obtained outside SCC.

5.2 The SCC Tool Object

The SCC tool object is an instance of class `SCCTool`, which is a subclass of MFE’s `Tool`. Class `SCCTool` does not declare any new attributes.

```
class SCCTool .
subclass SCCTool < Tool .
```

The functional module `SCC-SIGN` defines the grammar of the user commands supported by the SCC tool object. These are the commands available from SCC’s stand-alone version in addition to a new one for showing the state of `SCCProof` objects.

```
fmod SCC-SIGN is
  including FULL-MAUDE-SIGN .
  op scc_ . : @ModExp@ -> @Command@ .
  op submit . : -> @Command@ .
  op trust . : -> @Command@ .
  op show state . : -> @Command@ .
  op SCC help . : -> @Command@ .
  ...
endfm
```


The command (`scc MN.`) checks the sufficient completeness of module with name *MN*. The command (`submit .`) submits the termination and sort-decreasingness proof obligations to the corresponding tools in MFE for the active SCC proof object, if any. The command (`trust .`) trusts the sufficient completeness proof for the active SCC proof object, if any. The command (`show state .`) displays the state of each SCC proof object, and command (`SCC help .`) displays the help menu of Maude's SCC.

Two bodies for internal messages are defined. Namely, a message body for checking sufficient completeness and a message body for acknowledging that a module is sufficiently complete. The latter type of message is only created when the sufficient completeness check has been successful, and the termination and sort-decreasingness assumptions have been checked.

```
op check_sc_ : Module -> MFEMsgBody .
op module_is_sufficiently_complete : Module -> MFEMsgBody .
```

Observe that command (`scc _.`) and internal message body `check_sc_` refer to the same functionality offered by the SCC tool object, but with different inputs: the former takes a module expression as input, while the latter takes a module as input. Regardless of this typing difference, it is convenient to have exactly one entry point for commands referring to the same functionality. On the one hand it facilitates source code maintainability and debugging. On the other hand, it helps to avoid repetition of significant amounts of source code. To address this issue, the SCC tool object exclusively handles internal messages while it rewrites user commands in parsing messages to internal messages: it amounts to evaluating the module expression given by the user to a module in the database of modules and, if successful, to creating an internal message with the module and with the “from-to” data of the parsing message. There is an additional rule handling the case in which the module expression in the parsing message cannot be evaluated for a module, notifying the failure to the user. These rules correspond to updated versions of previously existing rules that handled user commands in SCC.

Rewrite rule `check-sc` below specifies the creation of an SCC proof object for checking the sufficient completeness of a module *M*. Here, function `processSCCheck` encapsulates the calls to functionality already available from SCC for the sufficient completeness check, and function `createSCCProof` encapsulates the instantiation of the new SCC proof object and updates to the attributes of the SCC tool object.

```
var X@SCCTool : SCCTool .   vars O O' : Oid .   var M : Module .
var Atts : AttributeSet .   var MNReg : Map{ModuleName, Oid} .

crl [check-sc] :
  < O : X@SCCTool | reg : MNReg, Atts > (to O from O' : check_sc M)
=> if not isParameterized?(M) and-else not M :: STheory
  then createSCCProof(O', < O : SCCTool | reg : MNReg, Atts >,
    processSCCheck(M))
  else < O : SCCTool | reg : MNReg, Atts >
    (to O' from O : output(
      mfe-error('SCC 'cannot 'check 'parameterized 'modules
        'or 'theories. '\n'))
    fi   if not getName(M) in domain of MNReg .
```

SCC operates on unparameterized modules with initial semantics. If module M conforms to these two constraints, then a new SCC proof object is instantiated with the emptiness result of the corresponding automaton. The registry attribute `reg` of the SCC tool object is updated with the name of module M and the unique object identifier of the newly created proof object. In the case that module M does not conform to these constraints, an error message is issued to the user, no SCC proof obligation is instantiated, and the SCC tool object remains unchanged (this is done by a rule omitted here).

Dependencies. The SCC tool object depends on termination and sort-decreasingness checks by MTT and CRC tool objects. In general, tool dependencies can be resolved at instantiation, for which the controller object provides information of the tools available in the environment (including itself).

The SCC tool object is instantiated by defining an *instantiation token* that takes a map representing the available tools as input and by a rewrite rule that creates the instance of the tool object with the information provided in the map. The map in the instantiation token identifies a (tool) name TN with a (tool) object identifier O whenever the tool object for TN has object identifier O . Observe that this instantiation mechanism with a map as a parameter benefits the modularity and extensibility of MFE: if more tools become available in MFE, there is no need to modify the way tool objects are currently instantiated in MFE.

A new SCC tool object is created by a term `init-scc(TS)` of sort `Configuration`, with TS a term of sort `Map{ToolName, Oid}`. The following rewrite rule `init-scc-to` instantiates the SCC tool object and creates a message to display.

```
op init-scc : Map{ToolName, Oid} -> Configuration .

var TS : Map{ToolName, Oid} .

rl [init-scc-to] :
  init-scc(TS)
  => < TS["SCC"] : SCC |
      tools : TS,
      grammar : SCC-GRAMMAR,
      current : null-oid,
      index : 0,
      reg : empty >
  (to TS["MFE"] from TS["SCC"] : output(string2qidList(scc-banner))) .
```

The tool map TS is used to assign the object identifier to the SCC tool object, namely, the object identifier with associated tool name "SCC". Tool names are globally known and the controller is responsible for their uniqueness. Since SCC proof objects do not exist when the SCC tool object is created, attribute `index` is set to value 0 and attribute `reg` is set to value `empty`. As the result of the SCC tool object successful creation, an output message is sent to the controller object with a welcoming message, here encoded by constant term `scc-banner`.

5.3 Making SCC Operational in MFE

In order to make the SCC tool object operational in MFE, it needs to be registered with the controller object and added to the object configuration. A unique tool name and a unique object identifier are required for registering the SCC tool object with the controller object. In MFE, the string "SCC" is the global tool name for the SCC tool object and `scc` its object identifier. Therefore, the tools map in the controller object is updated with the pair "SCC" \mapsto `scc`.

The SCC tool object is added to the initial configuration of MFE by means of the initialization token `init-scc(T)`, with the updated tools map T .

```
op TOOLS+SCC : -> Map{ToolName, Oid} .
eq TOOLS+SCC = (TOOLS, "SCC" |-> scc) .

rl [init] :
  init
  => [ nil,
      < mfe : Controller |
        db : initialDatabase, input : nilTermList, output : nil,
        default : 'CONVERSION, current-tool : mfe, tools : TOOLS+SCC >
      ...
      init-scc(TOOLS+SCC),
      nil ] .
```

6 Using MFE

In this section, we illustrate some features and commands of MFE on the classical example of the readers and writers, using a slightly modified version of the presentation in [4, Sections 12.3 and 12.4].¹ In fact, a similar proof can be found in [4], but using the tools in isolation and with no support for keeping track of the pending proof obligations.

In this specification, a state is represented by a term $\langle R, W \rangle$ where R and W are, respectively, the number of readers and writers accessing a critical resource. In the system, there should not be more than one writer, or writers and readers at the same time. To obtain this behavior, a writer can only access the critical resource if no nobody else is using it, and a reader can gain access to the critical resource only if there are no writers using it. Readers and writers can leave the critical resource at any time.

The following modules `MBOOL` and `MNAT` define, respectively, sorts `MBool` and `MNat` of Boolean values and natural numbers.

```
(fmod MBOOL is
  sort MBool .
  ops true false : -> MBool [ctor] .
endfm)
```

¹ The changes introduced are due to the different treatment of built-ins by the different tools. To avoid conflicts we do not use any built-in in the example. We set the automatic inclusion of the `BOOL` module off and, although not used, the automatic inclusion of module `TRUTH-VALUE` on.

```
(fmod MNAT is
  sort MNat .
  op 0 : -> MNat [ctor] .
  op s : MNat -> MNat [ctor] .
endfm)
```

The readers-writers system can be specified as follows.

```
(mod READERS-WRITERS is
  protecting MNAT .
  sort Config .
  op <_,_> : MNat MNat -> Config [ctor] .
  vars R W : MNat .
  rl [wrt+] : < 0, 0 > => < 0, s(0) > .
  rl [wrt-] : < R, s(W) > => < R, W > .
  rl [rdr+] : < R, 0 > => < s(R), 0 > .
  rl [rdr-] : < s(R), W > => < R, W > .
endm)
```

Before reducing or rewriting any term in this module, we must check the expected execution requirements, namely, the equations being ground terminating and Church-Rosser, and the rules being ground coherent with respect to the equations. We can perform all these checks in MFE. During the verification process, tools keep record of pending proof obligations, are able to submit them to appropriate tools in the environment, and complete their proofs upon reception of messages announcing the discharging of assumptions. In fact, we can choose to complete the proof in different orders. For example, we could check first the termination, then the Church-Rosser property, and then its coherence, or we could choose to directly attempt the coherence proof and let the submission of the proof obligations help in the process.

Let us do first the Church-Rosser proof. To carry such a check we first select the CRC tool.

```
Maude> (select tool CRC .)
CRC has been set as current tool.
```

Since there are no equations in READERS-WRITERS, this module is trivially (ground) Church-Rosser.

```
Maude> (check Church-Rosser .)
Church-Rosser check for READERS-WRITERS
  There are no critical pairs.
  The specification is confluent.
  The module is sort-decreasing.
  Success: The module is therefore Church-Rosser.
```

We now check module READERS-WRITERS ground coherent. We select the ChC tool:

```
Maude> (select tool ChC .)
ChC has been set as current tool.
```

and then issue the checking command:

```
Maude> (check coherence .)
Coherence checking of READERS-WRITERS
  All critical pairs have been rewritten and no rewrite with rules
  can happen at non-overlapping positions of equations left-hand sides.
  The termination and Church-Rosser properties must still be checked.
```

The coherence property assumes the ground termination and ground Church-Rosser properties of equations in module READERS-WRITERS. We use command (submit .) to ask the environment to submit the pending proof obligations to the corresponding tools.

```
Maude> (submit .)
The Church-Rosser goal for READERS-WRITERS has been submitted to CRC.
The termination goal for the functional part of READERS-WRITERS has been
submitted to MTT.
Success: The functional part of module READERS-WRITERS is terminating.
Church-Rosser check for READERS-WRITERS
  There are no critical pairs.
  The specification is confluent.
  The module is sort-decreasing.
Success: The module is therefore Church-Rosser.
The functional part of module READERS-WRITERS has been checked terminating.
The module READERS-WRITERS has been checked Church-Rosser.
Success: The module READERS-WRITERS is coherent.
```

The ground termination and Church-Rosser properties of the functional part of module READERS-WRITERS are checked, answers to the requests are received, and the coherence checker automatically completes the proof and sends to the controller object the success message. The proof of the Church-Rosser property was already in the environment, and therefore the answer is directly returned without further checking. When requested, the termination proof is attempted and it succeeds. When ChC receives all the messages informing of successful discharging of both proof obligations, it completes the proof and sends the corresponding success message.

We are now ready to prove some properties of module READERS-WRITERS. For instance, we verify mutual exclusion in READERS-WRITERS, that is, at most one reader or one writer uses the critical resource at a particular time. We could do this by using Maude's search command or its LTL model checker. However, since the reachable state space is infinite for any initial state, we first need to define an abstraction of the system and prove its correctness. We use the following predicates and abstraction, in which all states having readers are collapsed to a simpler state (it is not relevant to know how many readers are using the critical resource, but whether there is any or not using the critical resource).

```
(mod READERS-WRITERS-PREDS is
  protecting MBOOL .
  protecting READERS-WRITERS .
  ops mutex one-writer : Config -> MBool [frozen] .
```

```

vars M N : MNat .
eq mutex(< s(N), s(M) >) = false .
eq mutex(< 0, N >) = true .
eq mutex(< N, 0 >) = true .
eq one-writer(< N, s(s(M)) >) = true .
eq one-writer(< N, s(0) >) = true .
eq one-writer(< N, 0 >) = true .
endm)

(mod READERS-WRITERS-ABS is
  including READERS-WRITERS-PREDS .
  var N : MNat .
  eq [abs] : < s(s(N)), 0 > = < s(0), 0 > .
endm)

```

To check both the execution and the invariant-preservation properties of this abstraction, we need to check: the equations being ground confluent, sort-decreasing, and terminating; the equations being sufficiently complete; and the rules being ground coherent with respect the equations.

The use of CRC, ChC, SCC, and MTT to carry on these proofs is presented in [4, Section 12.4]. Here we show how the different proofs can be completed inside the environment without the need of switching between execution environments.

We start by checking the sufficient completeness of READERS-WRITERS-ABS:

```

Maude> (select tool SCC .)
SCC has been set as current tool.

Maude> (scc READERS-WRITERS-ABS .)
Checking sufficient completeness of READERS-WRITERS-ABS ...
To complete the proof the specification must be proved ground
sort-decreasing and weakly-terminating.

```

We then submit these assumptions to other tools in the environment.

```

Maude> (submit .)
The sort-decreasingness goal for READERS-WRITERS-ABS has been submitted
to CRC.
The termination goal for the functional part of READERS-WRITERS-ABS has
been submitted to MTT.
Success: Module READERS-WRITERS-ABS is sort-decreasing.
Success: The functional part of module READERS-WRITERS-ABS is terminating.
Success: Module READERS-WRITERS-ABS is sufficiently complete.

```

CRC has not requested a termination proof for READERS-WRITERS-ABS, and therefore has not been informed of the result of the termination proof. Let us just do that. We first select the CRC tool as active tool.

```

Maude> (select tool CRC .)
CRC has been set as current tool.

```

By requesting the CRC's, we realize that the check was completed, and that only ground termination is necessary to complete the ground confluence proof and, with it, obtain a ground Church-Rosser proof for READERS-WRITERS-ABS.

```
Maude> (show state .)
State of the tool:
- Church-Rosser check for READERS-WRITERS :
  There are no critical pairs.
  The specification is confluent.
  The module is sort-decreasing.
  The module is therefore Church-Rosser.
- Church-Rosser check for READERS-WRITERS-ABS :
  All critical pairs have been joined.
  The specification is locally-confluent.
  The module is sort-decreasing.
```

We submit the pending proof obligation.

```
Maude> (submit .)
The termination goal for the functional part of READERS-WRITERS-ABS has
been submitted to MTT.
The functional part of module READERS-WRITERS-ABS has been checked
terminating.
Success: The module READERS-WRITERS-ABS has been checked Church-Rosser.
```

Then, we check module READERS-WRITERS-ABS ground coherent.

```
Maude> (select tool ChC .)
ChC has been set as current tool.

Maude> (check ground coherence READERS-WRITERS-ABS .)
Ground coherence checking of READERS-WRITERS-ABS
The following critical pairs cannot be rewritten:
  cp READERS-WRITERS-ABS1 for abs and rdr-
  < s(0), 0 >
  => < s(#1:MNat), 0 > .
The termination and Church-Rosser properties must still be checked.
```

A critical pair is returned by the ChC tool in the form of a reachability goal. Also, ground termination and Church-Rosser proofs must be obtained. Some of these proofs have already been found, but ChC has not been informed. We submit the pending proof obligations.

```
Maude> (submit .)
The Church-Rosser goal for READERS-WRITERS-ABS has been submitted to CRC.
The goal for critical pair READERS-WRITERS-ABS1 has been submitted to ITP.
The termination goal for the functional part of READERS-WRITERS-ABS has
been submitted to MTT.
The module READERS-WRITERS-ABS has been checked Church-Rosser.
The functional part of module READERS-WRITERS-ABS has been checked
terminating.
```

The ITP does not provide methods to prove the joinability of critical pairs. However, we can carry on a proof by reasoning by cases and using Maude's searching command as in [4, Section 12.4]. We can then use the `(trust .)` command to inform the tool that the proof was completed out of the ITP.

```
Maude> (select tool ITP .)
ITP has been set as current tool.

Maude> (trust .)
Eliminated current goal.
The critical pair READERS-WRITERS-ABS1 has been trusted.
Success: The module READERS-WRITERS-ABS is ground-coherent.
```

Now that the abstraction has been proved correct, we can check both invariants:

```
Maude> (search in READERS-WRITERS-ABS :
      < 0, 0 > =>* C:Config
      such that mutex(C:Config) = false .)
No solution.

Maude> (search in READERS-WRITERS-ABS :
      < 0, 0 > =>* C:Config
      such that one-writer(C:Config) = false .)
No solution.
```

7 Conclusion

The Maude Formal Environment is an executable and highly extensible software infrastructure written in Maude within which a user can interact with several tools to mechanically verify properties of Maude specifications. MFE exploits Maude as a reflective declarative language and system based on rewriting logic in which computation corresponds to efficient deduction by rewriting. We have explained the main design decisions in MFE and integrated, as a proof of concept, five important formal analysis tools with highly heterogeneous designs: namely, Maude's Termination Tool, Church-Rosser Checker, Coherence Checker, Sufficient Completeness Checker, and Inductive Theorem Prover. We also presented a brief overview of the steps underwent to integrate Maude's SCC in MFE and explained how MFE's design decisions allowed for an easy integration. It is important to highlight that the approach taken here for extending MFE with the SCC is one of many possible ways to benefit from the software infrastructure offered by MFE. Finally, we gave a fair overview of some of the features and commands of MFE on the classical example of the readers and writers.

Much work remains ahead. First of all, more tools such as Maude's LTL and LTLR Model Checkers, Maude's Invariant Analyzer Tool, and Real-Time Maude could be integrated in MFE. This will result in a more interesting environment with features for handling broader applications with less effort for the user. One could also think of handling proof obligations such as those for the protecting and extending importations of modules, for the instantiation of parameterized modules, or simply the termination and

Church-Rosser assumptions for equational simplification. More ambitiously, a graphical user interface and support for better interoperability will enhance the user experience with the formal environment. The graphical user interface could be developed, for instance, as a plugin in the Eclipse environment. IMAude and the IOP platform might also be a good candidates for improving tool interoperability and providing the environment with a graphical user interface.

Acknowledgements. The authors would like to thank the anonymous referees for comments that helped to improved the paper. The first author has been partially supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184. The second author has been partially supported by NSF grants CNS 07-16638 and CCF 09-05584.

References

1. User interfaces for theorem provers, <http://www.informatik.uni-bremen.de/uitp/>
2. Aspinall, D., Lüth, C.: Special issue on user interfaces in theorem proving. *Journal of Automated Reasoning* 39(2) (2007)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.: Maude: Specification and programming in rewriting logic. *Theoretical Computer Science* 285, 187–243 (2002)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Clavel, M., Durán, F., Eker, S., Meseguer, J., Stehr, M.O.: Maude as a formal meta-tool. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1684–1703. Springer, Heidelberg (1999)
6. Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.: The Maude formal tool environment. In: Mossakowski, T., Montanari, U., Haverdaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 173–178. Springer, Heidelberg (2007)
7. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science* 12(11), 1618–1650 (2006)
8. Durán, F., Lucas, S., Bevilacqua, V.: MTT: The Maude termination tool (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 313–319. Springer, Heidelberg (2008)
9. Durán, F., Lucas, S., Meseguer, J.: Methods for proving termination of rewriting-based programming languages by transformation. *Electronic Notes in Theoretical Computer Science* 248, 93–113 (2009)
10. Durán, F., Lucas, S., Meseguer, J.: Termination modulo combinations of equational theories. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 246–262. Springer, Heidelberg (2009)
11. Durán, F., Meseguer, J.: Maude’s module algebra. *Science of Computer Programming* 66(2), 125–153 (2007)
12. Durán, F., Meseguer, J.: A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In: Ölveczky, P. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 69–85. Springer, Heidelberg (2010)
13. Durán, F., Meseguer, J.: A Maude coherence checker tool for conditional order-sorted rewrite theories. In: Ölveczky, P. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 86–103. Springer, Heidelberg (2010)

14. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming* (submitted for publication, 2011)
15. Durán, F., Ölveczky, P.C.: A guide to extending Full Maude illustrated with the implementation of Real-Time Maude. *Electronic Notes in Theoretical Computer Science* 238(3), 83–102 (2009)
16. Franssen, M., van den Brand, M.: Design of a proof repository architecture. In: *Proceedings of the 1st Workshop on Modules and Libraries for Proof Assistants (MLPA 2009)*, pp. 19–23. ACM (2009)
17. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) *IJCAR 2006*. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
18. Hemer, D., Long, G., Strooper, P.: Plug-in proof support for formal development environments. In: *Proceedings of the 2005 Australasian Symposium on Theory of Computing (CATS 2005)*, pp. 69–79. Australian Computer Society, Inc. (2005)
19. Hendrix, J.: *Decision Procedures for Equationally Based Reasoning*. Ph.D. thesis, University of Illinois at Urbana-Champaign (2008)
20. Hendrix, J., Clavel, M., Bevilacqua, V.: A sufficient completeness reasoning tool for partial specifications. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, pp. 165–174. Springer, Heidelberg (2005)
21. Hendrix, J., Meseguer, J., Ohsaki, H.: A sufficient completeness checker for linear order-sorted specifications modulo axioms. In: Furbach, U., Shankar, N. (eds.) *IJCAR 2006*. LNCS (LNAI), vol. 4130, pp. 151–155. Springer, Heidelberg (2006)
22. Lucas, S.: MU-TERM: A tool for proving termination of context-sensitive rewriting. In: van Oostrom, V. (ed.) *RTA 2004*. LNCS, vol. 3091, pp. 200–209. Springer, Heidelberg (2004)
23. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set, Hets. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
24. Rocha, C., Meseguer, J.: Constructors, sufficient completeness and deadlock freedom of rewrite theories. In: Fermüller, C.G., Voronkov, A. (eds.) *LPAR-17*. LNCS, vol. 6397, pp. 594–609. Springer, Heidelberg (2010)

Multisimulations: Towards Next Generation Integrated Simulation Environments

Leila Jalali, Sharad Mehrotra, and Nalini Venkatasubramanian

University of California at Irvine, USA
{ljalali, sharad, nalini}@ics.uci.edu

Abstract. In this paper, we consider the challenge of designing a reflective middleware to integrate multiple autonomous simulation models into an integrated simulation environment (multiasimulation) wherein we can model and execute complex scenarios involving multiple simulators. One of the limitations of the simulators is that they are developed by domain experts who have an in-depth understanding of the phenomena being modeled and typically designed to be executed and evaluated independently. Therefore, the grand challenge is to facilitate the process of pulling all of independently created models together into an interoperating multisimulation model where decision makers can explore different alternatives and conduct low cost experiments. We aim to build such integrated simulation environments by creating a loosely coupled federation of pre-existing simulators. We evaluate our proposed methodology via a detailed case study from the emergency response domain by integrating three disparate pre-existing simulators – a fire simulator (CFAST), an evacuation simulator (Drillsim) and a communication simulator (LTESim).

Keywords: Reflective Middleware, Simulation Integration, Metamodels, Methodology.

1 Motivation

In this paper, we consider the challenge of designing a reflective middleware to integrate multiple autonomous simulation models into an integrated environment wherein we can model and execute complex scenarios involving multiple simulators. Modelling and simulation is an important methodology to address a variety of real-world problems; it offers numerous advantages instead of experimenting with the real system itself. Simulation is cheaper, quicker, and enables what-if analyses for better system design [7].

This is particularly true in domains such as emergency response where response plans and methods are validated by simulating disasters and their impact on people and infrastructure. A variety of simulators; e.g., loss estimation tools (HAZUS [13], INLET [14], CAPARS [15], CFD [12]), fire spread simulators (CFAST [10], CCFM [19]), evacuation simulators (DrillSim [9], SDSS [11]), transportation simulators (VISIM [17], PARAMICS [18]) etc., that model different aspects of disasters, their impacts, and response/mitigation processes have been developed. While these simulators are individually important in understanding disasters, their integrated and

concurrent execution can significantly enhance the understanding of the phenomena and interdependencies between multiple aspects of the complex processes. Consider, for instance, a fire simulator, CFAST, that simulates the impact of fire and smoke in a specific region and calculates the evolving distribution of smoke. Since fire and smoke can affect health conditions of individuals in the region of fire, one may wish to further study its impact on the evacuation process as captured within an evacuation simulator, e.g. DrillSim. Similarly, the progress of fire (captured by CFAST) may create infeasible paths/exits for evacuation (as captured by DrillSim). Such what-if analyses can significantly improve the understanding of adverse impacts such as increase in evacuation times or increased exposure to undesired particulates enabling intelligent decision making to improve the response. Not only do we need to understand how will fire and smoke distribute in a specific region, we also need to plan what traffic routes will people use to evacuate the affected regions, what demands will be placed on the hospital services in the region, etc. The individual simulation models such as those for studying the impact of fire and smoke need to be integrated with those analyzing the traffic movement through the highways and arteries of the affected area, and with those analyzing the resource constraints of hospital systems among others.

The need for integrated execution of simulators is well recognized and is the main driver of U. S. Department of Defense (DoD) High Level Architecture (HLA) initiative [4] which has become the de-facto standard technical architecture for military simulations. HLA aims to promote interoperability and reusability among simulators. While HLA is suited to developing new simulators that can be easily integrated, its broader applicability to combining pre-existing simulators is questionable [21]. HLA forces developers to provide a particular functionality or to conform to specific standards in order to participate in the integration process; the rigid assumptions and limitations on participants makes it difficult to integrate pre-existing simulators without significant modification (especially in non-military domains).

In this paper, we consider the problem of integration of pre-existing simulators. We refer to such an integrated simulation environment as a *multisimulation*. We aim to build multisimulations by creating a loosely coupled federation of pre-existing simulators. We explore a reflective middleware approach to address challenges of integrated simulation environments in which interoperability of different simulators can be ultimately achieved in a flexible and efficient manner. Unlike the significant code rewrite required in the HLA case, our multisimulation framework permits individual simulators to maintain their autonomy (i.e. retain their internal representations of time/state etc.), thereby avoiding the need for rigid common interfaces across simulators.

This paper is organized as follows. In Section 2, we discuss our multisimulation architecture and the main challenges. In Section 3 we discuss our methodology for simulation integration that supports the interoperability of multiple existing simulation models. In Section 4 we discuss the implementation of our system prototype. We evaluate our proposed approach via a detailed case study that integrates multiple real world simulators. Finally we draw conclusions.

2 Multisimulation Architecture

We propose a reflective middleware architecture (Figure 1) to support the development of integrated simulation platforms. Our initial efforts [24] focused on using structural reflection [1] to reify (abstract out) the structure of objects and components of the underlying simulators. The base-level consists of the various (pre-existing) simulators that must be integrated. In the proposed architecture, integration of different simulators can be ultimately achieved by using the meta-level for specifying/modeling the properties of the different simulators and reasoning about the interactions among the different simulators. The meta-level is built on base-level simulators; reification of base-level entities yield data structures at the meta-level, modified features of these structures that implement the integration are then reflected to the base-level. A closer look at the base-level simulators themselves reveals that the structural aspects of the simulation application are not merely in the simulator code, backend databases and models stored in domain-specific formats contain aspects of the simulators that may need to be explored as well. In general, there can be many kinds of meta-level entities to cover various integration aspects.

Given the potential black-box nature of simulators developed by experts in diverse domains, we believe that achieving a completely automated plug-and-play integration of simulators is a very difficult, if not infeasible challenge. Our goals are more modest– we intend to develop enabling tools that will simplify the task of simulation integration with a wide range of simulators that vary in the degree to which they expose their interfaces and implementations. Our solution does not require simulator developers to adhere to a strict programming interface or conform to particular design styles - the ability to flexibly interoperate with multiple simulators is our goal.

By using the metamodeling capability the model elements that need to be integrated can be extracted. In other words, in our approach, we formulate the metamodel that captures concepts of interest using a publish- subscribe mechanism for data exchange – here, subscribers (the simulation integration tasks) express interest in aspects that they want to observe (implemented by base-level

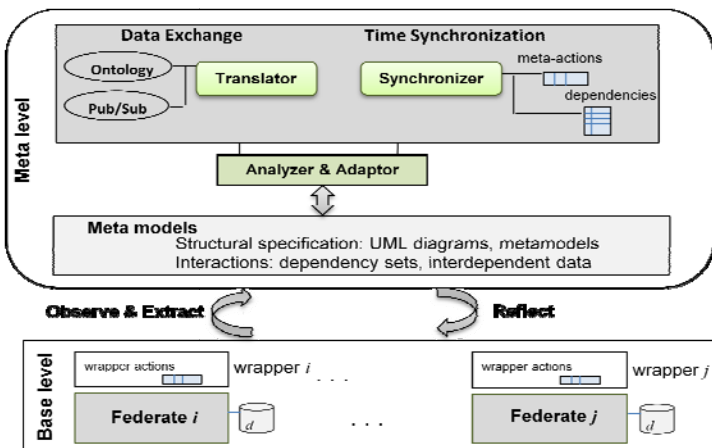


Fig. 1. Multisimulation Architecture

simulators) – when changes in these monitored aspects occur at the base simulators, the meta-level entities receive information or updates of interest via publishers. A pre-existing set of ontology models assist in the matching process for the pub-sub implementation of the simulation integration task; these include domain ontologies that are representations of knowledge in a well-circumscribed domain. Interoperability of different simulators can be achieved by sharing and understanding the metamodels. Implementing the semantic constraints for simulation integration is a human in the loop process which results in the annotations that are invisible to base-level computation and are provided to the meta-level.

In contrast to prior work on simulation integration (e.g. HLA) [4], [2], [3], [5], [27] in our architecture we do not need to integrate simulations tightly into a common framework, but we make it feasible to semi-automatically compose simulation models via a looser coupling approach that avoids the need to adhere to a rigid common interface, which can hinder leveraging prior work. We explore a reflective architecture to address challenges of integrated simulation environments in which interoperability of different simulators can be ultimately achieved in a flexible and efficient manner while preserving the autonomy of the individual simulators.

3 Integration Methodology

In this section, we describe the general structure of our methodology and its relevant issues. The complex process of integration is decomposed in several phases, and for every phase several tasks are specified, with the strategies to be followed. We describe step by step, different phases, making use of an example to make it easy to understand.

Figure 2 demonstrates the basic steps of methodology for simulation integration. The first step is to extract metadata from basic simulators and to describe it using metamodels. Next is to analyze metamodels to discover inter-dependencies between simulators. The first two steps are pre-processing steps that are human-in-loop process. When federation runs, meta-level modules ensure the correctness until the end of simulation. In the following we describe each step in details.

3.1 Preprocessing Steps: Extract Simulators' Metadata and Dependencies

The first step is to extract simulator-related meta-data and describe it at meta-level using metamodels. Metamodels are abstracts of lower-level details of integration and interoperability which make the underlying simulator more understandable. Figure 3 shows our meta-model. There are several key classes in the metamodel: model type, actions, model elements (data items) which could be local data or shared data, input or output parameters, actions, and constraints. We construct our metamodel using UML (Unified Modeling Language).

Model type includes information about the type of simulation model. In general simulators can be categorized into Discrete-event, Agent-based, System dynamics. They also can be categorized based on the time management mechanism that they employ as time stepped simulators or event based simulators [1, 7]. In time stepped

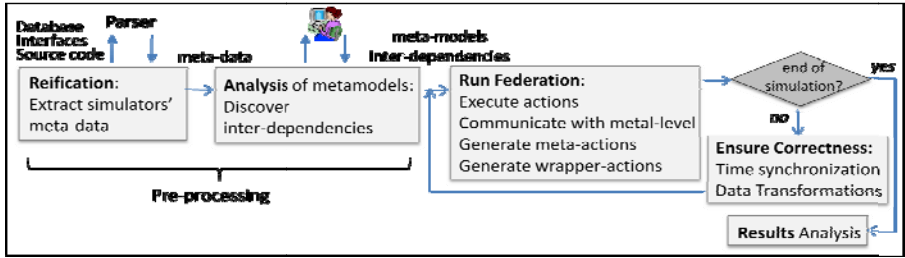


Fig. 2. The Basic Steps of Methodology

simulators, for each execution of the main control loop the simulation time is incremented by one quantum of time Δt . In the case of event based simulators, execution is driven by an event list, each event has a time stamp (usually causality preserving) and the simulation time jumps from one event time stamp to the next without representing all the values in between. For example for Drillsim [9] we have agent-based and time-step as model type.

Model elements are the main elements of a simulation and can be captured from the interfaces, the source code, or databases. We develop a set of tools to extract the simulator information as metamodels from the base-level simulators. Model elements consist of simulation model features. Since we are interested in structural reflection, currently we only use structural features which include classes and attributes. We may also take behavioral features into account to represent operations and associations in future. We implemented a parser using a tool for large scale code repositories search to extract the entities and attributes from a complex and large simulators using the simulator’s source code, interfaces, and databases. Then we group extracted information into features to capture the structure of the simulator. The features are put into the same class if they are considered equivalent.

Since our metamodel needs to take several domain expert simulators into account, the metamodel should be comprehensive, yet extensible. In our metamodel, we also consider input and output parameters. The careful examination of the features in

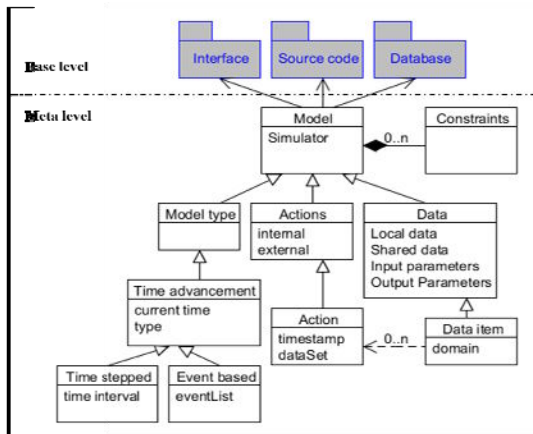


Fig. 3. Basic Metamodel

various simulators of the different domains has allowed us to identify and categorize common features using key classes. Finally, constraints are the number of limits for the simulation parameters in the simulation model. We will discuss complete examples of metamodels in our case study later.

In the second step, we analyze the metamodels to discover the interdependencies between simulators. We use dependency descriptors to specify the dependency between a data item d in simulator S_i and a data item d' in simulator S_j when updates on d' need to be reflected into d : $S_i \leftarrow S_j = \langle d \in D_i, d' \in D_j, f \rangle$. Note that dependency notion is directional. S_i is the supplier simulator, S_i is the consumer simulator. Here, d and d' are *interdependent data items*. In general, there can be more than one dependency between two simulators describing multiple aspects of their relationships. A dependency function, f , defines the relationship between two data items values. At each iteration, the new value of d is determined by the dependency function $f: Dom(d') \rightarrow Dom(d)$, that is $v = f(v')$. For each dependency between simulators such a dependency function is defined at meta-level.

3.2 Run Federation

We consider each simulator's execution as a sequence of actions (time steps in time stepped simulators or events in event based simulators). Typically simulators execute their set of actions independently from the beginning to the end of simulation in an uncoordinated way. To participate in a federation, each of these simulators needs to be modified, i.e. the introduction of synchronization points at which the simulations needs to stop processing its actions and communicate with the meta-level in order to be synchronized with other simulators.

Simulators are interfaced with met-level by using wrappers. The wrapper determines the external actions for which the simulator needs to communicate with meta-level and sends them to meta-level. External actions are those actions that access at least one data item which is an interdependent data item (there exists a dependency between this data item and another data item in another simulator). Upon receiving such actions from a simulator, the meta-level generates meta-actions to notify its dependent simulators.

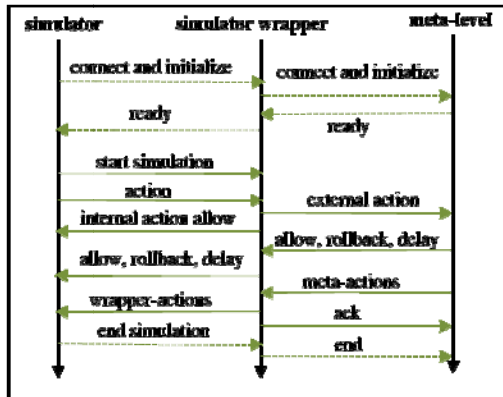


Fig. 4. Base-level federates, wrapper, and meta level interactions (step 3)

Figure 4 shows the details of Step 3 in our methodology. First, the wrapper sends a request for connection to the meta-level. The meta-level confirms the connection and sends information about interdependent data items and dependencies to the wrapper. Once simulation starts, the wrapper determines the external actions and sends corresponding requests to the meta-level. Upon receiving a request, the meta-level modules evaluate the dependencies and respond to the wrapper with a decision (allow, rollback, or delay) based on the scheduling approach used (which will be discussed in the next section). It also sends to the wrapper meta-actions that contain the updates by other simulators. The wrapper reflects the received meta-actions on the execution of the underlying simulator. This loop is continued until the end of simulation.

4 Challenges

There are several challenges in integrating multiple autonomous simulation models. The first challenge arises from modeling complex scenarios using multiple simulation models and the analysis of cause-effect relationships between those models. Given the potential black-box nature of simulators developed by experts in diverse domains we believe that achieving a completely automated plug-and-play integration of simulators is a very difficult, if not infeasible challenge. Our goals are more modest – we intend to develop enabling tools that will simplify the task of simulation integration with a wide range of simulators that vary in the degree to which they expose their interfaces and implementations. We do not want to require simulator developers to adhere to a strict programming interface or conform to particular design styles - the ability to flexibly interoperate with multiple simulators is our goal. Another challenge arises from the fact that each simulator uses its own models and entities; these must now be integrated in the context of a single simulation. The simulators need to exchange the data and have a correct interpretation of the data they send and receive. Time synchronization is yet another challenge. When integrating simulators, there is a need for synchronization of time between the different models. In the following we discuss the details:

(a) *Managing Complexity of Interoperating Systems.* Integration of independently created models can lead to a complex interoperating system of systems that need to be managed efficiently. Understanding the interoperability issues that arise in this context is the main aim of multisimulations. In our proposed approach, we use meta models to describe simulator-related meta-data; a way to infer data transformations, and a means of specifying and automatically executing orchestration. Metamodels can make the underlying simulator more understandable. Additionally, metamodels are abstracts of lower-level details of integration and interoperability. Other challenges arise from heterogeneity of the data that simulation models need to exchange. Since we are working with existing simulation models, it is necessary to analyze the data types used internally by the simulators. It might be possible to adapt prior work on data transformation and integration [26]. We plan to investigate whether such techniques can be extended to account for or detect potential data exchange issues that will arise.

(b) Correctness. Another challenge in integrated simulation environments is to ensure the correctness of multisimulations (e.g. preserving causality among different simulation models). In particular, we focus on time synchronization and data consistency as critical problems that must be addressed to ensure the correct interoperability of the concurrently executing simulators. The simulation clock that controls simulation time during execution of a simulation resides within each simulator itself. Time synchronization mechanisms are needed to ensure causal correctness for models that use different time advancement mechanisms. Most of available synchronization methods need the participants to agree on a common interpretation of time and a common time advancement method. Our goal is to leverage existing simulators, as is, while enabling data interchange between them and to accommodate multiple time management and advancement mechanisms implemented internally in participating simulators, preserving the autonomy of the individual simulators. We need to describe the semantics of the internal time advancement in different simulation models (e.g., whether it is a continuous-time model with observations made at regular time intervals, a discrete-time model with observations only at “ticks,” or a discrete-event model with observations only at irregularly spaced event-occurrence times). The spatial coordinate system must also be specified so that different models can be spatially aligned. We came across both issues in our exercise and resolved them with appropriate interpolations of data and transformations to overcome mismatches.

(c) Scalability. Accurate modeling and analysis of large scale and complex scenarios presents a scalability challenge. Modeling such complex scenarios places considerable stress on the system resources. Such scenarios involve a lot of entities, e.g. agents with complex behavior operating on dynamic environments. In this paper we focus on interoperability and correctness issues. Scalability is currently another ongoing topic of research on multisimulations [25]

We focus on time synchronization and data consistency as critical problems that must be addressed to ensure the correct interoperability of the concurrently executing simulation models. In the following we discuss the details of our approach for federation time synchronization and data consistency.

4.1 Time Synchronization

Time synchronization services is a research area with a very long history. In general the time synchronization mechanisms can fall into two different categories: 1) conservative, and 2) optimistic [16]. A conservative strategy ensures the legality of simulator actions by delaying the actions such that the dependencies are preserved in the concurrent execution of actions of different simulators. This approach prevents action roll-backs. A simulator can proceed if the synchronizer can guarantee that by executing its external action, no dependencies will be violated. In the optimistic strategy, we accept the fact that violations occur, but instead of trying to prevent them by delaying the actions, we simply choose to detect them after the action has executed and then resolve the violation when it does occur; by aborting the actions that caused the violation.

We categorize simulators based on the time management mechanism that they employ as being time stepped or event based (Table 1)[7]. In time stepped simulators,

Table 1. Time-stepped and Event-based Simulators

Time-stepped simulator	Event-based simulator
<pre> while (simulation in progress) do for each tick do read data; modify data; time = time + Δt; end for end while </pre>	<pre> while (simulation in progress)do Event e= nextEvent; while(e!=null)do process(e); time= timestamp(e); e= nextEvent; end while end while </pre>

for each execution of the main control loop the simulation time is incremented by one quantum of time Δt . In the case of event based simulators, execution is driven by an event list, $E = \{e_m | m = 1, 2, \dots\}$, each event has a time stamp (usually causality preserving) and the simulation time jumps from one event time stamp to the next without representing all the values in between. For every two events e_a and e_b we have the following property: $timestamp(e_a) \leq timestamp(e_b)$ when $a \leq b$. We allow different simulators to have different levels of granularity in their events or timestamps.

Just as in any concurrency controller, the synchronizer can follow a conservative or optimistic strategy for scheduling actions.

Conservative approach. A conservative strategy ensures the legality of schedules by delaying the actions such that the dependencies are preserved in the concurrent execution of actions of simulators. The delay will cause the simulator to freeze until the synchronizer allows it to proceed. This approach prevents action roll-backs. A simulator can proceed if the synchronizer can guarantee that by executing its action, no dependencies will be violated; otherwise, the action will be delayed.

Optimistic approach. In some applications it is quite common to be in a situation where although simulators are working simultaneously on interdependent data, violations are infrequent and dependencies continue to be preserved. When this is the case, an optimistic strategy becomes efficient. In the optimistic approach, we accept the fact that violations occur, but instead of trying to prevent them by delaying the actions, we choose to detect them after the action has executed and resolve the violation when it does occur; by aborting the actions that caused the violation.

Above strategies may become more (or less) effective as a multisimulation progresses. The efficacy of a specific strategy at a point in time is a factor of the underlying dependencies and actions taken by the concurrently executing simulators. Initially, the cost of abort is small, so the optimistic strategy will be preferred. However, as the simulator proceeds, aborts costs become increasingly high. Therefore, the conservative strategy becomes more effective. We plan to propose a hybrid approach that combines the benefits of both the optimistic and conservative strategies by considering the underlying dependencies and the costs of delays and aborts to make an informed decision for an action.

4.2 Data Transformation

In general, the data management module provides data transfer that preserves the meaning and relationships of the data exchanged between two simulators. Since we

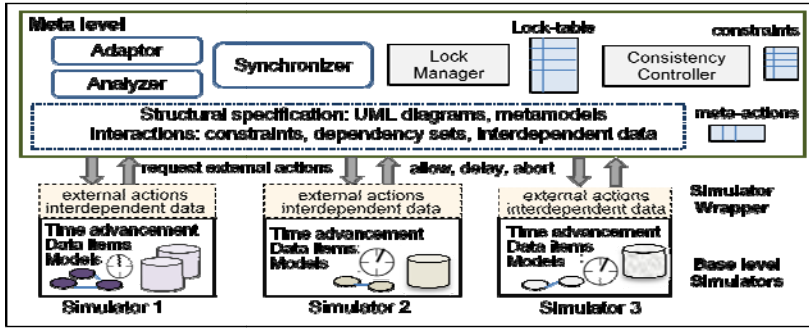


Fig. 5. Multisimulation Framework

are working with existing simulators, we cannot use the methods based on the common representation of data. Each simulator may have its own data representation which can not be easily modified. We used data translators that work based on the dependencies between federates. If the data translators are implemented correctly, they can provide immediate conduits to publish or subscribe to information.

To integrate a simulator to the multisimulation, adaptors components need to be built. The purpose of developing the adaptors is to provide a descriptor of a simulator to implement a standard interface that makes the run-time multisimulation capable of controlling the data exchange between multiple simulators. Adaptors are responsible for transforming data values of entities of one simulator to the corresponding values of entities of another simulator. For instance, a traffic simulator provides updates for other simulators on traffic congestion values for the links in the network geography. If other simulators use a different geography, a conversion must take place to map a value on the first geography to one or more values on other geographies.

We plan to adapt prior work on data exchange [26], which infers a default transformation mapping from a source schema to a target schema. Our goal is to investigate whether such mapping-generation algorithms can be extended to account for or detect potential time-management, geometry-management, and unit-conversion issues that may arise. In any case, there are standard transformations that will be required for the majority of model mash-ups. These include unit conversions, time and space interpolations and aggregations, and database join-operations on files. Future work includes identifying the set of such functions and establishing a standard library. Such transformation functions will need to be highly scalable. Other non-standard transformations can be quite challenging (e.g., aligning or combining different social networks in multiple agent-based simulation models, or allocating household caloric intake among household members).

5 Prototype System Implementation

In this section we discuss the general structure of our methodology, its relevant issues, and the implementation of a prototype multisimulation system. Figure 5 shows different modules in the prototype system. Consistent with our metaarchitecture design philosophy, the design aims to separate the base level aspects of each

simulator (this includes the simulator code, the backend databases and models stored in domain-specific formats) from the meta-level synchronization and adaptation mechanisms. Base-meta interactions occur through simulator wrappers that handle the processing of external actions in each simulator by forwarding requests to meta-level. There are 3 key modules at meta-level: (a) a *Synchronizer* which uses the proposed approaches to monitor and control concurrent execution in the multisimulation. This module also makes use of a lock manager to coordinate concurrent access to simulators data items. (b) an *Analyzer* which analyzes the interactions between simulators using meta-models to capture the dependencies which stored in a separate table and indexed by its corresponding interdependent data items. (c) an *Adaptor* which manages the data exchange and adapts information that is passed between simulators through the design of wrapper modules for each simulator.


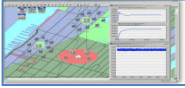
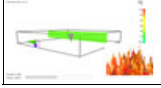
In the initialization steps the wrapper sends a request for connection to the synchronizer. The synchronizer confirms the connection and sends information about interdependent data items and dependencies to the wrapper. Once simulation starts, the wrapper determines which actions are external actions and sends corresponding requests to the synchronizer. Upon receiving a request, the meta-level modules evaluate the dependencies and respond to the wrapper with a decision (allow, rollback, or delay) based on the scheduling approach used. It also sends to the wrapper meta-actions that contain the updates by other simulators. The wrapper reflects the received meta-actions on the execution of the underlying simulator. This loop is continued until the end of simulation.

The implementation of allow and delay in the wrapper is straightforward; it will proceed or freeze the simulation respectively. In the case of rollback, associated with the rollback notification from synchronizer is a time, t , which indicates the time in the past to which simulation needs to be rolled back. One option is to start the simulation from time t , initialize all the interdependent and local data items values to the values that they had in time t , and run the simulation – obviously this involves a high overhead for storing/checkpointing the data item values at each instance of time, especially when working with pre-existing simulators. In the case of simulators when it is not possible to start a simulation from a random time in the past, we will be required to rerun the simulation from its start time until it reaches time t .

5.1 Integrating Real-World Simulators

To ground our work in reality, we develop a case study for simulation integration using three pre-existing real world simulators from the emergency response domain – the primary goal is to validate our approach and synchronization solutions and understand issues in its realization. The specific simulators are (1) CFAST, a fire simulator that simulates the effects of fire and smoke inside a building and (2) Drillsim, an activity simulator that model a response activity evacuation and (3) LTESim, a communication simulator for the next generation wireless network infrastructure. Table 1 summarizes the three simulators and their properties. In our case study, we focus primarily on integrating simulation and models aimed at informing emergency response policy decision making, but we expect our framework and methods will be applicable to other complex problem domains.

Table 2. Three Real-World Simulators

Evacuation Simulator	Communication Simulator	Fire Simulator
 <ul style="list-style-type: none"> ◆ Drillsim [9], Time stepped ◆ Open source (in Java) ◆ Parameters: health profile, visual distance, speed of walking, num. of ongoing call, etc. Output: num. of evacuees, injuries, etc. 	 <ul style="list-style-type: none"> ◆ LTEsim [20], Event based ◆ Open source (in Matlab) ◆ Parameters: num. of transmit and receive antennas, uplink delay, network layout, channel model, bandwidth, frequency, etc. Output: pathloss, throughput, etc. 	 <ul style="list-style-type: none"> ◆ CFAST [10], Time stepped ◆ Black-box (no access to source) ◆ Parameters: bldg geometry, materials of construction, fire properties, etc. Output: temperatures, pressure, gas concentrations: CO₂, etc.

1) **Fire simulator:** CFAST, the Consolidated Model of Fire and Smoke Transport, is a simulator that simulates the impact of fires and smoke in a specific building environment and calculates the evolving distribution of smoke, fire gases, and temperature [10]. CFAST has several interfaces to input the parameters that contain information about the building geometry, fire properties, and etc. The simulator produces outputs that contain information about temperatures, ignition times, gas concentrations, and etc.

2) **Activity simulator:** Drillsim is a multi-agent that plays out the activities of a crisis response process, e.g. building evacuation in response to an evolving fire hazard. Drillsim simulates human behavior in a crisis at fine granularities [9] - agents represent an evacuee, a building captain, etc. Every agent has a set of properties associated with it, such as physical perceptual profile (e.g., range of sight, speed of walking) and the current health status of the agent (e.g. injured, unconscious).

3) **Communication simulator:** LTEsim, the communication simulator in our case study, is a LTE System Level simulator [20] which abstracts the physical layer and performs network level simulations of 3GPP Long Term Evolution with lower complexity. We chose LTEsim because the LTE standard has several improvements in capacity, speed, and latency and will be the technology of choice for most existing 3GPP mobile operators [8]. LTEsim considers several parameters to model the communication infrastructure (such as number of transmit and receive antennas, network layout, bandwidth, pathloss, and etc.).

In our integration scenario, the fire simulator, CFAST, is used to simulate the impact of fire and smoke in a specific region and calculates the evolving distribution of smoke; fire and smoke can affect evacuation process, e.g. people's health condition, in the evacuation simulator, Drillsim, which has impacts on communication patterns in communication simulator, LTEsim. Such integration is useful to conduct better what-if analyses and understand various factors that can adversely delay evacuation times or increase exposure and consequently used to make decisions that can improve safety and emergency response times. The first step is to specify meta-models for the three base level simulators and dependencies across them (see Appendix). The following are the examples of information interchanged among simulators:

- A harmful condition in CFAST can affect an individual's health in Drillsim.
- Smoke in CFAST can decrease an agent's visual distance in Drillsim.

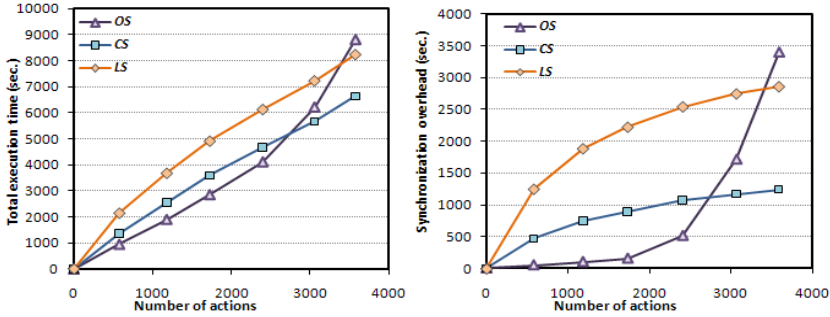


Fig. 6. (a) Average synchronization overhead (b) Total execution time vs. the number of actions

- The number of ongoing communications in Drillsim can affect network pathloss and throughput in LTEsim.
- Pathloss in LTEsim can be used to determine connectivity/coverage in Drillsim.

In our current implementation, several such dependencies specified (the actual number of dependencies required was in the range of 10-50 for most situations).

5.2 Initial Results

Our experiments are based on the case study described above where we integrate 3 real world simulators. In our experiments, we implemented techniques for synchronization across the three simulators: We implemented three different solutions for time synchronization across the three simulators: Lock-step approach (LS), conservative approach (CS), and optimistic approach (OS). Lock-step approach is the most conservative approach for the purpose of evaluating the other two approaches, by having a lock step schedule. All simulators advance step by step, and at any step they synchronize by locking at data item level until the next step. The lock table is maintained at meta-level. By locking shared data in the beginning of action and releasing the locks at the end, we can prevent deadlocks. In OS, we consider wait time before each rollback to be 0.5 s.

Table 3. Comparison between HLA and Multisimulation Architecture

Criterion	HLA	Multisimulation Architecture
Objective	<ul style="list-style-type: none"> – Interoperability – Reusability 	<ul style="list-style-type: none"> – Semantic Interoperability, Reusability – Flexibility
Domain	<ul style="list-style-type: none"> – Defense 	<ul style="list-style-type: none"> – Flexible via use of domain ontologies
Complexity	<ul style="list-style-type: none"> – Low level knowledge needed – Lack of semantic interoperability 	<ul style="list-style-type: none"> – No need to conform the internal properties – Semantic constraints implemented at the metalevel
Time Management	<ul style="list-style-type: none"> – Optimistic and conservative methods 	<ul style="list-style-type: none"> – Optimistic and conservative methods
Separation of Concerns	<ul style="list-style-type: none"> – Merges domain-specific and integrated simulation aspects 	<ul style="list-style-type: none"> – Separate concerns related to simulation domain to those related to integration mechanisms

We studied the synchronization overhead and the total execution time using different techniques. We measure synchronization overhead by adding the synchronization overhead in all simulators. In *CS*, we considered the total delay time, i.e. the duration a simulator is blocked and the locking overhead, i.e. the time needed for acquiring or releasing locks, to calculate the synchronization overhead. In *OS*, we considered the total rollback time. Figure 6-a and 6-b illustrate the average synchronization overhead per time step and average execution time for different numbers of actions. In our base case, the number of dependencies between simulators is 30 (a reasonably large number for our case study). The synchronization overhead in *CS* is much lower as compared to *OS* during later phases of execution. This is due to the high rollback time in *OS*.

Table 3 presents a brief comparison of the reflective architecture to HLA. Using HLA outside the defense domain such as our case study is very complex, if not impossible. In HLA low level knowledge needed from participants. Each simulator must use the common data format that leads to simulations that are very closely coupled to an underlying database. Since the HLA environment is a fully distributed simulation environment, the simulators must fully conform to the designated features of the HLA standard. Note that transforming existing simulators to conform to the standard may not always be feasible. In our reflective architecture each simulator can have its own data representation, internal time management, and data management. Therefore, we do not force the simulators to change their internal properties. Another advantage of our reflective architecture is separation of concerns, that is, separate the concerns related to the simulation domain from those related to the integration mechanisms. Additionally it provides a design that is more adaptable, flexible and easier to extend.

6 Related Work and Conclusions

To best of our knowledge, simulation integration has been studied in two domains – (a) military command-and-control [4], [2], [5], [27], and (b) games and virtual environments [3]. The U.S. Department of Defense (DoD) has promoted the development of standards to provide a common framework in which simulators can be integrated. These include standards such as SIMulator NETworking (SIMNET) [27], Distributed Interactive Simulation (DIS) [5], Aggregate Level Simulation Protocol (ALSP) [2], High Level Architecture (HLA) [4]. These standards provide specific services for interoperability in niche applications, for example DIS for human-in-the-loop simulators or ALSP for war games. The recent HLA effort has become the defacto standard technical architecture for military simulations – it aims to promote interoperability and reusability between simulators.

While HLA is suited to developing new simulators that can be easily integrated, its broader applicability to combine pre-existing simulators is questionable [6], [21]. It is a complex standard designed specifically for the military domain and is not transparent enough – too much low level knowledge is needed from the practitioner. HLA forces developers to provide a particular functionality or to conform to specific standards in order to participate in the integration process; the rigid assumptions and limitations on participants makes it difficult to integrate pre-existing simulators

without significant modification (especially in non-military domains). As in the case with the HLA architecture, solutions in the game community are also prescriptive - they force the developers to provide a particular functionality to participate in the integration process and have different assumptions/limitations on how participants interact. Such methods are unsuited to the integration of pre-existing simulators.

In this paper we proposed a reflective middleware architecture for simulation integration that implements structural reflection to alleviate the flexibility issues in current simulation integration techniques. In this architecture, the meta-level is structured as a series of metamodels representing the various simulators. We have implemented a detailed case study from the emergency response domain by integrating 3 disparate simulators: a fire simulator (CFAST), an evacuation simulator (Drillsim) and a communication simulator (LTESim). Future research will focus on addressing challenges in the complexity associated with generalizing the meta-models for simulators, integrating simulators in other domains including earthquake and transportation simulators, and addressing the challenges of data transformation in multisimulations.

References

- [1] Kon, F., Costa, F., Blair, G., Campbell, R.H.: The Case for Reflective Middleware. *Communications of the ACM* 45(6), 33–38 (2002)
- [2] Weatherly, R., Seidel, D., Weissman, J.: Aggregate Level Simulation Protocol. In: *Summer Computer Simulation Conference*, pp. 953–958 (1991)
- [3] Jain, S., McLean, C.R.: Integrated simulation and gaming architecture for incident management training. *Simulation*. In: *Proc. of the Winter Simulation*, pp. 904–913 (2005)
- [4] Kuhl, F., Weatherly, R., Dahmann, J.: *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice-Hall, New Jersey (1999)
- [5] Davis, P.K.: Distributed Interactive Simulation (DIS) in the Evolution of DoD Warfare Modeling and Simulation. *Proceedings of the IEEE* 83(8), 1138–1155 (1995)
- [6] Boer, C., Bruin, A., Vebraeck, A.: Distributed simulation in industry- a survey: part 2 – experts on distributed simulation. In: *Winter Simulation Conference*, pp. 1061–1068 (2006)
- [7] Fujimoto, M.R.: *Parallel and Distributed Simulation Systems*. John Wiley Inc. (2000)
- [8] McQueen, D.: 3GPP LTE: the momentum behind LTE adoption. *IEEE Communication* 47, 44–45 (2009)
- [9] Balasubramanian, V., Massaguer, D., Mehrotra, S., Venkatasubramanian, N.: DrillSim: A Simulation Framework for Emergency Response Drills. *ISI*, 237–248 (2006)
- [10] Peacock, R., Jones, W., Reneke, P., Forney, G.: CFAST– Consolidated Model of Fire Growth and Smoke Transport (Version 6) User’s Guide, NIST Special Publication (2005)
- [11] De Silva, F.N., Eglese, R.W.: Integrating Simulation Modeling and GIS: Spatial Decision Support Systems for Evacuation Planning. *JORS* 51(4), 423–430 (2000)
- [12] Abanades, A., Sordo, F., Lafuente, A., Martinez-Val, J.M., Munoz, J.: Application of computational fluid dynamics (CFD) codes as design tools. In: *5th Int. Conf. on ISFA* (2007)
- [13] HAZUS-MH: Multi-hazard Loss Estimation Methodology. User Manual (2003)

- [14] Cho, S., Huyck, C.K., Ghosh, S., Eguchi, R.T.: Development of a Web-based Transportation Modeling Platform for Emergency Response. In: 8th Conf. on Earthquake Eng. (2006)
- [15] CAPARS, <http://www.alphatrac.com/PlumeModelingSystem>
- [16] Jefferson, D.: Virtual Time. *ACM Trans. Programming Lang. Sys.* (3), 404–425 (1985)
- [17] Verkehr, A.: VISIM V3.6 Innovative Transportation (2001)
- [18] Cameron, G., Wylie, B., McArthur, D.: PARAMICS- Moving Vehicles on the Connection Machine. In: Conf. on High Performance Networking and Computer, pp. 291–300 (1994)
- [19] Cooper, L.Y, Forney, G.P.: The consolidated compartment fire model (CCFM) computer code application CCFM.VENTS - Part I: Physical basis. NISTIR 4342 (1990)
- [20] LTE System Level Simulator, <https://www.nt.tuwien.ac.at/>
- [21] Boer, C., Bruin, A., Vebraeck, A.: Distributed Simulation in Industry - a survey, part 3- the HLA standard in industry. In: Proc. of the 40th Conf. on Winter Sim, pp. 1094–1102 (2008)
- [22] Huang, J., Tung, M., Hui, L.: Ming-Che Lee An Approach for the Unified Time Management Mechanism for HLA Source Simulation, vol. 81(1), 45–56 (2005)
- [23] Ramamritham, K., Calton, P.: A Formal Characterization of Epsilon Serializability. *IEEE Transactions* (1995)
- [24] Jalali, L., Venkatasubramanian, N., Mehrotra, S.: Reflective Middleware Architecture for Simulation Integration. In: ARM 2009, Urbana Champaign, Illinois (2009)
- [25] Balasubramanian, V., Kalashnikov, D., Mehrotra, S., Venkatasubramanian, N.: Efficient and scalable multi-geography route planning. In: EDBT 2010, Switzerland (2010)
- [26] Haas, L. M., Hernández, M.A., Ho, H., Popa, L., Roth, M.: Clio Grows Up: From Research Prototype to Industrial Tool. In: Proc. ACM SIGMOD, pp. 805–810 (2005)
- [27] Pope, A.: The SIMNET Network and Protocols, Technical Report 7102. BBN, MA (1989)

Semantics, Simulation, and Formal Analysis of Modeling Languages for Embedded Systems in Real-Time Maude

Peter Csaba Ölveczky

Department of Informatics, University of Oslo

Abstract. This survey paper presents an overview of how Real-Time Maude has been used to provide a formal semantics and formal analysis capabilities to a wide range of modeling languages for embedded systems, namely, a behavioral subset of the avionics modeling standard AADL, a synchronous version of AADL, the discrete-event models of the powerful graphical modeling language Ptolemy II, two very different approaches to extend model transformations with time, and an imperative language for handset software.

1 Introduction

Real-time embedded systems (RTEs)—such as automotive, avionics, and medical systems—are hard to design correctly, since subtle timing aspects impact system functionality, yet are safety-critical systems whose failures may cause great damage to persons and/or valuable assets. However, most modeling languages for RTEs that are used in industry currently lack a formal semantics, which not only limits unambiguous communication between model developers, but also implies that models described in such languages cannot be subjected to formal analysis to prove safety properties or identify security vulnerabilities. Furthermore, some modeling languages are not executable, which limits the possibility to even simulate their models. There is therefore a clear need for:

- A formal semantic framework in which the precise semantics of a modeling language for RTEs can be defined in a natural way; and
- associated simulation and formal analysis tools which support the automated formal analysis of models in such languages.

Furthermore, to be useful for model-based system engineering in practice, the formal analysis framework should also:

1. Allow model developers to define analysis commands without understanding the formal language or the formal representation of their models; and
2. provide formal analysis results, such as counterexamples in temporal logic model checking, that the model developer can easily understand.

A number of advanced modeling tools provide a code generation infrastructure to support the generation of deployment code from a design model. Once the formal semantics of a modeling language has been defined, we can leverage this code generation infrastructure to automatically synthesize a *formal verification model* from the informal design model, enabling a *formal model engineering* process that combines the convenience of modeling using an informal but intuitive modeling language with formal analysis.

There exist a number of formal analysis tools for real-time systems (see [49]). However, there is a significant gap between the formalisms of these tools, such as timed automata [3] or timed Petri nets [14,46], that sacrifice expressiveness for decidability, and the expressiveness of modeling languages for industrial RTESSs.

In contrast to such formal tools, Real-Time Maude [40]—which extends the rewriting-logic-based Maude system [15] to support the formal specification, simulation, and model checking of real-time systems—emphasizes expressiveness and ease of specification over algorithmic decidability of key properties. In Real-Time Maude, the state space and the functional properties of the system are defined as an equational specification, the instantaneous local transitions are modeled as rewrite rules, and time advance is modeled by tick rewrite rules. Real-Time Maude is particularly suitable for modeling real-time systems in an object-oriented style. Because of its expressiveness, Real-Time Maude has been successfully applied to a wide range of advanced state-of-the-art systems that are beyond the pale of timed automata, including wireless sensor networks algorithms [21,42], scheduling algorithms that require unbounded queues [36], and large multicast protocols [41,23]. It is also worth pointing out that, although key properties for Real-Time Maude are in general undecidable, they are decidable under conditions that are satisfied by many systems encountered in practice [38].

Real-Time Maude’s natural model of time, together with its expressiveness, should make it a suitable semantic framework for modeling languages for RTESSs. Such languages then also get the following formal analysis capabilities for free:

- simulation;
- reachability analysis;
- linear temporal logic (LTL) model checking for for untimed LTL properties, as well as time-bounded LTL model checking for analyzing systems with an infinite reachable state space; and
- TCTL_{≤,≥} model checking of *timed* CTL formulas [22].

Real-Time Maude addresses the desiderata (1) and (2) above as follows:

1. A key Real-Time Maude¹ feature that makes it easy for the user to define his/her analysis queries, without having to understand Real-Time Maude or the formal representation of his/her model, is the possibility to equationally define *parametric* atomic state propositions (and “state patterns” for reachability analysis). This allows us to define a useful set of parametric state

¹ Most of the appealing features of Real-Time Maude mentioned in this paper are inherited from the Maude system that Real-Time Maude extends.

propositions in the Real-Time Maude interpreter of a language, making it easy for the user to define both search patterns and temporal logic formulas.

2. A key requirement to (i) understanding the results of Real-Time Maude analyses, and (ii) being able to map them back into the original modeling formalism is to have, respectively, a small *representational distance* between the original models and their formal counterparts, and a one-to-one correspondence between these models. Given that hierarchical composition and encapsulation play key roles in modeling languages for industrial systems, the possibility of defining *hierarchical* objects enable us to achieve both small representational distance and the above one-to-one correspondence.

In this survey paper, we illustrate the use of Real-Time Maude as both a semantic framework and a simulation and formal model checking tool for modeling languages for RTEs by summarizing recent such applications of Real-Time Maude on the following modeling languages:

1. A behavioral subset of the AADL [45] modeling standard for avionics and embedded automotive systems.
2. The Synchronous AADL modeling language [4] that can be used to define synchronous system designs in AADL.
3. Ptolemy II discrete-event (DE) models. Ptolemy II [18] is a well established graphical modeling and simulation tool used in industry. The Ptolemy II DE modeling language is fairly challenging as it combines a synchronous fixed-point semantics with time and hierarchical models.
4. A real-time extension of the MOMENT2 model transformation framework [9,8] that supports the formal specification and verification of model-based real-time and embedded systems within the Eclipse Modeling Framework.
5. The eMotions [44,43] framework for defining domain specific visual modeling languages, which provides powerful high-level constructs for defining the timed behaviors of the visual models.
6. An SDL-inspired modeling language for handsets, developed at DOCOMO Labs, which has more expressive timer features than Erlang [19].

To illustrate this use, we give, for each modeling language, a brief overview of the language, a (part of a) small example of a model, an overview and some fragments of the Real-Time Maude semantics, and an overview of the formal analysis support provided by Real-Time Maude for the language. The definition of the Real-Time Maude semantics of each of the above modeling languages has been published separately elsewhere [34,5,11,43,1]. A brief overview of the use of Real-Time Maude for formal model engineering of embedded systems, that does not discuss the semantics of modeling languages, appears in [33].

Related Work. That rewriting logic is a suitable semantics framework in which a range of formal models of concurrency can be naturally represented was shown already in [24,25,27]. The rewriting logic semantics project [29] provides rewriting logic semantics and Maude analysis to an impressive set of programming languages, including Java, Beta, Haskell, Lisp, Python, and Smalltalk. Furthermore, it has been shown that real-time rewrite theories are a natural semantic

framework for real-time systems in which a range of *formal models* for such systems, including timed and hybrid automata, timed Petri nets, timed Actors, etc., can be represented [37,17].

Paper Structure. Section 2 briefly introduces Real-Time Maude. Sections 3 to 8 present the six modeling languages and their Real-Time Maude semantics and analysis, and Section 9 gives some concluding remarks.

2 Real-Time Maude

A *rewrite theory* [24,13] is a tuple $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic theory* [26] with E a set of possibly conditional equations and membership axioms, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A . The theory $(\Sigma, E \cup A)$ specifies the system’s state space as an algebraic data type. R is a collection of *labeled conditional rewrite rules* specifying the system’s local transitions, each of which has the form² $[l] : t \longrightarrow t' \text{ if } \bigwedge_{j=1}^m u_j = v_j$, where l is a *label*. Such a rule specifies a *one-step transition* from a substitution instance of t to the corresponding substitution instance of t' , *provided* the condition holds. The rules are universally quantified by the variables appearing in the Σ -terms t, t', u_j , and v_j , and are applied *modulo* the equations $E \cup A$.³

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* [37], that is, a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$, such that:

1. $(\Sigma, E \cup A)$, where E is terminating and confluent modulo A , contains a specification of a sort `Time` defining the (discrete or dense) time domain.
2. The rules in R are decomposed into:
 - “ordinary” rewrite rules that model *instantaneous* change, and
 - *tick (rewrite) rules* that model the elapse of time in a system. Such tick rules have the form $[l] : \{t\} \xrightarrow{u} \{t'\} \text{ if } \textit{cond}$, where $\{_ \}$ is a constructor of a new sort `GlobalSystem` and u is a term of sort `Time` denoting the *duration* of the rewrite. In Real-Time Maude, tick rules are written

`crl [l] : {t} => {t'} in time u if cond.`

The initial state of a system must be equationally reducible to a term $\{t_0\}$. The form of the tick rules then ensures uniform time elapse in all parts of a system.

We briefly summarize the syntax of Real-Time Maude and refer to [15] for more details. Operators are introduced with the `op` keyword: `op f : s1 . . . sn -> s.`

² In general, the condition may also contains rewrites $w_i \longrightarrow w'_i$ and *memberships* $t_k : s_k$; however, this paper does not use that extra generality. Furthermore, in Maude and Real-Time Maude, an equational condition $u_i = w_i$ can also be a *matching equation*, written $u_i := w_i$, which instantiates the variables in u_i to the values that make $u_i = w_i$ hold, if any. See [15] for further explanations.

³ Operationally, a term is reduced to its E -normal form modulo A before any rewrite rule is applied in Real-Time Maude. Under the coherence assumption [48] this is a complete strategy to achieve the effect of rewriting in $E \cup A$ -equivalence classes.

They can have user-definable syntax, with underbars ‘ $_$ ’ marking the argument positions. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are used by the Maude engine to match terms *modulo* the declared axioms. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `r1` and `cr1`. The mathematical variables in such statements are declared with the keywords `var` and `vars`. An equation $f(t_1, \dots, t_n) = t$ with the `owise` (for “otherwise”) attribute can be applied to a subterm $f(\dots)$ only if no other equation with left-hand side $f(u_1, \dots, u_n)$ can be applied.⁴

In object-oriented Real-Time Maude modules, a *class* declaration

```
class C | att1 : s1, ... , attn : sn .
```

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n . An *object* of class C in a given state is represented as a term $\langle O : C \mid att_1 : val_1, \dots, att_n : val_n \rangle$ of sort `Object`, where O , of sort `Oid`, is the object’s *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n . In a concurrent object-oriented system, the state is a term of the sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. Since a class attribute may have sort `Configuration`, we can have *hierarchical* objects which contain a subconfiguration of other (possibly hierarchical) objects and messages.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
r1 [l] : < 0 : C | a1 : 0, a2 : y, a3 : w, a4 : z > =>
        < 0 : C | a1 : T, a2 : y, a3 : y + w, a4 : z >
```

defines a parameterized family of transitions (one for each substitution instance) which can be applied whenever the attribute `a1` of an object `0` of class `C` has the value `0`, with the effect of altering the attributes `a1` and `a3` of the object. “Irrelevant” attributes (such as `a4`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

A *subclass* inherits all the attributes and rules of its superclasses.

Formal Analysis in Real-Time Maude. To cover all time instants, in particular for dense time domain, the tick rules typically have the form

```
cr1 [tick] : {t} => {t'} in time x if x <= u /\ cond .
```

where x is a new variable of sort `Time` not occurring in t and not initialized by matching equations in *cond*. This ensures that time can advance by *any* amount

⁴ A specification with `owise` equations can be transformed to an equivalent system without such equations [15].

less than or equal to u . Such tick rules are called *time-nondeterministic* and are not directly executable, since many choices are possible for instantiating x . In contrast to, e.g., timed automata, where the restrictions in the formalism allow the discretization of the dense time domain by defining “clock regions,” so that all states in the same clock region satisfy the same properties [3], for the more expressive Real-Time Maude formalism there is not such a “quotient.” Instead, Real-Time Maude executes time-nondeterministic tick rules by offering a choice of different *time sampling strategies* [39], so that only some moments in the time domain are visited. The choice of such strategies includes advancing time by a fixed amount Δ in each application of a tick rule, or advancing time to the next moment when some action must be taken.

Taking a selected time sampling into account, a Real-Time Maude specification is *executable* under reasonable conditions, such as the equations being confluent and terminating, possibly modulo some structural axioms [15], and the theory being coherent [48].

We summarize below the Real-Time Maude analysis commands. All Real-Time Maude analysis commands and their semantics are explained in [39].

Real-Time Maude’s *timed fair rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax

```
(tfrew  $t$  in time <=  $timeLimit$  .)
```

where t is the term to be rewritten (“the initial state”), and $timeLimit$ is a ground term of sort **Time**. Real-Time Maude extends Maude’s *search* command—which uses a breadth-first strategy to search for states that are reachable from the initial state, match the *search pattern*, and satisfy the *search condition*—to search for states that can be reached within a given time interval from the initial state. The unbounded search command is written

```
(utsearch [1]  $t$  =>*  $pattern$  such that  $cond$  .)
```

Real-Time Maude extends Maude’s *linear temporal logic model checker* to check whether each behavior, possibly “up to a certain time,” satisfies a temporal logic formula. *State propositions*, possibly parametrized, should be declared as operators of sort **Prop**, and their semantics is defined by equations of the form

```
ceq { $statePattern$ } |=  $prop$  =  $b$  if  $cond$ 
```

for b a term of sort **Bool**, which defines the state proposition $prop$ to hold in all states $\{t\}$ such that $\{t\} \models prop$ evaluates to **true**. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as **True**, **False**, \sim (negation), \wedge , \vee , \rightarrow (implication), \square (“always”), \diamond (“eventually”), **U** (“until”), and **W** (“weak until”). The command

(mc $t \models u \text{ formula .}$)

then checks whether the temporal logic formula *formula* holds in all behaviors starting from the initial state t . Unbounded model checking is guaranteed to terminate if the state space reachable from the initial state t is finite. When the reachable state space is infinite, *time-bounded* (search and) LTL model checking, in which each behavior is starting in t is only analyzed up to a certain time bound, can be used to ensure termination of the model checking.

Real-Time Maude has also recently been equipped with a model checker for *timed* computation tree logic (TCTL _{\leq, \geq}) properties [22]. TCTL _{\leq, \geq} formulas are defined with the usual CTL operators True, \sim , \wedge , \dots , E (where $E\varphi$ means that there exists a path from the state in which the *path formula* φ holds), A (where $A\varphi$ means that the *path formula* φ holds in each path path from the state), but where the until operator U is indexed with a time bound $\sim r$, for $\sim \in \{<, \leq, \geq, >\}$, so that the *path formula* $\phi_1 U_{\sim r} \phi_2$ holds in a path iff a state satisfying the *state formula* ϕ_2 can be reached in time r' with $r' \sim r$, and so that ϕ_1 holds in all states up to that ϕ_2 -state.

Since the model checking commands execute time-nondeterministic tick rules according to the chosen time sampling strategy, only a subset of all possible behaviors is analyzed. Therefore, Real-Time Maude analysis are in general *not sound and complete* for a given property. However, the reference [38] gives easily checkable conditions that are satisfied by many large Real-Time Maude applications and that ensure that Real-Time Maude analysis are indeed sound and complete for reachability and untimed LTL properties.

3 AADL

The *Architecture Analysis & Design Language* (AADL) [45] is an industrial standard used in avionics, aerospace, automotive, medical devices, and robotics communities to describe a performance-critical embedded real-time system as an assembly of software components mapped onto an execution platform.

The AADL standard is defined using English prose, which makes it ambiguous and also fails to make explicit important assumptions. In joint work with José Meseguer, I have defined the Real-Time Maude semantics of a subset of the *software components* of AADL [34]. This subset defines the architectural and behavioral specification of a system as a set of hierarchical components with ports and connections, with the behaviors defined by Turing-complete transitions systems. Together with Artur Boronat, we have also developed an OSATE plugin that generates a Real-Time Maude specification from an AADL model.

Overview of AADL. An AADL model describes a system as a hierarchy of hardware and software components. Hardware components include: *processor* components that schedule and execute threads; *memory* components; *device* components; and *bus* components that interconnect processors, memory, and devices. Software components include *thread* components that model the

application software to be executed using AADL's *behavior annex* [20]. The OS-ATE modeling environment provides a set of plug-ins for front-end processing of AADL models on top of Eclipse.

In the behavioral subset of AADL for which we have defined a Real-Time Maude semantics, a component *type* specifies the component's *interface* and *properties*, and a component *implementation* specifies the internal structure of the component in terms of a set of *subcomponents* and a set of *connections* linking the ports of the subcomponents. *System* components are the top level components, and a set of *thread* components define their dynamic behaviors. The *dispatch protocol* of a thread determines when the thread is executed. For example, a *periodic* thread is activated at time intervals of the specified period, and an *aperiodic* thread is activated when an event arrives at one of its ports.

The behavior of a thread is defined by a set of guarded state transitions. The actions that are performed when a transition is applied may update local state variables, generate new outputs, and/or suspend the thread for a given amount of time. Actions are built from such basic actions using sequencing, conditionals, and finite loops. When a thread is activated, an enabled transition is nondeterministically selected and applied; if the resulting state is not a *complete* state, another transition is applied, until a complete state is reached (or the thread is suspended).

An AADL Example. For an AADL example, consider a network of medical devices, consisting of a *controller*, a *ventilator machine* that assists a patient's breathing during surgery, and an *X-ray* device [32]. Whenever a button is pushed to take an X-ray, the ventilator machine waits one second and then pauses for two seconds, and the X-ray should be taken after two seconds. To execute the system, we add a *test activator* that pushes the button every second.

The entire system *Wholesys* is a closed system that does not have any features (i.e., ports) to the outside world. Hence, its *type* (interface) is empty:

```
system Wholesys
end Wholesys;
```

The *implementation* of the entire system describes the architecture of the system, with four subcomponents and the connections linking these subcomponents:

```
system implementation Wholesys.impl
  subcomponents TestActivator: system TA.impl;   Xray: system XM.impl;
                Controller: system CTRL.impl;   Ventilator: system VM.impl;
  connections
    event data port Controller.xmContrOutput -> Xray.ctrlInput;
    event data port Controller.vmContrOutput -> Ventilator.ctrlInput;
    event data port Ventilator.feedback -> Controller.feedback;
    event data port TestActivator.pressEvent -> Controller.commandInput;
end Wholesys.impl;
```

The test activator *system*, which generates an event every second, contains a single thread *TestActivator* that is an instance of the following *taThread.impl*:

```

thread taThread
  features      pressEvent: out event data port Behavior::integer;
  properties    Dispatch_Protocol => periodic;      Period => 1 sec;
end taThread;

thread implementation taThread.impl
  annex behavior_specification {**
    states      s0: initial complete state;
    transitions  s0 -[ ]-> s0 {pressEvent!(1);}; **};
end taThread.impl;

```

The thread is dispatched every second; it then applies the transition once (since `s0` is a complete state), and outputs the value 1 through the port `pressEvent`.

Real-Time Maude Semantics of Behavioral AADL. The references [34,35] explain the Real-Time Maude semantics for the targeted fragment of AADL in detail. The semantics of a component-based language can naturally be defined in an object-oriented style, where each component instance is modeled as an object, and where the hierarchical structure of components is reflected in the nested structure of objects. An AADL component instance is represented as an object instance of a subclass of the following class `Component`:

```

class Component | features : Configuration, subcomponents : Configuration,
                  properties : Properties, connections : ConnectionSet, ...

```

The attribute `features` denotes the ports of a component; `subcomponents` denotes its subcomponents; `properties` denotes its *properties*; and `connections` denotes its port connections. The `Thread` class is declared as follows:

```

class Thread | behavior : ThreadBehavior, status : ThreadStatus, ...
subclass Thread < Component .

```

The `behavior` attribute denotes the transition system of the thread, and `status` indicates its current execution status (`active`, `completed`, etc.).

A type declaration of a component (`System`, `Process`, or `Thread`)

```

system typeName [features: ports] [properties: properties] end typeName;

```

maps `typeName` to a set of ports and a set of properties. We therefore consider `system` as a function that maps a type name to the type's interface; hence the above AADL declaration translates to the equation

```

eq system(typeName) = features portsRTM properties propertiesRTM .

```

where `portsRTM` denotes the Real-Time Maude representation of `ports`. Likewise, an AADL component *implementation* declaration, such as

```

system implementation typeName.implName
  ...
end typeName.implName;

```

translates to an *equation*

```
var INSTANCE-NAME : Oid .
eq INSTANCE-NAME system typeName . implName
  = < INSTANCE-NAME : System | features : features(system(typeName)),
                                properties : properties(system(typeName)), ... >
```

A component declaration *instanceName*: *system typeName.implName* in AADL then translates to the Real-Time Maude term

```
instanceName system typeName . implName.
```

For example, the above AADL definition of *Wholesys.imp* is translated to

```
eq INSTANCE-NAME system Wholesys . imp
  = < INSTANCE-NAME : System |
    features : features(system(Wholesys)),
    properties : properties(system(Wholesys)),
    subcomponents :
      (TestActivator system TA . impl)      (Xray system XM . impl)
      (Controller system Controller . impl) (Ventilator system VM . impl),
    connections :
      (Controller . xmContrOutput --> Xray . ctrlInput) ;
      (Controller . vmContrOutput --> Ventilator . ctrlInput) ;
      (Ventilator . feedback --> Controller . feedback) ;
      (TestActivator . pressEvent --> Controller . commandInput) > .
```

The test activator thread *taThread* and its implementation *taThread.impl* are translated as follows:

```
eq thread(taThread) = features (pressEvent out event data thread port)
  properties DispatchProtocol(Periodic); Period(1 Sec).

eq INSTANCE-NAME thread taThread . impl
  = < INSTANCE-NAME : Thread |
    features : features(thread(taThread)),
    subcomponents : none, connections : none,
    properties : properties(thread(taThread)), ...
    behavior : states initial: s0 complete: s0
               transitions s0 -[]-> s0 {(pressEvent ! (1))} > .
```

The real-time concurrent semantics is defined by equations and rewrite rules specifying “message” transportation, mode switches, thread dispatch, thread execution, and timed behavior. For example, the following rewrite rule specifies the execution of an *active* thread. If the thread is in state *L1*, and there is a transition from *L1* whose guard evaluates to *true*, then the transition is executed. The resulting *status* is *sleeping(...)* if the statement list *SL* contains *delay* statements; otherwise, the thread is *completed* if the resulting state *L2* is a complete state, and remains *active* if *L2* is not a complete state:

```

crl [apply-transition] :
  < 0 : Thread | status : active, features : PORTS,
    behavior :
      states current: L1 complete: LS1 others: LS2
      state variables VAL
      transitions (L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS >
=>
  < 0 : Thread | status : (if SLEEP then sleeping(SLEEP-TIME) else
    (if (L2 in LS1) then completed else active fi) fi),
    features : NEW-PORTS,
    behavior :
      states current: L2 complete: LS1 others: LS2
      state variables NEW-VALUATION
      transitions (L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS >
if evalGuard(GUARD, PORTS, VAL)
  /\ transResult(NEW-PORTS, NEW-VALUATION, SLEEP-TIME) :=
    executeTransition(L1 -[GUARD]-> L2 {SL}, PORTS, VAL)
  /\ SLEEP := SLEEP-TIME > 0 .

```

The function `executeTransition` executes a given transition in a state with a given set `PORTS` of ports and assignment `VAL` of the state variables. The function returns a triple `transResult(p, σ, t)`, where p is the state of the ports after the execution, σ denotes the resulting values of the state variables, and t is the sum of the `delays` in the transition actions. The transitions are modeled as a multiset of single transitions; therefore, *any* enabled transition can be applied in the rule.

Formal Analysis of Behavioral AADL Models in Real-Time Maude.

Our *AADL2Maude* OSATE plug-in uses OSATE's code generation facility to automatically generate Real-Time Maude specifications from AADL models. To allow the user to conveniently define his/her search and model checking commands without knowing Real-Time Maude or the Real-Time Maude representation of AADL models, we have defined some useful functions and parametrized state propositions. For example, the term

```
value of  $v$  in component  $fullComponentName$  in  $globalComponent$ 
```

gives the value of the state variable v in the thread identified by the *full component name* $fullComponentName$ in the system $globalComponent$. Likewise,

```
location of component  $fullComponentName$  in  $globalComponent$ 
```

gives the current location/state in the transition system in the given thread.

In the medical example, if `MAIN` is a system component name, the initial state is `{MAIN system Wholesys . impl}`, and `MAIN -> Xray -> xmPr -> xmTh` denotes the full component name of the `xmTh` thread. The ventilator machine must be pausing when an X-ray is being taken, so that the X-ray is not blurred. The following search command checks whether an *undesired* state, where the X-ray thread `xmTh` is in state `xray` while the ventilator thread `vmTh` is *not* in

state `paused`, can be reached from the initial state (the unexpected result shows a concrete unsafe state that can be reached from the initial state):

```
Maude> (utsearch [1]
  {MAIN system Wholesys . impl} =>* {C:Configuration}
  such that
    ((location of component (MAIN -> Xray -> xmPr -> xmTh)
      in C:Configuration) == xray
     and (location of component (MAIN -> Ventilator -> vmPr -> vmTh)
       in C:Configuration) /= paused) .)
```

```
Solution 1   C:Configuration --> ...
```

For LTL model checking purposes, our tool has pre-defined useful parametric atomic propositions, such as *full thread name @ location*, which holds when the thread is in state *location*, and *value of variable in component full thread name is value*, which holds if the current value of the local transition system variable *variable* in the given thread equals *value*. The first of these parametric atomic state propositions is defined as follows:

```
op @_ : PathName Location -> Prop .
var SYSTEM : Configuration .   var L : Location .   var PN : PathName .
eq {SYSTEM} |= PN @ L = (L == location of component PN in SYSTEM) .
```

We can use time-bounded LTL model checking to verify that an X-ray must be taken within three seconds of the start of the system (this command returned a counter-example revealing a subtle and previously unknown design error):

```
Maude> (mc {MAIN system Wholesys . impl} |=t
  <> ((MAIN -> Xray -> xmPr -> xmTh) @ xray) in time <= 3 .)
```

```
Result ModelCheckResult : counterexample( ... )
```

4 Synchronous AADL

In a number of systems targeted by AADL, such as integrated modular avionics systems and distributed control systems in motor vehicles, the system design is essentially a *synchronous design* that must be realized in an asynchronous distributed setting. The key idea of the *PALS architectural pattern* [28,30] is to reduce the design, verification, and implementation of a distributed real-time system to that of its much simpler synchronous version, provided that the network infrastructure guarantees bounds on the messaging delays and the skews of the local clocks. For a synchronous design *SD* and network bounds Γ , we then have a semantically equivalent asynchronous distributed design *PALS(SD, Γ)*. Not only is this important from a system design perspective, but it also makes model checking verification feasible. For example, in [28] we show that for an avionics system developed at Rockwell-Collins, the synchronous design model has

185 reachable states and can be model checked in less than a second, whereas—even in an optimal setting with perfect local clocks, no message delays, and no execution times—the corresponding asynchronous model has more than 3 million reachable states and can be model checked in 2000 seconds. If we allow the message delay to be either 0 or 1 time unit, then no model checking is feasible.

To allow modelers to take advantage of PALS while using AADL, I have, in joint work with Kyungmin Bae, José Meseguer, and Abdullah Al-Nayeem, identified a “synchronous subset” of behavioral AADL, called *Synchronous AADL*, that is suitable to define the synchronous PALS designs in AADL [4].

In Synchronous AADL, we consider a number of *deterministic* components (i.e., threads) that work together with a nondeterministic *environment* that can nondeterministically generate any Boolean outputs that satisfy the *environment constraint*. Such a collection has a *synchronous semantics*: all components perform a transition simultaneously, and whenever a component has a feedback connection to itself and/or to any other component, then the corresponding output becomes an input for any such component at the *next* step.

Synchronous AADL components are formalized in Real-Time Maude in the same way as ordinary AADL components. A synchronous step of the system is formalized in Real-Time Maude by the following tick rewrite rule:

```
var SYSTEM : Object .      var VAL : Valuation .      var VALS : ValuationSet .

crl [syncStepWithTime] :
  {SYSTEM}
=> {applyTransitions(transferData(applyEnvTransitions(VAL, SYSTEM)))}
  in time period(SYSTEM)
  if containsEnvironment(SYSTEM) /\ VAL ;; VALS := allEnvAssignments(SYSTEM).
```

The function `allEnvAssignments` uses Maude’s SAT solver to find all valuations of the Boolean variables in the environment thread that satisfy the environment constraint. The union operator `_;;_` is declared to be associative and commutative; therefore, *any* of these valuations is nondeterministically assigned to the variable `VAL` in the matching condition `VAL ;; VALS := allEnvAssignments(SYSTEM)`. The function `applyEnvTransitions` then performs the environment transition that outputs the values of the variables given by the selected valuation `VAL`. The function `transferData` then transfers the data from the output ports to the receiving input ports and then clears the output ports. Finally, the function `applyTransitions` applies transitions in each non-environment thread until a *complete* state is reached in the thread. The function `period` extracts the period of the system.

The *SynchAADL2Maude* tool is an OSATE plug-in that supports both the generation of a Real-Time Maude model from an Synchronous AADL model, and the formal analysis of the synthesized Real-Time Maude model from *within OSATE*. *SynchAADL2Maude* inherits the predefined atomic propositions from our AADL analysis library, and also allows the modeler to define new formulas and LTL model checking commands in XML within OSATE.

We have used *SyncAADL2Maude* to verify (in less than 10 seconds for each property) a Synchronous AADL model of the avionics system mentioned above. Figure 1 shows the *SyncAADL2Maude* window for this example. Real-Time Maude code generation and model checking are performed by clicking on the **Code Generation** button and the **Do Verification** button, respectively. The LTL properties that the avionics system should satisfy have been entered into the tool, and are shown in the “AADL Property Requirement” table. The **Do Verification** button has been clicked and the results of the model checking are shown in the “Maude Console.”

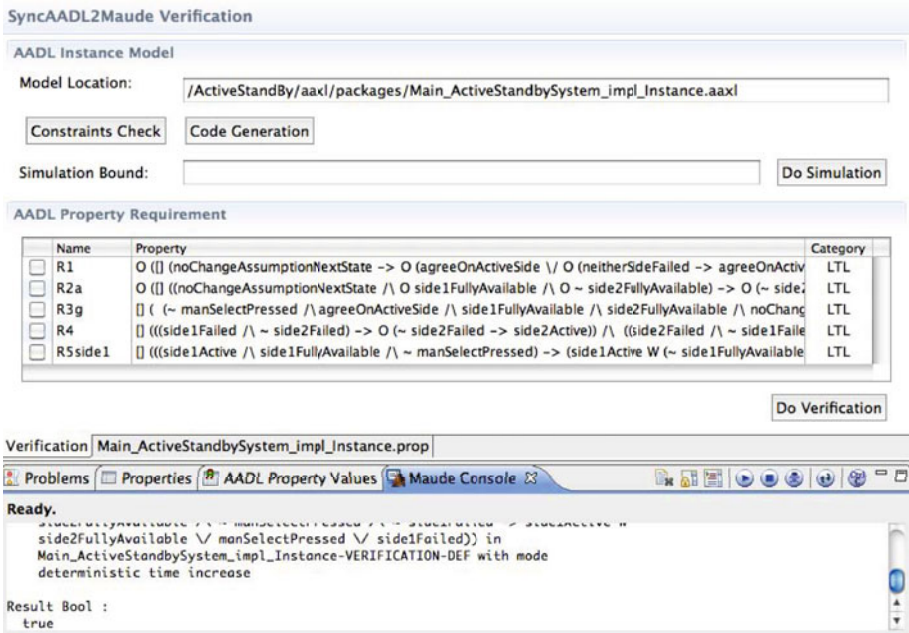


Fig. 1. SyncAADL2Maude window in OSATE

The properties to be verified are managed by the associated XML property file. For example, to add an LTL model checking command to verify a property R1 defined as the LTL formula below, we just add the following `command` tag to the property file:

```
<command>
  <name>R1</name>
  <value type = "ltl">
    [] (noChangeAssumptionNextState
      -> O (agreeOnActiveSide \ / O (noSideFailed -> agreeOnActiveSide))).
  </value>
</command>
```


5 Ptolemy II Discrete-Event Models

Ptolemy II [18] is a well established graphical modeling and simulation tool for real-time and embedded systems that is used in industry. In Ptolemy II, real-time systems are modeled as *discrete-event* (DE) models. Like many graphical modeling languages, Ptolemy II DE models lack formal verification capabilities.

In each iteration of the system, all components with input execute *synchronously*. Unlike in Synchronous AADL, connections between two components are *not* delayed by one round. That is, since connections are instantaneous and the components execute in lock-step, we must compute the *fixed point* of the input for each component in the round before its execution; this input comes from the output of another actor's execution in the same synchronous round.

Real-Time Maude code generation and verification has been integrated into Ptolemy II by Kyungmin Bae, so that a large subset of Ptolemy II DE models can be verified from within Ptolemy II. The paper [5] explains the Real-Time Maude semantics and formal analysis support for Ptolemy II DE models in detail.

Ptolemy II DE Models. A Ptolemy II model consists of a set of *actors* with *input ports* and *output ports*, where communication channels pass *events* from one port to another. Such a model can be encapsulated as a *composite* actor, which may also have input and output ports. Each event has two components: a *tag* and a *value*. A tag t is a pair $(\tau, n) \in \mathbb{R}_{\geq 0} \times \mathbb{N}$, where τ is the *timestamp* denoting the model time at which the event occurs, and n is the *microstep index*.

The operational semantics of DE in Ptolemy II can be explained with the following pseudo-code:

```

Q := empty; // Initialize the global event queue
for each actor A do A.init(); // Possibly add initial events to Q
while Q is non-empty {
  E := set of all simultaneous events at the head of Q;
  remove E from Q;
  initialize ports with values in E or "unknown";
  while port values changed {
    for each actor A do A.fire(); // May change port values
  } // Fixed-point reached for the current tag
  for each actor A do
    A.postfire(); // Update actor state; may also generate new events
}

```

An *event queue* is used for the execution. Events in the event queue are ordered by their tags. In each iteration of the system, the events with the smallest tag are extracted from the event queue and presented to the actors that receive them. All other outputs are first set to *unknown*. Then, the actors receiving events or input are fired in an arbitrary order, possibly repeatedly, until a fixed-point of all output values is reached. Finally, when the fixed-point for the port values has been found, the actors that have received input or have been fed events are executed, in the sense that their states are updated and that they may generate future events that are inserted into the event queue (*postfire*).

A Ptolemy II DE model can contain many different kinds of actors, including clocks that generate events, different kinds of timers, *delay* actors that output their input event after a fixed delay, and finite state machine (FSM) actors, that are similar to the the transition systems of AADL threads. Ptolemy II assumes that actor behaviors are deterministic.

Example. Figure 2 shows a hierarchical Ptolemy II model of a fault-tolerant traffic light system at a pedestrian crossing, consisting of one car light and one

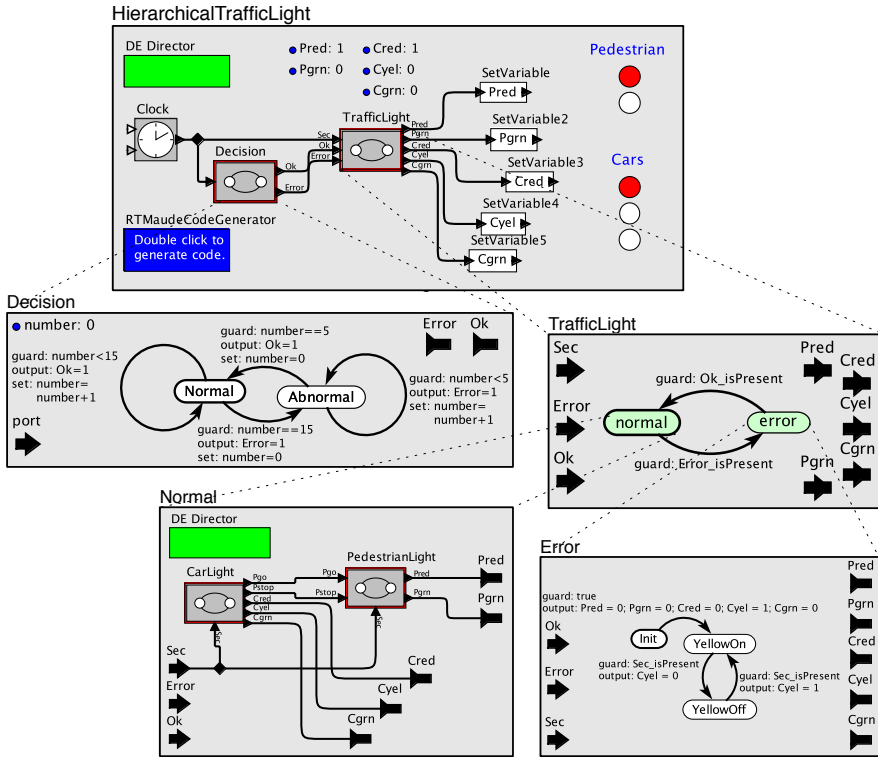


Fig. 2. A hierarchical fault-tolerant traffic light system in Ptolemy II

pedestrian light. Each light is represented by a set of *set variable* actors (Pred and Pgrn represent the pedestrian light, and Cred, Cyel, and Cgrn represent the car light). A light is *on* iff the corresponding variable has the value 1. Assuming that the *clock* actor generates an event every time unit, the FSM actor Decision “generates” failures and repairs by alternating between staying in location Normal for 15 time units and staying in location for Abnormal for 5 time units, and by sending events to the TrafficLight through its Error and Ok ports accordingly. During normal operations, the lights are controlled by the FSM actors CarLight and PedestrianLight that send values to set the variables; in

addition, **CarLight** sends signals to the **PedestrianLight** actor through its **Pgo** and **Pstop** output ports. Figure 3a shows the FSM actor **PedestrianLight**. This actor has three input ports (**Pstop**, **Pgo**, and **Sec**), two output ports (**Pgrn** and **Pred**), three internal states, and three transitions. It reacts to signals from the car light by turning the pedestrian lights on and off. For example, if the actor is in local state **Pred** and receives input through its **Pgo** port, then it goes to state **Pgreen**, outputs the value 0 through its **Pred** port, and outputs the value 1 through its **Pgrn** port. Figure 3b shows the FSM actor **CarLight**. Whenever the

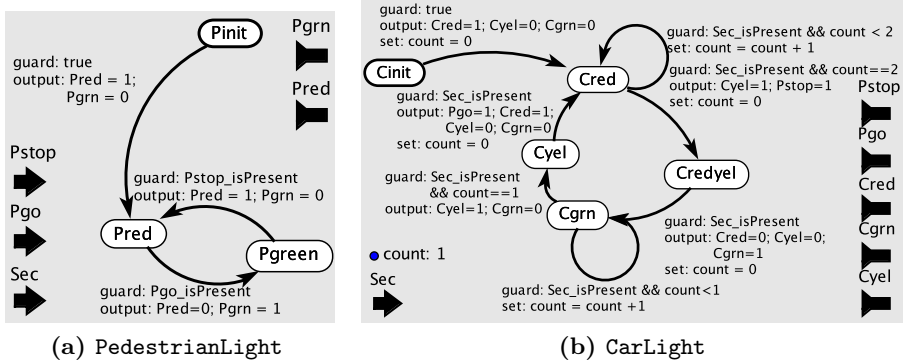


Fig. 3. The FSM actors for pedestrian lights and car lights

TrafficLight model operates in error mode, all lights are turned off, except for the yellow light of the car light, which is blinking.

Real-Time Maude Semantics. The Real-Time Maude semantics is defined in an object-oriented style, where the global state has the form of a *multiset*

```
{actors connections < global : EventQueue | queue : event queue >}
```

where *actors* are objects modeling the actor instances in the Ptolemy model, *connections* are its connections, and *event queue* denotes the global event queue.

Each Ptolemy II actor is modeled in Real-Time Maude as an object instance of a subclass of the class **Actor**, that contains the *ports* and the *parameters* of the actor. Composite actors add an attribute, **innerActors**, denoting its inner actor objects and connections:

```
class Actor | ports : Configuration, parameters : Configuration .
class CompositeActor | innerActors : Configuration .
class AtomicActor .
subclass CompositeActor AtomicActor < Actor .
```

For example, the class **Delay**, that models *Delay* actors that output any received event after a fixed delay, given as a **parameter**, and the class **CurrentTime**, that models *Current Time* actors that output the current time when given some input, are declared as follows:

```
class CurrentTime | currentTime : Time .
class Delay .
subclass Delay CurrentTime < AtomicActor .
```

A *port* is represented as an object, with a name (the identifier of the port object), a status (**unknown**, **present**, or **absent**, denoting the “current” knowledge about whether there is input/output in the current iteration), and a **value**:

```
class Port | status : PortStatus, value : Value .
class InPort .      class OutPort .      subclass InPort OutPort < Port .
sort PortStatus .
ops unknown present absent : -> PortStatus [ctor] .
```

Our semantics has three rewrite rules: the first rule is a tick rule that advances time until the first events in the event queue are scheduled (and reduces the remaining time of the other events according to the elapsed time), and the second rule (not shown) is a “microstep tick rule” that advances “time” with some microsteps if needed to enable the first event in the event queue:

```
vars SYSTEM PORTS PARS : ObjectConfiguration .      var O : Oid .
vars P P' : PortId .      var PS : PortStatus .      vars V TV : Value .
var EVTS : Events .      var QUEUE : EventQueue .      var T : Time .
var NZT : NzTime .      var N : Nat .

r1 [tick] :
  {SYSTEM < global : EventQueue | queue : (EVTS ; NZT ; N) :: QUEUE >}
=>
  {delta(SYSTEM, NZT
    < global : EventQueue | queue : (EVTS ; O ; N) :: delta(QUEUE, NZT) >}
  in time NZT .
```

The following rewrite rule performs a synchronous step of the system when the remaining timer and microstep of the first events in the event queue are zero:

```
r1 [executeStep] :
  {SYSTEM < global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE >}
=>
  {< global : EventQueue | queue : QUEUE >
    postfire(portFixPoints(releaseEvt(EVTS) clearPorts(SYSTEM)))} .
```

The function `clearPorts` sets the **status** of each port to **unknown**. The function `releaseEvt` takes all the ripe events and puts them into the corresponding output ports. This is done by “messages” that percolate down to the appropriate port. The function `portFixPoints` computes all the port values in this round. All ports’ **status** is **unknown** in the beginning of the iteration. Then, the output ports that got events from the event queue have **status present** (as well as a value in the port). All ports connected to the ports with events will also get the

output, and set their port `status` to `present`. Furthermore, when the status and values of a subset of an actor's input ports are known, it can decide whether or not it will generate output in this iteration. Some actors, like *current time* actors, always generate immediate output when receiving input; some actors, like *delay* actors, *never* generate new *immediate* output upon receiving input, while some actors, like FSM actors, may or may not generate immediate output through a given port – depending on its current state and its inputs.

The idea behind the definition of `portFixPoints` is simple. The state has the form `portFixPoints(objects and connections)`. For each case when the status of an `unknown` port can be set to either `present` or `absent`, there is an equation

```
eq portFixPoints(< 0 : ... | ports : <P:Port | status : unknown > PORTS, ... >
               connections and other objects)
= portFixPoints(< 0 : ... | ports : <P:Port | status : present, value : ... >
               PORTS, ... >
               connections and other objects) .
```

(and similarly for deciding that input/output will be `absent`). The fixed-point is reached when no such equation can be applied. Then, the `portFixPoints` operator is removed by using the `owise` construct of Real-Time Maude:

```
eq portFixPoints(OBJECTS) = OBJECTS [owise] .
```

In addition to generic equations, such as transferring an event from a port to a connecting port, `portFixPoints` must be defined for each kind of actor. For example, when the *current time* actor gets an input, it outputs the current model time, and when its input port is `absent`, its output port is set to `absent`:

```
ceq portFixPoints(< 0 : CurrentTime | currentTime : T,
                 ports : <P:InPort | status : PS >
                 <P':OutPort | status : unknown > >
                REST)
= portFixPoints(< 0 : CurrentTime | ports : <P:InPort | >
                 <P':OutPort | status : PS, value : T > >
                REST)
if PS /= unknown .
```

Computing fixed-points for hierarchical models is nontrivial. The main idea is that `CompositeActors` apply the `portFixPoints` operator to its `innerActors` configuration *only* if it transmits a new `present/absent` value to its inner actors, or if there are untreated events from the event queue among its inner actors, or further down in the actor hierarchy. See [5] for details.

Finally, `postFire` “executes” the step on the computed fixed points by changing the states of the objects and by generating new events that are sent to the global event queue. For example, if a time delay actor has input in its `'input` port, it generates an event with delay equal to the current value of the `'delay` parameter. If this delay is 0, the microstep is 1, otherwise the microstep is 0:

```

eq postfire(< 0 : Delay | status : enabled,
            parameters : < 'delay : Parameter | value : TV> PARS,
            ports : < 'input : InPort | status : present, value : V>
                  < 'output : OutPort | > PORTS >)
= schedule-evt(event(0 ! 'output, V), TV, if TV == 0 then 1 else 0 fi)
  < 0 : Delay | > .

```

Formal Verification of Ptolemy II DE Models in Ptolemy II. We have used Ptolemy II's very nice code generation infrastructure to integrate both the synthesis of a Real-Time Maude verification model from a Ptolemy II design model as well as Real-Time Maude model checking of the synthesized model into Ptolemy II itself. When the blue `RTMaudeCodeGenerator` button in a Ptolemy II DE model is double-clicked, Ptolemy II opens a dialog window (shown in Fig. 4) which allows the user to start code generation and to give model checking commands to formally analyze the generated model.

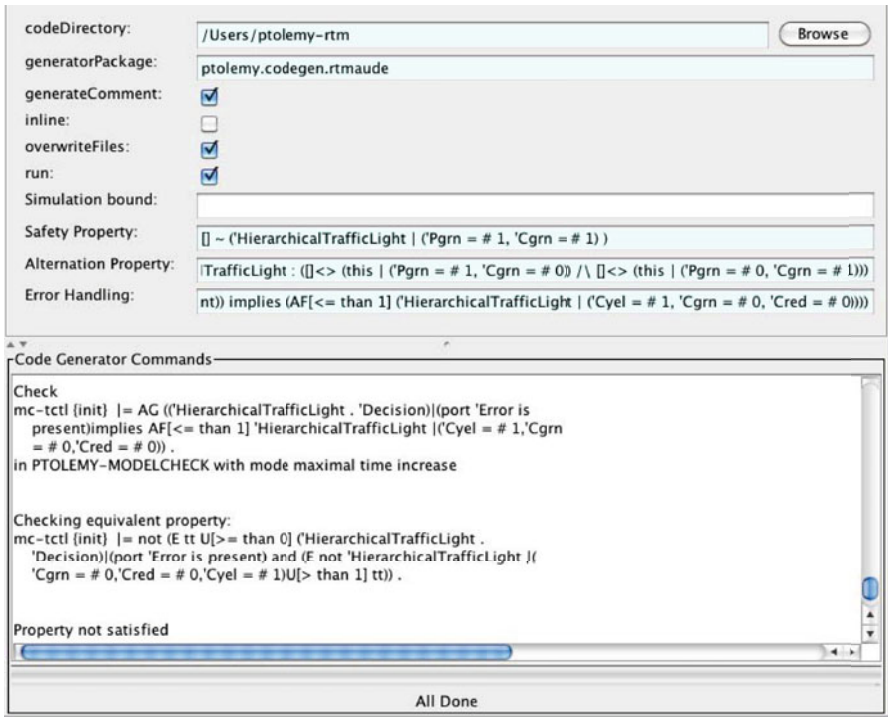


Fig. 4. Dialog window for the Real Time Maude code generation and analysis

We have also predefined in our model checker useful atomic propositions similar to those in our AADL model checker. For example, the proposition

$$\text{actorId} \mid \text{var}_1 = \text{value}_1, \dots, \text{var}_n = \text{value}_n$$

holds in a state if the value of the parameter var_i of the actor $actorId$ equals $value_i$ for each $1 \leq i \leq n$, where $actorId$ is the *global actor identifier* of a given actor. Similarly, $actorId \mid port\ p\ is\ value$ and $actorId \mid port\ p\ is\ status$ hold if, respectively, the port p of actor $actorId$ has the value $value$ and status $status$. For FSM actors and modal models, the proposition $actorId \ @\ location$ is satisfied if and only if the actor $actorId$ is in “local state” $location$.

In the traffic light system, the following *timed* CTL property states that the car light will turn yellow, *and only yellow*, within 1 time unit of a failure:

```
AG (('HierarchicalTrafficLight . 'Decision | port 'Error is present)
    => AF[<= 1] ('HierarchicalTrafficLight | 'Cyel = 1, 'Cgrn = 0, 'Cred = 0))
```

Model checking this property returns a previously unknown counter-example which shows that, after a failure, the car light may show red or green in addition to blinking yellow. We could observe this undesired behavior also during simulations of the model in Ptolemy II (after we had been made aware of the flaw during Real-Time Maude verification).

6 Timed Model Transformations in MOMENT2

The MOMENT2 [9,10] formal model transformation framework is based on a formalization of MOF meta-models in rewriting logic. The static semantics of a system is given as a class diagram (or meta-model) describing the set of valid system states (or models) that are represented as object diagrams, and the dynamics of a system is defined as an *in-place model transformation*. In joint work with Artur Boronat, I have extended MOMENT2 to support the definition of timed behaviors by providing a set of basic constructs such as clocks and timers [11].

Timed Model Transformations in MOMENT2. In MOMENT2, a model transformation is defined as a set of production rules. Each such rule

```
rl l { nac dl nacl { NAC } such that cond ; ...
    lhs { dl { L } }; rhs { dl { R } }; when cond; ... }
```

has a left-hand side L , a right-hand side R , a set of (possibly conditional) negative application conditions NAC , and a condition with the *when* clause. L , R , and NAC contain model patterns, where nodes are object patterns and unidirectional edges are references between objects. For instance, in the pattern $A : \text{Class1} \{ a = V, r = B : \text{Class2} \{ .. \}, .. \}$ an object A of type Class1 has an attribute a , whose value is bound to the variable V , and has a reference r that points to an object B of type Class2 . Several models can be manipulated with a single production rule in MOMENT2. The identifier dl is used to identify which model should be matched by a given model pattern.

To add timed behaviors to models transformations we provide a small set of built-in types for defining *clocks*, *timers*, and *timed values*, which are clocks that

increase with a given rate. The *Ecore* metamodel for these constructs is given in Fig. 5. A *Timer* whose *on* attribute is *true* *decreases* its *value* according to the elapsed time. When the *value* reaches 0, time advance is blocked, forcing the use of a model transformation rule *which also modifies the timer* by either turning off the *Timer* (that is, the *on* attribute is set to *false*), or by resetting its *value*. The *value* of a *Clock* is increased according to the elapsed time, and the *value* of a *TimedValue* object is increased by *the elapsed time multiplied with the rate*.

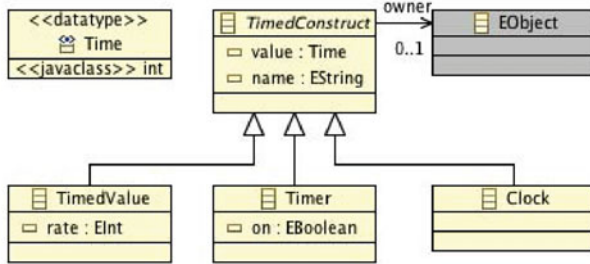


Fig. 5. Ecore metamodel of the predefined timed constructs

A Round Trip Time Example. Consider a very simple protocol for finding the *round trip time* between two neighboring nodes in a network; that is, the time it takes for a message to travel from source to destination, and back. The initiator starts a round of this protocol by sending a *request* message to the other node and recording the time at which it sent the request. When the responder receives the request message, it immediately sends back a *reply* message. When the initiator receives the reply message, it can easily compute the round trip time using its local clock. Since the network load may change, and messages may get lost, the initiator starts a new round of the protocol *every* 100 time units. We assume that the message transmission time is between 5 and 20 time units.

Figure 6 shows the structural Ecore model for this example. The attribute *rtt* of a *Node* denotes the latest computed round trip time value; *lastSentTime* denotes the time that the last request message was sent; *roundTimer* points to the timer upon whose expiration the node starts another round of the RTT protocol; and the *Clock* denotes the local clock of the node. To model transmission delay, each message has an associated clock (to avoid that the message is read too early) and a timer (to ensure that the message is not read too late).

The following rule models the transformation when an active round timer of a node *A* expires (that is, equals 0). The node then sets the value of *lastSentTime* to the current time (as given by its local clock), resets its timer to 100, and generates a *request* message. This message sets its *age* clock to 0, and sets its timer to 20, ensuring that the message is read (or lost) within 20 time units⁵:

⁵ Variable names are capitalized in our model transformation rules.

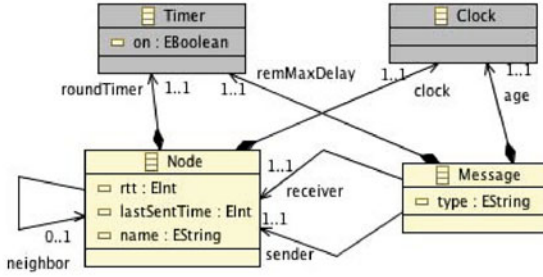


Fig. 6. Class diagram for the RTT example

```

rl sendRequest {
  lhs { model {
    A : Node { clock = C : Clock { value = TIME },
              neighbor = B : Node { },
              roundTimer = RT : Timer { value = 0, on = true } } } };
  rhs { model {
    A : Node { clock = C : Clock { value = TIME },
              neighbor = B : Node { },
              roundTimer = RT : Timer { value = 100, on = true },
              lastSentTime = TIME }
    M : Message { age = MA : Clock { value = 0 },
                  sender = A : Node { },
                  receiver = B : Node { },
                  remMaxDelay = RMD : Timer { value = 20, on = true },
                  type = "request" } } };
}

```

The next rule models the reception (and consumption) of a `request` message. Since message transmission takes at least 5 time units, this can only happen when the `age` clock of the message is greater than or equal to 5. As a result of applying this rule, a `reply` message is created, and sent back to the node A:

```

rl replyRequest {
  lhs { model {
    B : Node { }
    M : Message { age = MA : Clock { value = MSGAGE },
                  sender = A : Node { },
                  receiver = B : Node { },
                  type = "request" } } };
  rhs { model {
    B : Node { }
    NEW-MSG : Message { age = MA2 : Clock { value = 0 },
                       sender = B : Node { },
                       receiver = A : Node { },
                       remMaxDelay = T : Timer { value = 20, on = true },
                       type = "reply" } } };
  when MSGAGE >= 5; }
}

```

The model transformation rules for reading a reply message and for modeling message losses are not shown.

Non-intrusive Timed Specifications. The above approach requires changing the structural design of the application, so that timer and clock objects can be defined in the state. However, one can also define a timed system in a *non-intrusive* way, in which the user-defined metamodel of the system is *not* modified, by having *another* model that contains the timed constructs. The timed behavior of the system is defined with a *multi-model* transformation with two domains, one for the model and one for the time model that extends the initial model. See [11] for details.

Formal Semantics. The Real-Time Maude semantics of timed model transformations extends the rewriting logic semantics of model transformations given in [8]. In particular, the rewrite rules defining the semantics of model transformations are inherited and are considered to be *instantaneous* rewrite rules modeling *instantaneous change*. The real-time rewrite semantics adds the single “standard object-oriented” tick rule (see [39])

```
cr1 [tick] : {<< OC >>} => {<< delta(OC, X) >>} in time X if X <= mte(OC)
```

where OC is a variable denoting multisets of objects representing instances of classes, and X is a variable of sort `Time`.

As usual in Real-Time Maude specifications, the function `delta` defines the effect of time elapse on a system, and the function `mte` defines the *maximum time elapse* possible in a system before some action must be taken. In this case, `delta` decreases the `value` of each active timer in OC according to the elapsed, increases the `values` of the clocks and timed values accordingly, and leaves the non-timed-constructs unchanged. The function `mte` just returns the smallest `value` of any active timer. These functions are defined in the usual Real-Time Maude style and their definitions are therefore not given here.

Formal Analysis. Since the MOMENT2 currently uses Maude as its execution engine, MOMENT2 provides the usual *untimed* Maude analysis methods. Nevertheless, we can also easily perform *time-bounded* analyses by just adding a single unconnected `Timer`—whose initial value is the time bound—to the initial state. When this timer expires, time will not advance further in the system, since no rule resets or turns off the timer. The ability to perform time-bounded analysis is not only useful *per se*, but also makes (time-bounded) LTL model checking analysis possible for systems with an infinite reachable state space, such as our RTT example, that can otherwise not be subjected to LTL model checking.

MOMENT2 provides convenient syntax for specifying search patterns. For example, the following search command searches for a reachable “bad” state

where the recorded `rtt` value is *not* within the desired set of values; that is, by searching for a node `N : Node { rtt = RTT }` whose round trip time value is `RTT <> 0 and (RTT < 10 or RTT > 40)`. We search for one counterexample, and without any bound on the depth of the search tree (`[1,unbounded]`):

```
search [1,unbounded] =>* domain model { N : Node { rtt = RTT } }
                        such that RTT <> 0 and (RTT < 4 or RTT > 16)
```

7 Domain-Specific Visual Languages in e-Motions

The e-Motions model transformation framework [44] for domain-specific visual languages is also based on EMF, with Ecore meta-models defining the abstract syntax of the language, and where the concrete syntax maps elements of the abstract syntax onto graphical objects. Although the specification of behaviors is based on in-place model transformations, the approach to timed behaviors is different from the one taken in MOMENT2. e-Motions does not add constructs such as clocks and timers to support timed model transformations because:

- Since e-Motions does not support multimodel transformations, adding such constructs would necessitate intrusive modifications of the user-defined meta-models to define timed behaviors.
- The use of the timed constructs is considered a very low-level way of specifying timed model transformations; akin to “assembly programming” for model transformations. Since e-Motions focuses on making the framework appealing to domain-specific developers, e-Motions tries to make the definition of timed model transformations as intuitive and high-level as possible.

Therefore, e-Motions provides a powerful high-level way of defining timed behaviors by providing different kinds of *timed model transformation rules*.

Overview of e-Motions. A basic *atomic* timed model transformation rule has the form

$$l : [NAC]^* \times LHS \xrightarrow{[t_{min}, t_{max}]} RHS$$

where l is the rule’s label; LHS (its left-hand side), RHS (its right-hand side), and the optional NAC s (negative application conditions) are model patterns that represent state fragments. The NAC and LHS patterns express the precondition for the rule to be applied, whereas RHS represents the effect of the action. The interval $[t_{min}, t_{max}]$ defines the range of the possible *durations* of the rule, which is executed as follows:

- For any combination of models elements, a rule instance is *triggered* as soon the elements match the LHS and do not match any NAC .
- At any time $t \in [t_{min}, t_{max}]$ after the instance of the rule is triggered, it is executed; i.e., the model fragment is replaced by the corresponding instance of RHS , as long as all the model elements of the triggering match are still present in the model.

An atomic rule can also be declared to be *soft*, which means that it is not triggered eagerly, and/or may be declared to be *periodic*, in which case it is triggered periodically (for each instance) as long it is enabled. In addition to atomic rules, there are also *ongoing rules* that do not have a fixed duration but are applied as long as the precondition holds. Such rule model continuous actions. These are powerful high-level constructs that typically imply that many different timed rules are being applied simultaneously to an object.

An Example. I again use the round trip time example in Section 6. Its abstract syntax is shown in Fig. 7, and Fig. 8 shows the concrete syntax, mapping each class to a visual object. Figure 9 shows a *periodic* atomic rule with duration 0, and period 100. It models the sending of a *RequestMessage* with time stamp *requestTime* equal to the current value of the clock *c* of node *n1*. The message is sealed when sent, and the atomic rule with duration in $[5, 20]$ shown in Fig. 11 models the message transfer time, when any message goes from sealed to unsealed message. The instantaneous atomic rule shown in Fig. 10 models the response of this message, which results in a *ResponseMessage* with the original time stamp being sent back immediately. The rule that assigns the *rtt* value when receiving the response message is not shown. Finally, the *ongoing* rule shown in Fig. 12 models each clock being continuously updated according to the elapsed time *T*.

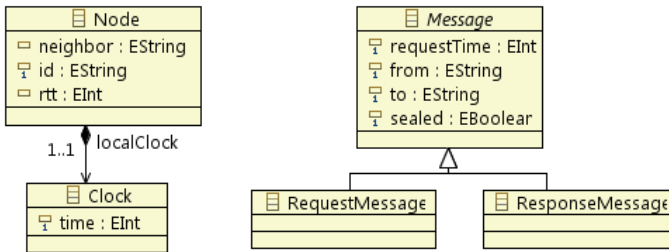


Fig. 7. Meta-model for the RTT example



Fig. 8. Concrete syntax for the RTT example

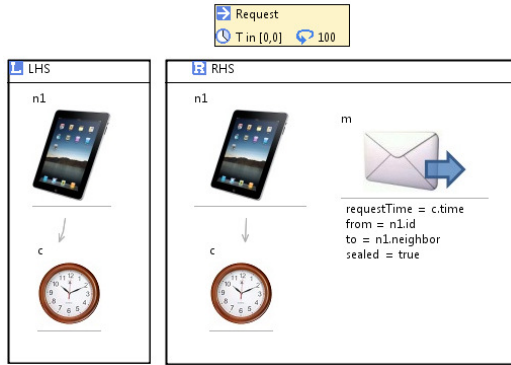


Fig. 9. Periodic instantaneous rule initiating a round of the RTT protocol

Real-Time Maude Semantics. I focus on the real-time aspects and refer to [43] for an explanation of how to specify meta-models, models, and in-place model transformations in Maude. A model is represented in Real-Time Maude as a set of objects. The main idea of the semantics is to split timed model transformations into two parts: the triggering of a rule instance, and the ensuing execution of the triggered rule instance. When a rule instance is triggered, the system adds a new **ActionExec** object to the state, with information about the participating objects, the label of the action, and two timers denoting, respectively, the smallest time until the action can be executed and the maximum time remaining until the action must be executed. After the first of these timers has expired, and not later than the expiration of the later timer, and the objects involved in the action remain in the state, the action is executed: The objects in the rule instance’s left-hand side are replaced by the corresponding instance of the rule’s right-hand side.

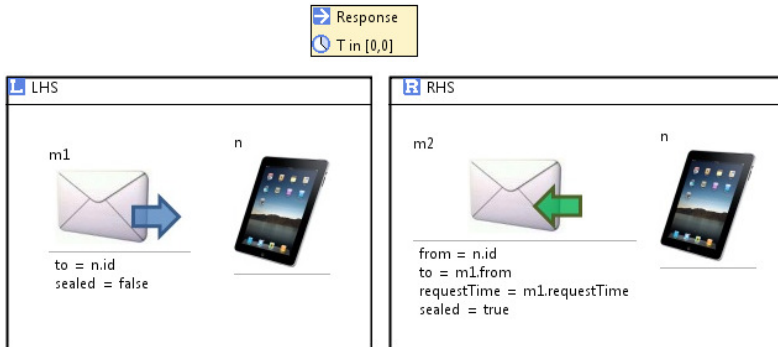


Fig. 10. Responding to a request

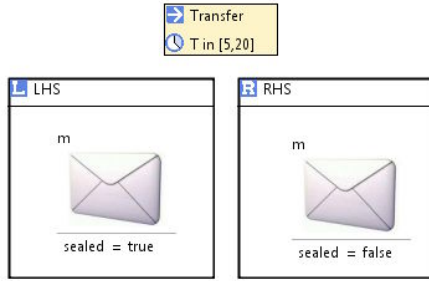


Fig. 11. Modeling message transfer

In addition to the rules treating the triggering and execution of rule instances, the Real-Time Maude semantics has the standard tick rule for object-oriented Real-Time Maude specifications:

```
cr1 [tick] : {MODEL} => {delta(MODEL,T)} in time T if T <= mte(MODEL) .
```

where `delta` increases the value of the system clock, modifies the system values according to the ongoing rules, and decreases the timers of the `actionExec` objects. As usual, the `mte` function gives the value until an `actionExec` objects must be executed, and is 0 whenever a non-soft atomic rule instance that does not already have an `actionExec` object in the state is enabled.

Formal Verification of e-Motions Model Transformations. Model transformations can be simulated visually in the e-Motions tool, which is available as an Eclipse plug-in. For reachability and LTL model checking analyses, however, the tool generates the corresponding Maude model, and the analysis has to be done at the Maude level, albeit with support for the main Real-Time Maude time sampling strategies. The operator `<<_>>` can be used to define analysis commands without knowing the Real-Time Maude representation of an e-Motions model. The term `<< ocl-expr ; model >>` evaluates the OCL expression `ocl-expr` in the model `model`. To search for a model that satisfies the OCL expression `ocl-expr` and is reachable from an initial model `myModel`, we use the command

```
search [1] init(myModel) =>* {MODEL:@Model} in time T:Time
  such that << ocl-expr ; MODEL:@Model >> .
```

For example, the following command checks whether a model with a node whose `rtt` attribute is set to a value outside the interval `[10,40]` can be reached:

```
search [1] init(rtttpModel) =>* { MODEL:@Model } in time T:Time
  such that
    << Node@rttp.allInstances -> exists ( n |
      (n.rtt@OCLsf > 40) or (n.rtt@OCLsf < 10)) ; MODEL:@Model >> .
```

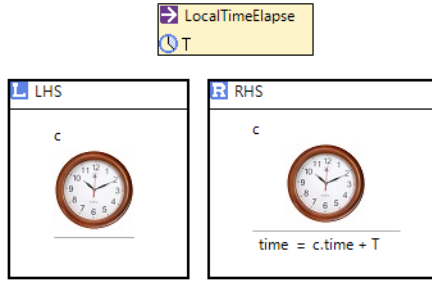


Fig. 12. Ongoing rule modeling the advance of the clock

8 A Modeling Language for Handset Software

In [1], Musab AlTurki and researchers at DOCOMO USA Labs describe a simple but powerful specification language, called \mathcal{L} , that is claimed to be well suited for describing a spectrum of behaviors of various software systems. The language provides flexible SDL-inspired timing constructs that yield a more expressive language for timed behaviors than Erlang [19], since some nested timing patterns, which can be expressed in \mathcal{L} , are not expressible in Erlang [1].

The language has an expression language, imperative features for describing sequential computations, and asynchronously communicating processes that can be dynamically created or destroyed. It is worth remarking that just the dynamic process creation places \mathcal{L} outside the class of systems that can be represented as timed automata; so does its expression language and imperative features which make \mathcal{L} Turing-complete.

The Language \mathcal{L} . A command in \mathcal{L} can be an assignment statement, a variable declaration using a **let** statement, or the usual conditional or while loop statements. The language also has a few process-level commands, which include creating a new process, destroying the current process, and sending and receiving messages to/from a process. The timed behaviors can be specified with two *timer* constructs: **set** starts a timer, and **release** drops a timer. The expiration of a timer triggers a *signal* that can be checked by a **receive** command.

Example. Figure 13 shows an example of an \mathcal{L} specification that specifies the “initiator” in our round trip time example. The variable *r_{tt}* denotes the computed RTT value, and *clock* denotes the time since the last request was sent by the initiator. Since \mathcal{L} , as given in [1], does not have clocks, we encode this clock by increasing its value by one whenever the *tickTimer* expires. A round of the protocol starts when the initiator receives a “computeRTT” message. It then resets *r_{tt}* and *clock* and starts the *tickTimer* and the *resendTimer*. As long as an RTT value is not found, the node waits for signals/messages from the timers or the environment. If it receives a signal that its *tickTimer* expired, it increases the *clock* and resets the *tickTimer*. If it receives an “RTTresponse” message, its

sets *rtt* to the current value of *clock* and turns off the two timers. If the *resendTimer* expires, the initiator sends another request, sets the *clock* to zero, and resets its *resendTimer*. Other signals/messages are ignored.

```

module initiator is
  let rtt = 0, clock = 0 in
    while (true)
      receive "computeRTT" in {
        send "RTTrequest" to "responder";
        set tickTimer to 1;
        set resendTimer to 60;
        rtt := 0; clock := 0;
        while (rtt == 0)
          receive m in {
            case tickTimer : {clock := clock + 1; set tickTimer to 1};
            case "RTTresponse" :
              {rtt := clock; release tickTimer; release resendTimer};
            case resendTimer :
              {send "RTTrequest" to "responder"; clock := 0;
               set resendTimer to 100};
            default : { };
          }
      }

```

Fig. 13. An \mathcal{L} specification of the initiator of the round trip time protocol

Real-Time Maude Semantics. The formal semantics of \mathcal{L} is given as an object-oriented Real-Time Maude specification, in which each process is represented by an object

$\langle id : \text{Process} \mid name : x, env : env, program : p, timers : timers, msgs : msgs \rangle$

where *env* is the current valuation of the variables, *p* is the statements the process should execute, *timers* is a set of active timers, each of which is represented as a name/time pair, and *msgs* is a queue of incoming messages. Program statements are executed as instantaneous rewrite rules. For example, the following rewrite rule specifies the execution of the statement **set** *x* **to** *t*, which sets a timer *x* to *t*:

```

r1 [setTimer] :
  < P : Process | env : E, program : set x to t ; S, timers : TIMERS >
=>
  < P : Process | program : S, timers : (TIMERS ; timer(x, eval(t, E))) >.

```

This rule adds a new timer/time-to-expiration pair to the set of timers, where the expression *t* is evaluated in the environment *E*. The tick rule is the “standard” tick rule for object-oriented systems, augmented with a test for whether a statement is enabled and must be executed before time advances:


```

cr1 [tick] :
  {SYSTEM} => {delta(SYSTEM,T)} in time T
  if T <= mte(SYSTEM) /\ inactive(SYSTEM) .

```

where, as expected, `mte` returns the smallest timer value in the system, and `delta` decreases all timer values in the state according to elapsed time.

Formal Analysis Support. The Real-Time Maude definition of the semantics of \mathcal{L} also provides a prototype analysis tool for \mathcal{L} , and the paper [1] shows model checking of fairly complex temporal logic properties. However, there does not seem to be any support for automatically translating \mathcal{L} specifications into Real-Time Maude models. Therefore, the user must define the initial state `init` corresponding to the Real-Time Maude representation of the initial state and program of the specification; the search for bad RTT values is then done in Real-Time Maude in the usual way:

```

(utsearch init =>* {< P:Pid : Process | env : ('rtt |-> N:Nat) VAL:Valuation >
                  REST:Configuration}
  such that N:Nat > 0 and (N:Nat < 10 or N:Nat > 40) .)

```

9 Concluding Remarks

This survey paper has shown how Real-Time Maude has been used to provide a formal semantics and formal model checking capabilities for a wide range of modeling languages for embedded systems. These modeling languages include (subsets of) an industrial modeling standard for avionics and automotive systems, a powerful synchronous graphical modeling language used in industry, two very different approaches to add timed behaviors to EMF-based model transformation frameworks, and an imperative language for handset applications.

To give the reader an impression of the source modeling languages and their complexity, as well as of their Real-Time Maude semantics, I briefly introduced the modeling languages and provided a small example for each of them. A main conclusion that can be drawn these languages is that they are based on Turing-complete expression and transition languages, and that as part of defining the semantics of a modeling language, there is a need to define an interpreter for expressions and local transitions. This is done in the formal semantics in this paper by a combination of rewrite rules and the use of equationally defined functions to compute the effect of single transitions, fixed points, etc. This need to define an interpreter for Turing-complete parts should make it hard or impossible to define the semantics of such languages not only in languages based on timed automata or timed Petri nets, but also in more expressive real-time languages such as IF [12] and BIP [6].

The paper has also described how a model could be formally analyzed in Real-Time Maude, preferably by automatically synthesizing a Real-Time Maude model from the given model, and using predefined atomic propositions that

makes it easy to define suitable analysis queries. This enables a formal model engineering that combines the convenience of modeling using an intuitive domain-specific modeling language with automated formal analysis. Again, a necessary condition for this to be possible is that the expressive (Real-Time) Maude system provides model checkers despite being based on Turing-complete formalisms.

Some distinguished modeling languages for embedded systems, such as SDL, timed extensions of UML, and Erlang, have not yet been given a Real-Time Maude semantics; however, subsets of SDL have been given a *timed rewriting logic* semantics [47] and untimed core Erlang models have been giving rewriting logic semantics [31]. Since this survey paper focuses on modeling languages that are also aimed at embedded systems, I have not included the very nice work by AlTurki and Meseguer on the formal semantics, analysis, and distributed implementation for the Orc web services orchestration language [2] or the Real-Time Maude formalization of a timed extension of the Creol language [7].

Acknowledgments. I thank the Festschrift editors for giving me the opportunity to honor Carolyn Talcott. I have had the great pleasure to work together with Carolyn from my early days as a Ph.D. student, when we worked together on using a prototype of Maude to invalidate a communication protocol in what may well have been the first time Maude was used to break a protocol [16], to our present collaboration on defining the formal semantics of Timed Rebeca. Furthermore, the work presented in this paper, in particular the work on the actor-based Ptolemy II language, builds on Carolyn's work on the formal semantics of actors as well as on her work as the leader of the Maude group at SRI. I am grateful to you, Carolyn, for your kindness, encouragement, and excellent scientific advice during my years as a researcher. Finally: thanks for a very pleasant day (re)discovering my home town, Budapest, together during ETAPS 2008. Happy Birthday, Carolyn!

I am also grateful to José Meseguer for our work together on most of the work summarized in this paper. I also thank the other collaborators on these efforts: Abdullah Al-Nayeem, Kyungmin Bae, Artur Boronat, Paco Durán, Edward Lee, and Stavros Tripakis, as well as Musab AlTurki for discussions on the language \mathcal{L} . Thanks are also due to the anonymous reviewer for many helpful comments on an earlier version of this paper. Finally, I gratefully acknowledge financial support from The Research Council of Norway through the Rhythm project.

References

1. AlTurki, M., Dhurjati, D., Yu, D., Chander, A., Inamura, H.: Formal specification and analysis of timing properties in software systems. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 262–277. Springer, Heidelberg (2009)
2. AlTurki, M., Meseguer, J.: Real-time rewriting semantics of Orc. In: Proc. PPDP 2007. ACM, New York (2007)
3. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* 126(2), 183–235 (1994)

4. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM 2011. LNCS. Springer, Heidelberg (to appear, 2011)
5. Bae, K., Ölveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Science of Computer Programming* (to appear, 2011), doi:10.1016/j.scico.2010.10.002
6. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. *IEEE Software* 28(3), 41–48 (2011)
7. Bjørk, J., Johnsen, E.B., Owe, O., Schlatte, R.: Lightweight time modeling in Timed Creol. In: Proc. RTRTS 2010 (2010)
8. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 18–33. Springer, Heidelberg (2009)
9. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
10. Boronat, A., Meseguer, J.: Algebraic semantics of OCL-constrained metamodel specifications. In: Oriol, M., Meyer, B. (eds.) TOOLS EUROPE 2009. LNBIP, vol. 33, pp. 96–115. Springer, Heidelberg (2009)
11. Boronat, A., Ölveczky, P.C.: Formal real-time model transformations in MOMENT2. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 29–43. Springer, Heidelberg (2010)
12. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)
13. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. *Theoretical Computer Science* 360(1-3), 386–414 (2006)
14. Cerone, A., Maggiolo-Schettini, A.: Time-based expressivity of time Petri nets for system specification. *Theoretical Computer Science* 216(1-2), 1–53 (1999)
15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
16. Denker, G., García-Luna-Aceves, J.J., Meseguer, J., Ölveczky, P.C., Raju, Y., Smith, B., Talcott, C.: Specification and analysis of a reliable broadcasting protocol in Maude. In: 37th Annual Allerton Conference on Communication, Control, and Computation. University of Illinois (1999)
17. Ding, H., Zheng, C., Agha, G., Sha, L.: Automated verification of the dependability of object-oriented real-time systems. In: Proc. WORDS 2003. IEEE (2003)
18. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE* 91(2), 127–144 (2003)
19. Erlang home page, <http://www.erlang.org/>
20. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: Proc. ICECCS 2007. IEEE (2007)
21. Katelman, M., Meseguer, J., Hou, J.: Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 150–169. Springer, Heidelberg (2008)

22. Lepri, D., Ölveczky, P.C., Ábrahám, E.: Timed CTL model checking in Real-Time Maude (submitted for publication)
23. Lien, E., Ölveczky, P.C.: Formal modeling and analysis of an IETF multicast protocol. In: Proc. SEFM 2009. IEEE Computer Society (2009)
24. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96, 73–155 (1992)
25. Meseguer, J.: Rewriting logic as a semantic framework for concurrency: a progress report. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 331–372. Springer, Heidelberg (1996)
26. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
27. Meseguer, J.: Research directions in rewriting logic. In: Berger, U., Schwichtenberg, H. (eds.) Computational Logic, NATO Advanced Study Institute, Marktobendorf, Germany, July 29–August 6. NATO ASI Series F: Computer and Systems Sciences, vol. 165, pp. 347–398. Springer, Heidelberg (1998)
28. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 303–320. Springer, Heidelberg (2010)
29. Meseguer, J., Rosu, G.: The rewriting logic semantics project. *Theoretical Computer Science* 373(3), 213–237 (2007)
30. Miller, S.P., Cofer, D.D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. DASC 2009. IEEE (2009)
31. Neuhäuffer, M.R., Noll, T.: Abstraction and model checking of core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science* 176(4), 147–163 (2007)
32. Ölveczky, P.C.: Towards formal modeling and analysis of networks of embedded medical devices in Real-Time Maude. In: Proc. SNPD 2008. IEEE (2008)
33. Ölveczky, P.C.: Formal model engineering for embedded systems using Real-Time Maude. *Electronic Proceedings in Theoretical Computer Science* 56, 3–13 (2011)
34. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 47–62. Springer, Heidelberg (2010)
35. Ölveczky, P.C., Boronat, A., Meseguer, J., Pek, E.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude (2010), report
36. Ölveczky, P.C., Caccamo, M.: Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 357–372. Springer, Heidelberg (2006)
37. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science* 285, 359–405 (2002)
38. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science* 176(4), 5–27 (2007)
39. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation* 20(1-2), 161–196 (2007)
40. Ölveczky, P.C., Bevilacqua, V.: The Real-Time Maude tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008)
41. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design* 29(3), 253–293 (2006)

42. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. *Theoretical Computer Science* 410(2-3), 254–280 (2009)
43. Rivera, J.E.: *On the Semantics of Real-Time Domain Specific Modeling Languages*. Ph.D. thesis, Universidad de Málaga (2010)
44. Rivera, J.E., Durán, F., Vallecillo, A.: On the behavioral semantics of real-time domain specific visual languages. In: Ölveczky, P.C. (ed.) *WRLA 2010*. LNCS, vol. 6381, pp. 174–190. Springer, Heidelberg (2010), see also the e-Motions web page http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions
45. SAE AADL Team: AADL homepage (2009), <http://www.aadl.info/>
46. Srba, J.: Comparing the expressiveness of timed automata and timed extensions of petri nets. In: Cassez, F., Jard, C. (eds.) *FORMATS 2008*. LNCS, vol. 5215, pp. 15–32. Springer, Heidelberg (2008)
47. Steggles, L.J., Kosiuczenko, P.: A timed rewriting logic semantics for SDL: A case study of the alternating bit protocol. In: *Proc. WRLA 1998*. *Electronic Notes in Theoretical Computer Science*, vol. 15, Elsevier (1998)
48. Viry, P.: Equational rules for rewriting logic. *Theoretical Computer Science* 285, 487–517 (2002)
49. Wang, F.: Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE* 92(8), 1283–1307 (2004)

Computational Biology: A Programming Perspective

Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen,
and Søren Bjerregaard Vrist

Department of Computer Science (DIKU), University of Copenhagen, Denmark
{hartmann,neil,simonsen,seet}@diku.dk
DIKU URL: <http://www.diku.dk>

Abstract. Computation via biological devices has been the subject of close scrutiny since von Neumann’s early work some 60 years ago. In spite of the many relevant works in this field, the notion of *programming* biological devices seems to be, at best, ill-defined. While many devices are claimed or proved to be computationally universal in some sense, the full step to a bona fide programming language is rarely taken, and one question is noticeable by its absence: If the device is universal, *where are the programs?*

We begin with an extensive review of the literature on programming-related biocomputing; and briefly identify some strengths and shortcomings from a programming perspective. To show concretely what one could see as programming in biocomputing, we outline (from recent work) a computation model and a small programming language that are biologically more plausible than existing silicon-inspired models. Whether or not the model is biologically plausible in an absolute sense, we believe it sets a standard for a biological device that can be both universal *and programmable*.

1 Context

The terms “biocomputing” and “systems biology”, taken in their broadest senses, span many fields and areas of research: biology, chemistry, physics, mathematics, electrical engineering and computer science among others. Two major subareas:

- Computer modeling of biological and biomolecular processes
- Biological hardware for computation

The terms “biomolecular computation” and “biomolecular computational model” occur with both meanings in the literature (see Section 4), sometimes referring to the one and sometimes the other subarea. This paper emphasises the second. More precisely: we take a *synthetic* viewpoint, concerned with building things as in the engineering and computer sciences. This is in contrast to the inevitable and ubiquitous *analytic* viewpoint common to the natural sciences, concerned with finding out how naturally evolved things work.

We ask: What can be *done or built or constructed*; and not: how does nature work? (Caveat: just as in engineering, one will need to understand nature's cause-and-effect sufficiently well to be able to *modulate* it, i.e., to use nature to effectively serve our purposes.)

1.1 A Theme: Biological Problem-Solving

Our main aim is thus to use the biological world as a computational medium. From a programming perspective, the main goal is problem-solving by writing programs. This section thus refers to solving problems *by biology*, not solving problems *about biology* or *in biology*.

Given: a computational problem. **To find:** a biological solution. Further, problem-solving by a program means finding a *general solution* (and not just one run of one algorithm on one input data instance).

A top-down approach is to devise a model of computation that

- satisfies requirements for computer science/engineering
- that could conceivably be realised in a biological medium
- in which programs are clearly visible, and programming can be done
- is a framework in which it is possible, at least in principle, to say when a problem has been solved

A bottom-up approach is to develop *biological toolkits and environments* in which such engineering can be done. Examples include: synthetic biology; DNA computing; membrane computing; and other computation models, e.g., peptides, whole cell computing, bio quantum computing and many others seen in the literature. Some of these toolkits and environments are already well-started and others just beginning.

A reservation: our impression is that many exciting papers have been written on computational biology, but there seem to be relatively few and small actual realisations with concrete, reproducible, automatable results. There are many possible reasons for this: intractability of physical biological media, difficulty of carrying out experiments, difficulty of measuring success, liberal funding, journalistic appeal, etc.

1.2 Anticipated Difficulties in Practical Computational Biology

Our main goal is to find a way to incorporate programming concepts into a biological context. In this paper we will sidestep some anticipated difficulties including

Instrumentation: In a laboratory setting (with test tubes, Petri dishes, chemical solutions, . . .) one would need to

1. put an experimental set-up into a known initial state
2. put input values into it
3. recognise when a computation has finished (or run “long enough”)
4. read output values out from it

Although these challenging problems are important, our approach is synthetic and top-down. Future work would be to integrate our approach with a bottom-up approach, and to realise the ideas in a biochemical laboratory.

Stochastic factors: Another dimension of difficulties that we will abstract away from is how to deal with the nondeterminism that seems built-in to nature, for example Brownian motion, noise (in an engineering sense), quantum state transitions, and many others.

Researchers have worked to alleviate these real problems, studying error-correcting processes from a theoretical viewpoint, and in nature as well (e.g., DNA repair).

1.3 About the Scientific Method and Automatability

From the analytic viewpoint: natural science is strongly based on the principle of *reproducibility*: if someone reports something in a scientific report, it should be possible for the readers (if sufficiently determined) to reproduce the results by running the same experiments themselves. This approach has had many successes, and some highly visible failures (e.g., cold fusion).

From the synthetic viewpoint: reproducibility is *relatively easy to achieve* in computer science. Computers have been carefully designed and built to be deterministic and predictable (else there is a bug in the computer's construction), so reproducing a result can consist of just running the program again.

Reproducibility is *much harder* in biological frameworks. There is typically no program to run, but rather long and complex experimental processes involving significant hand work, and human participation and/or evaluation. Further, randomness is real: what works in one experimental setting may not work in another, for reasons mentioned above. Much research has been aimed at just this problem: ensuring predictable behavior; and many reported experimental results seem to suffer from it.

A measure of progress in both analytic and synthetic work is *automatability*. A well-known real-world example: the rapid advances made in DNA analysis, e.g., for identification of people in legal or criminal settings. In computer science and engineering, the goal is a framework to make experimentation automatable so an experiment can be carried out a hundred or a thousand times. Further, there is an a priori desired known relation between input and output of the experiment: a relation that can be reliably expected to hold in all conceivable instances of just this experiment (else the hypothesis/principle/program is insufficient, incomplete, or just plain wrong).

This approach stands in great contrast to the analytical framework. Phrased somewhat strongly: if the expected output is fully known in advance, one is working not on an experiment, but on development rather than research. (To be fair: the synthetic approach is aimed at development, not at discovering new scientific truths.)

2 Motivation

2.1 Biocomputation

An initial hope biocomputing was to overcome some of the limiting factors of conventional microchip-based computers, similar to the hopes for quantum computation: can we break through the barrier of polynomial running time [5,1,17]?

A dream: instead of a computer built from microchips and based on electrical signals, a biocomputer would run on “hardware” based on biological materials with some biochemical or biological equivalent of electrical signals. The hope is that a completely new paradigm might yield an advantage over conventional computing by putting radically different constraints on computation. Advantages may arise from the mere difference of scale of materials, and complexity of computation possible for basic blocks of a biomolecular computer.

Research in the 1990s focused on exploring such possibilities. Adleman made a groundbreaking proof of concept in 1994: a DNA based computer that solved an instance of a special case of the traveling salesman problem with 7 cities [1]. Several other possible advantages of a biocomputer were theorized—including biology-inspired properties such as autonomy, self-assembly, self-repair, high information density [57,54], and advantages due to inherent parallelism:

In a world where parallel computing is in focus, molecular interactions like communication over noisy media and load balancing are inherent in the structure of molecules refined by evolution.

Garzon and Deaton, [57]

As many different molecules can be synthesized at low cost and exist abundantly around us [57], natural resources are plentifully available for biocomputation. Another great potential lies in the fact that a biocomputer can in principle interact directly with other biological material, including the cells in the human body [11]. Numerous researchers [23,114,9,7] mention the possibility for direct interactions with live cells in one way or another.

Some Remarks by Benenson [11] First, on the issue of generality:

Unlike a silicon computer that can be reprogrammed with a few hits on a keypad, a biomolecular computer is really a set of design tools which, when provided with a description of a computational task, generates a blueprint of a molecular network that can implement this task. One important challenge is to make sure that these tools are flexible enough to enable a sufficient variety of tasks.

Second, on universal biological machines:

In theoretical computer science this problem was solved by the invention of universal models of computation, such as the Turing machine.

Although it seems unlikely that similarly universal approaches could be realized with biomolecular building blocks anytime soon, biocomputer architectures could, and should, aspire to at least some degree of generality in the computational sense.

2.2 Biomolecular Algebras and Calculi

Formalisms have been devised to do computer modeling (from an analytical viewpoint) of observed biomolecular processes. Examples include the κ -calculus [39,36], BioAmbients [100], Biochemical Ground form [26,25], Strand Algebra [22], and Pathway Logic [123,31].

A natural next step is to think of these formalisms synthetically, e.g., to reduce the need for expensive laboratory experiments.

2.3 Programming

In research on biomolecular computation the words “programming” and “calculus” are often used; for example, [65,9,57,96] all mention programming as part of their title. *This is not the same as the “programming” familiar to computer scientists: writing a program in a programming language.* Instead, the papers cited use the word “programming” to describe the process of designing biological devices in a setup that will carry out *one* computation or simulation of biomolecular interactions, and not a generic program that can be run on many inputs.

Program Text 1.1. Program that appends two lists, in Standard ML syntax

```
fun append (a::as) bs = a::append as bs
| append [] bs = bs
```

For example, take a simple “append” program as in Program text 1.1. Any computer scientist would recognize this as a program. Compare this with, for example, the κ -calculus from [39,36]:

κ is a formalism for modeling molecular biology where molecules are terms with internal state and with sites, bonds are represented by names that label sites, and reactions are represented by rewriting rules. For example, $EGFR[tk^0](1^z)$ represents a molecule of species $EGFR$ that is not phosphorylated - the internal state tk is 0 - and that is bond to another molecule - its site 1 is labeled with a name z .

and a κ rule as seen in Figure 1. In our opinion, the connection to conventional programming, in the sense of solving a problem or writing an algorithm, is not obvious.

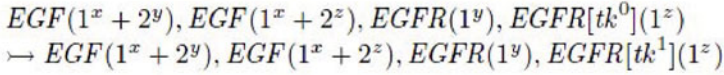


Fig. 1. κ rule defining the first step of *Receptor Tyrosine Kinase* from [44]

Figure 2 shows another example, used by Yin et al in [133] to illustrate “Pathway programming”. Here, the “programming” is in the process of designing the DNA to react and function in a specific way, and is *not* analogous to Program text 1.1.

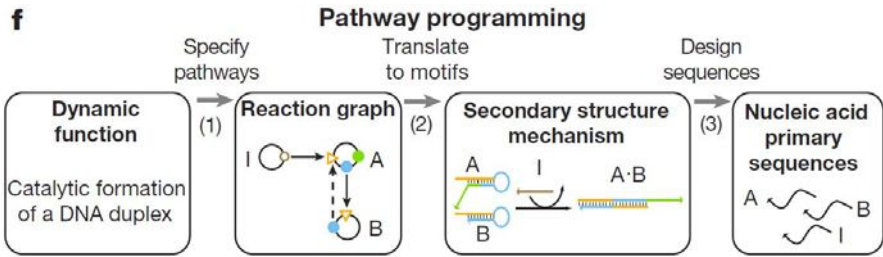


Fig. 2. From [133]. Images showing steps to construct a “molecular executable”.

Figure 3 illustrates the molecular implementation of a propositional logic query as presented by Ran, Kaplan and Shapiro in [98]. Both the “program” and the input are represented as specific molecular “objects”:

A compiler translates facts, rules and queries into their molecular representations and subsequently operates a robotic system that assembles the logical deductions and delivers the result. (Ran, Kaplan and Shapiro [98])

This version of biomolecular programming is closer to the idea of programming as in Program text 1.1, but it still conflicts with a key concept of conventional programming, e.g., as we say later:

A program is software, not hardware. Thus a program should itself be a concrete data object that can be replaced to specify different actions.

The central question: can program execution take place in a biological context? Evidence for “yes” includes many analogies between biological processes and the world of programs: *program-like behavior*, e.g., genes that direct protein fabrication; “switching on” and “switching off”; processes; and reproduction.

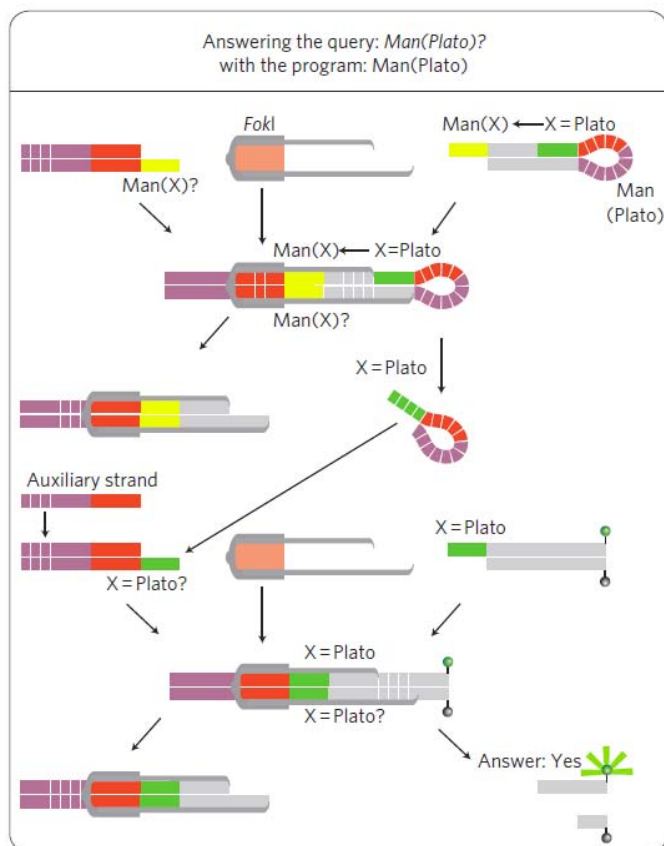


Fig. 3. Ran, Kaplan and Shapiro [98]: the molecular representation of a simple logic query

3 Biochemical Universality and Programming

We begin in Section 4, by evaluating some established results on biomolecular computational completeness from a programming perspective; and then constructively provide an alternative solution in Section 5. The new model seems biologically plausible, and usable for solving a variety of problems of computational as well as biological interest.

3.1 Baseline: Program Execution

What do we mean by a program (roughly)? An answer: a *set of instructions* that specify a *series (or set) of actions on data*. Actions are carried out when the

instructions are executed (activated,...) Further, a program is software, not hardware. Thus a program should itself be a *concrete data object that can be replaced* to specify different actions.

Program execution: we write $\llbracket \text{program} \rrbracket$ to denote the meaning or net effect of running **program**. A program meaning is often a function from data input values to output values. The **program** is *activated* (run, executed) by applying the semantic function $\llbracket _ \rrbracket$. The *task of programming* is, given a desired semantic meaning, to find a program that computes it. Some mechanism is needed to execute **program**, i.e., to compute $\llbracket \text{program} \rrbracket$. This can be done either by hardware or by software.

3.2 Turing Completeness of Computational Models

Turing completeness of a computation framework is typically shown by *reduction* from another problem already known to be Turing complete. Notation: let L and M denote languages (biological, programming, whatever), and let $\llbracket p \rrbracket^L$ denote the result of executing L -program p , for example an input-output function computed by p . Say that language M is at least as powerful as L if

$$\forall p \in L\text{-programs} \exists q \in M\text{-programs} (\llbracket p \rrbracket^L = \llbracket q \rrbracket^M)$$

A popular choice (for proving universality) is to let L be some very small Turing complete language, for instance Minsky register machines or two-counter machines (2CM). The next step is to let M be a biomolecular system of the sort being studied. The technical trick is to argue that, given any L -instance of (say) a 2CM program, it is possible to construct a biomolecular M -system that faithfully simulates the given 2CM. This discussion brings up a central issue:

Simulation as Opposed to Interpretation. Arguments to show Turing completeness are (as just described) usually by *simulation*: For each problem instance (say a 2CM program) one constructs (using any available method) a biomolecular system that solve the problem. However in many papers the construction of the simulator is done by hand by the author, and each problem instance require a new hand coded simulator. In effect the existential quantifier in $\forall p \exists q (\llbracket p \rrbracket^L = \llbracket q \rrbracket^M)$ is computed by hand. This phenomenon is clearly visible in papers on cellular computation models: completeness is shown by simulation rather than by interpretation.

In contrast, Turing's original "Universal machine" realises p 's computation by means of *interpretation*: a stronger form of imitation, in which the existential quantifier is realised by machine. Turing's "Universal machine" is capable of executing an arbitrary Turing machine program, once that program has been written down on the universal machine's tape in the correct format, and its input data has been provided. Our research follows the same line, applied in a biological context: we show that simulation can be done by general interpretation, rather than by one-problem-at-a-time constructions as mentioned by Benenson in [11], quoted in Section 2.1 of this paper.

3.3 Programs in a Biochemical World

Our goal is to express programs in a biochemical world. Programming assumptions based on silicon hardware must be radically re-examined to fit into a biochemical framework. We briefly summarize some qualitative differences.

- **There can be no pointers to data**, i.e., no addresses, links, or unlimited list pointers. In order to be acted upon, a data value must be *physically adjacent* to some form of actuator. A biochemical form of adjacency: a chemical bond between program and data.
- **There can be no action at a distance**: all effects must be achieved via chains of local interactions. A biological analog: *signaling*.
- **There can be no nonlocal control transfer**, e.g., no analog to GOTOs or remote function/procedure calls. However control loops can be accepted, provided the “repeat point” is (physically) near the loop end. A biological analog: *a bond* between different parts of the same program.
- On the other hand there exist available **biochemical resources** to tap, i.e., free energy so actions can be carried out, e.g., to construct local data, to change the program control point, or to add local bonds into an existing data structure. Biological analogs: Brownian movement, ATP, oxygen.

The first three points above are very different from basic architecture currently used in a silicon based computers: the addressing unit and the address decoder are built around the ideas of unbounded pointers to both data and program: “random access memory” and control transfers.

4 History and Literature Review

4.1 History

Our brief survey of historical developments is not in any way exhaustive, but should “scratch-the-surface” enough to give a feeling for the different aspects of the field. Some good overviews for further reading include Benenson [10], and Kari and Rozenberg [76,78].

Background. Nature-inspired models of computation can demonstrate both how nature inspires computation models, and how complexity can emerge from seemingly simple rules in nature. Examples include cellular automata, neural computation, evolutionary programming, swarm intelligence, artificial life, membrane computing, and amorphous computing [78].

Cellular Automata

These were considered in the early 1950s as a possible idealized model of biology [131, page 48]. The work of Von Neumann and Burks on a “universal constructor” [93] was inspired by self-reproduction both for biology and computers.

Roughly, a cellular automaton is a grid of cells, typically 2-dimensional, in which each cell is in one of a fixed finite set of states. First, the initial state ($t = 0$) is set up by choosing an initial assignment of states to cells. The next state of any cell, at time $t + 1$, is determined by applying a predetermined transition function to current state of the given cell and its immediate neighbors. For example “Any white cell with exactly three black-neighbors becomes a black cell.”¹.

The perhaps most famous cellular automaton is *Game Of Life*, the two-state two-dimensional cellular automaton by John Conway [55] which has exactly four rules. Figure 4 illustrates² the initial generation and four following generations of a *Game Of Life* cellular automaton. In [15] it is shown that universal computation is possible with the *Game Of Life*, and an actual implementation was demonstrated in [28]. Concretely, this means that for any Turing machine, an initial state exists, from which the *Game Of Life* will faithfully simulate the Turing machine.

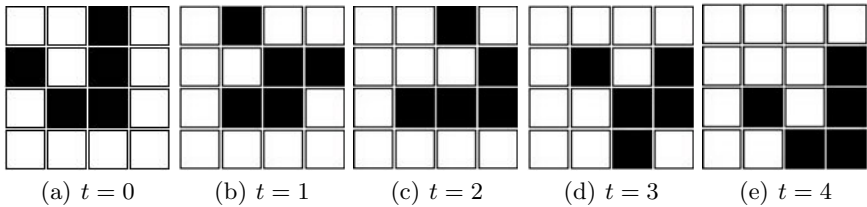


Fig. 4. Five steps of a *Game Of Life* life form called “glider” [15]. Notice that the last image contains the same shape as the first image. This means that the glider will continue to fly until it hits something.

Cellular automata are homogeneous, parallel, and all interactions are local. Further, it is possible to obtain natural physical properties such as reversibility and conservation laws by choosing local update rules properly [76]. This connection would be especially interesting if it is possible to let nature simulate cellular automata, and thereby prove that nature can obtain the Turing completeness properties as demonstrated for cellular automata [76]:

...then perhaps we eventually succeed to harness physical reactions of microscopic scale to execute massively parallel computations by running a computationally universal CA. ... While such truly programmable matter may be decades away, its potential is great

Wang Tiles

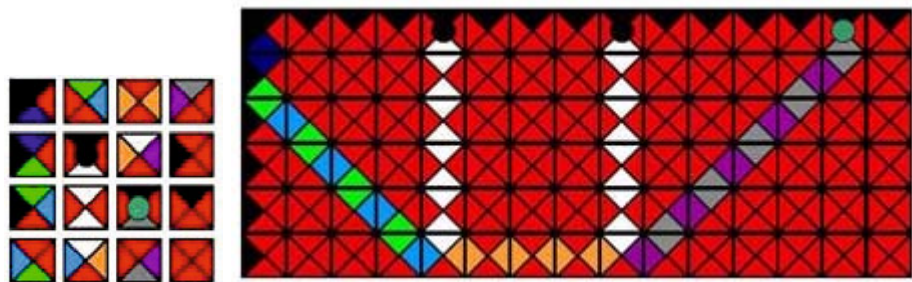
These were introduced by Hao Wang in 1961 [125]. A tiling system is a finite set of tile types. The tiles are equal-sized squares, with a color on each edge. A tiling is a two-dimensional arrangement (without rotation or reflection of the tiles) of

¹ Rule from *Game Of Life*.

² Images generated via <http://www.onderstekop.nl/GoF.php> - 2010-02-07

tiles, such that the touching edges of adjacent tiles must have the same color. Figure 5(a) shows a set of Wang tiles and 5(b) shows a tiling. It is undecidable, given a finite set of Wang tiles, whether they can tile the plane [14]: any Turing machine can be emulated by Wang tiles in such a way that if the tiles tile the plane, the Turing Machine never terminates [103].

Figure 5(b) shows³ an encoding of Wang tiles that represents the calculation of $5 + 9$ with the result of 14.



(a) Wang tiles used (b) The tiles types assembled into a pattern representing $5 + 9 = 14$. Notice the three special tiles with the circle in the middle in the top row. The two black ones represent the input numbers, and the green one represents the answer. For this set of tiles, addition with input and output is the only possible way to assemble them.

Fig. 5. Based on a diagram from “Tilings and Patterns” by Grunbaum and Shephard

Biocomputers. The idea of a biocomputer was first suggested by Feynman in 1959, and theoretically discussed through the eighties and beginning of the 1990s [54], for example in the visionary paper by Michael Conrad, “On design principles for a molecular computer” [33]. It was argued that the natural concurrency present in biocomputing could be used to solve difficult combinatorial problems, e.g., NP-complete problems.

In a landmark 1994 paper, Adleman demonstrated how a biological system can use reactions to solve a Directed Hamiltonian Path problem with DNA hybridization [1].

The advantage of such a “biocomputer” is potentially massive parallelism: the theoretical ability of DNA to contain data is 10^{21} bits per gram, and the energy requirement is low: 2×10^{19} operations are theoretically possible per joule [54]. Lipton found theoretical methods for solving NP-complete problems in general using DNA computers [82], and several enhancements were provided, e.g. Adleman et al discovered the solution to a 20-variable 3-SAT problem [18], and Winfree et al devised ways to use single strand DNA as memory [107,108].

Based on this research, several ways were devised to compute other search problems such as linear optimization problems [126], and Boolean search type satisfiability such as 3-coloring [27, sec 2.2]. The parallels between DNA-processing

³ Images in Figure 5 from <http://seemanlab4.chem.nyu.edu/XOR.html>

reactions and “forbidding/enforcing systems” further led to the usage of DNA-processing to solve these kinds of problems as well [3]. DNA-based arithmetic was developed in [62,27]. Further developments include the breaking of DES encryption with DNA [17], DNA Computer based on Biochips [137], playing TicTacToe against DNA [90], DNA encoded with finite automata for medical purposes [9,7], and molecular implementation of simple logic programs [98].

However, while the generation of all paths in Adleman’s original experiment took less than a second, it took weeks of manual lab work to extract the potential solutions from the DNA cocktail. The method is further hampered by the exponential growth of the size of the solutions: The weight of the DNA needed to represent the state space explored to find the solution of a 200 city Hamiltonian Path Problem would exceed the weight of the Earth itself [95].

Why DNA? Though the term “biocomputer” does not specify the specific computation medium, by far the most common medium is Nucleic Acids: the NA in DNA and RNA. This is possibly due to the fact that nucleic acids have a predictable “base pairing property” which makes them a powerful tool for biomolecular engineering [136]. In practice it seems that DNA and RNA behave “more deterministically” than other biological media (perhaps for evolutionary reasons).

Thus in the literature the terms “biocomputation” and “biocomputers” are often used interchangeably with “DNA Computer.” (Exceptions: [11] uses RNA; and there are articles on “peptide computing”, “whole cell computing” and “Quantum Molecular Computing”: [71], [118], [34].) Following the general trend, in this section we will assume unless stated otherwise that a biocomputer is based on DNA.

4.2 Computational Universality of DNA Computers

Universality and self-reproduction. Several articles argue “universality” by showing (in several ways) that Turing machines can be simulated. Oddly, articles we have seen do not mention self-reproduction, even though DNA is the key to biological self-reproduction. Further, self-reproduction was a major goal of von Neumann and Burks [93]. (That pathbreaking work did indeed show how to simulate a universal Turing machine; but this was only incidental, as a means to establish that cellular automata did not “reproduce” in some trivial way, e.g., as in the growth of crystals.)

Several different approaches have been taken to represent Turing machine operations by DNA operations or collection of operations (e.g., hybridization, annealing, ligation or polymerization). A partial list follows. The different articles use different terminology, notations and degrees of formalism, so we choose to quote their claims directly, rather than paraphrase and interpret their results:

- [12,13]: Discuss the theoretical possibility of RNA/DNA: “The enzyme may thus be compared to a simple tape-copying Turing machine that manufactures” its output tape rather than merely writing on it.
- [119] : Shows how to create “a non-deterministic Turing machine” (one *specific* Turing Machine at that)

- [5] Discusses a design method to simulate a Turing Machine via a technique for “text-insertion” that “provides a basis for implementing general Turing Machines”
- [104] : “ [we] propose an encoding for the transition table of a Turing machine in DNA oligonucleotides and a corresponding series of restrictions and ligations of those oligonucleotides that, when performed on circular DNA encoding of an instantaneous description of a Turing machine, simulate the operation of the Turing machine encoded in those oligonucleotides”
- [135] Proposes a new “splicing model” and “show that they have the same computational power as a Turing machine”
- [114]: A physical model (See figure 6) showing what a mechanical Turing machine on a biomolecular level could look like.
- [79,80] This reports that DNA mutagenesis “is theoretically universal by showing how Minsky’s 4-symbol 7-state Universal Turing Machine can be implemented”.
- [134] designs a complete “autonomous device capable of universal computation and universal translational motion”

Self assembly. Self assembly is a process whereby DNA can arrange itself in a shape without outside involvement. An example is the formation of double helix

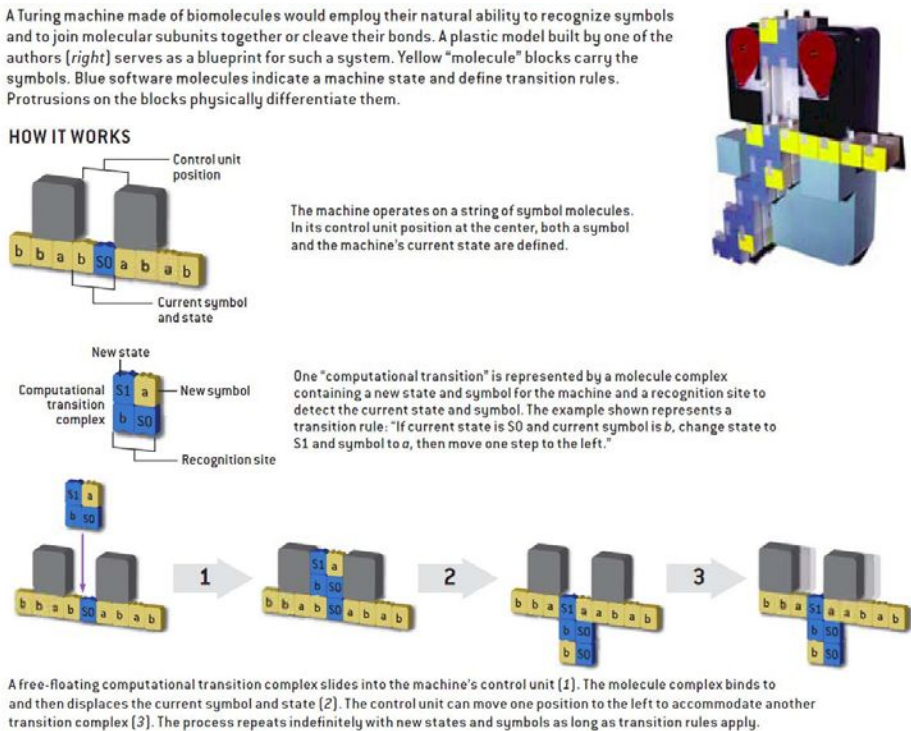


Fig. 6. From Shapiro and Benenson [113]

DNA from individual strands. It has been shown that DNA can do self assembly in a “programmable” way. Here “programmable” means that the final outcome of the self assembly process can be guided into an arbitrary two-dimensional shape, decided before the self assembly process begins.

The mechanism uses a concept of “sticky ends” of DNA structures [128,127]. An example is the famous nano scale “smiley” of [106].

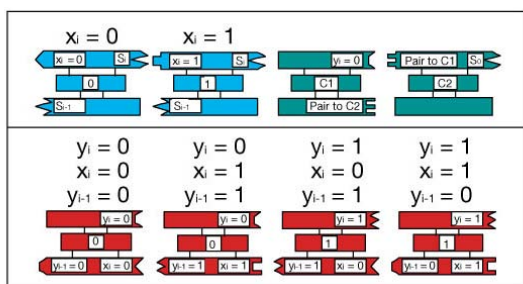
Based on the Wang tiling properties of DNA, [129] shows a way to simulate one-dimensional cellular automata, and [85] shows a way - via Wang tilings - to do a logical cumulative XOR. Figure 7 illustrates the principle of DNA tiling to realise a cumulative XOR gate. Figure 7(a) shows the building blocks used, and figure 7(b) shows how these can be tiled together to get the result of $Y = 1011$ when $X = 1110$.

Winfrey and coworkers used and extended the similarity between Wang tiles and DNA with sticky ends. Using known abilities of Wang tiles, they did a proof for universal computation abilities within self assembly of DNA [129]. Even though the theoretical computational power is unlimited, the practical usage of these powers is hampered by the difficulty of controlling the geometry and the specificity of binding interactions [105].

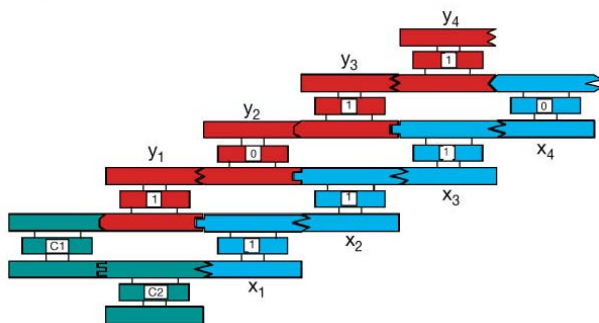
4.3 On Error Correction in Biological Contexts

In Section 1.2 we mentioned that “work has been done” to alleviate the stochastic factors when dealing with biomolecular computers. This section contains some references and quotes from the published research on the subject.

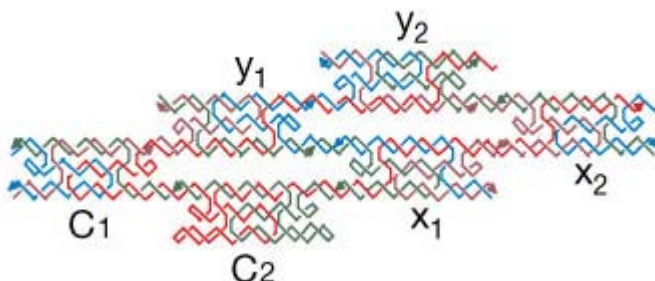
- Early work by Bennett [13,12] considered theoretical bounds in DNA computation: “The minimum error rate (equal to the product of the error rates of the writing and proofreading steps) is obtained when both reactions are driven strongly forward, as they are under physiological conditions.”
- Chiniforooshan et al [29]: “We achieve this by designing a single-input, single-output restoration gate”
- Roweis et al [107]: “we discuss several methods for achieving acceptable overall error rates for a computation using basic operations that are error prone.”
- Shlyahovsky et al [117]: “Finally, although the gate configuration consists of a complex structure composed of nine components, we find that the precise steps to set up the system and its activation at the defined conditions allow the result to be reproduced within a 15% error.”
- Winfree et al [120]: “The correctness of our systematic construction was predicated on several idealizations of DNA behavior, and it is worth considering the deviations that we would expect in practice”
- Rothmund et al [104] discuss in detail possible errors.
- Reif and LaBean [102] “Future Work” such as error correction and self-repair at the molecular scale“
- Murata and Stojanovich [91] discuss “error suppression and correction”



(a) Translation of the a DNA with sticky ends to tile-like building blocks. Notice the different shapes and the ends of the blocks only allowing specific pairings of blocks. X_i blocks encode the input number, where Y_i encodes the output number, all written in the middle.



(b) The dna-blocks assembled with X_i blocks put together to represent $X = 1110$. As the blocks can only be assembled in a specific way the result in is encoded in Y_i as $Y = 1011$



(c) Blocks as DNA structure

Fig. 7. Images from [85]

4.4 Process Calculus and Formal Methodology

Process calculi are concerned with providing formal specifications of concurrent processes.

An early process calculus was CCS as defined by [70], later followed by much research including the π -calculus [86,110].

Several calculi have been proposed for biomolecular modeling, both to describe natural biological systems, and to model how the entities in these system interact. Process calculi are the “other side of things” in relation to DNA computers and biocomputing: how do we provide an abstraction for biomolecular systems that allows for quick and effective sharing, comparison, and correction of scientific knowledge? Regev and Shapiro [99] published a landmark paper postulating the design of a “language for the cell” built on the insights of the articles [51] (where Fontana et al use the λ -calculus to describe natural systems) and [101] (where Shapiro et al use the π -calculus for biological modeling). Proposed calculi include⁴:

- π -calculus for biology [101] as just mentioned.
- BioAmbients - Extension of the above bio- π -calculus [100].
- Brane Calculi - based on membranes considered as active elements, with the whole computation happening on the membrane [20].
- CSS-R - Formal biology done in CCS-R [38].
- Bio-PEPA - Extension of the known process algebra PEPA [58], designed for biochemical networks [30].
- κ -calculus [39,36].
- Biochemical Ground Form [26,25].

The basic idea is to provide means, via formal calculus, to describe and design biomolecular setups with formal rules. In a larger setting this field is known as “Systems Biology.” At its core lie the fields of:

- Mathematical modeling, for example in terms of differential equations. This is a classical way to model biology.
- Computational modeling where the concepts of algebras [22,101,30], abstract interpretation [36,21] and process calculi [63] are used.

Computational modeling of biology, or the more apt “executable cell biology” [49], is a contemporary approach that uses computational power of conventional computers to reason about biomolecular systems.

Computational Power Often it is the case that these calculi are proven Turing complete, for example:

- π -calculus: proven by Milner by emulation of the λ -calculus [87].
- CCS [121].
- Brane Calculi [19].

⁴ Based on the survey by [63].

- κ -calculus [36].

As noted by Fontana et al. ([36] page 2), the fact that a biological representation language is Turing complete need not imply that biological reactions per se are Turing complete.

Applications to the building “biocomputers” or parts thereof is a popular recent research direction. For example, [22,96] use a theoretical algebra to as a tool for defining and designing biomolecular gates for specific setups. This provides a way to design a “biomolecular computer” from the basis of Boolean networks or Petri nets. From there, they compile into an intermediate “Strand Algebra” that can be translated further into a specific DNA Strand mixture. The authors claim that this mixture could potentially simply be mail-ordered from a bio company, at relative low cost.

That approach effectively combines the area of biocomputing with the techniques (and benefits) of biomolecular calculi. A computer scientist’s way to look at these biomolecular calculi could be as “a domain specific language for modeling biomolecular “things,” rather than a general purpose programming language like Python, C, or Java. The same connection can be seen in the title of [45] - A Domain-Specific Language for Programming in the Tile Assembly Model - for modeling “tiles” which can be used to direct DNA self-assembly into tile structure (similar to Winfree’s approach [127]).

4.5 Recent Developments

In recent years, it has become evident that biocomputing is perhaps not the answer to obtain fast solutions to computationally intractable problems (Parker, Cardelli [95,24]). Nevertheless, other potentials of a biocomputer in some form still interesting, e.g., self-replication, self-correction, massive parallelism, minuscule size, filters, ready availability, interfaces with other biological elements like humans. Several techniques to exploit the computational power of biomolecular systems are being explored, including cross-over from nano-research for tiling self-assembly [132,127], and a wide variety of specific proof-of-concept setups performing some specific computations.

Some recent research focuses on the abilities to create “biomolecular gates” in some form or the other. For example

- Papers [112,120,97,111,29,136] propose designs for DNA Strand displacement based gates (sequence based, enzyme free). Paper [29], building on the others, provides a design with four “desirable properties: scalable, time-responsive, energy-efficient and digital.” [29]
- Papers [117,48] manually design DNA gates for AND, OR, XOR by “DNA scaffolding”, report on the construction of a DNA based “library” of “DNAzymes”, and demonstrate how to create composite gates as well, partly based on the DNA Strand Displacement techniques as described above.

- Paper [60] proposes a way to create logic gates (enzymatic) that can be used repeatedly, as opposed to the use-once gates of DNA Strand Displacement as mentioned in the previous two items.
- Paper [11] reviews recent advances for using RNA as logic gates and the ability to do “molecular logic”. It describes the usage of RNA for computation in yeast and mammalian cells and notes that future work needs to address the bioengineering foundations of building these systems.
- Paper [81] reports on a successful coupling of molecular based logic gates, with silicon based microchips used to digitally read results of biomolecular computations. They claim that “the developed systems are the first examples of enzyme-based biocomputing systems interfaced with ordinary silicon-based electronics.”

With biological gates at hand it should be possible to develop “wetware hardware” for executing computations but “the state of the art in biomolecular circuits remains far behind its electronic equivalent” [29].

In [98] Ran, Kaplan and Shapiro take the design of logic gates a step further and use the gates for specific “systems”. Paper [98] addresses the issue that in general, all encoding and decoding of input and output is done manually in the lab for each problem, by developing a “compiler” that translates first-order logic statements and queries into the specific DNA strands that encode the input and the gates needed for calculating the result. Figure 3 shows a translation of a logic query into DNA structures.

In another direction, Winfree et al in [84] show how to use DNA origami [106] to direct a “molecular spider” to autonomously carry out sequences like “start,” “follow“, ”turn“ and ”stop“. This could be imagined to be used as a way to implement ”Turing-universal algorithmic behavior” [84].

4.6 Conclusion

Research in the area of biocomputing, biomolecular algebras/calculi has been massive. Although still very much work in progress, the possibility of biomolecular computers holds great promise. That being said, even though some concepts and words known from computer software engineering, and computer sciences, are also seen in biocomputer design and research, a gap remains to be filled. The ultimate goal for a programming language on a biomolecular platform would be the ability to abstract away unnecessary details about the biological “hardware” and to be able to focus on writing algorithms and solving problems. The programming niche of the field of biomolecular computer research is still, as indicated by this literature survey, very much relevant and mostly unexplored.

Our work is a step towards bridging the aforementioned gap.

5 The Blob Model of Programmed Universal Computation

We take a very simplified view of a (macro-)molecule and its interactions, with abstraction level similar to the Kappa model [40,26,44]. To avoid misleading de-

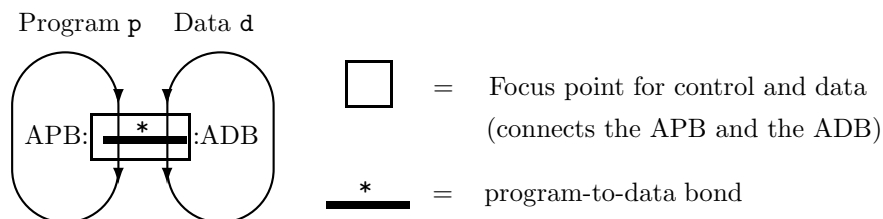
tail questions about real molecules we use the generic term “blob” for an abstract molecule. A collection of blobs in the biological “soup” may be interconnected by two-way *bonds* linking the individual blobs’ *bond sites*.

A *program p* is (by definition) a connected assembly of blobs. A data value *d* is (also) by definition a connected assembly of blobs. At any moment during execution, i.e., during computation of $\llbracket p \rrbracket(d)$ we have:

- One blob in *p* is active, known as the *active program blob* or APB.
- One blob in *d* is active, known as the *active data blob* or ADB.
- A bond $*$, between the APB and the ADB, is linked at a specially designate bond site, bond site 0, of each.

The main idea is to keep both program control point and the current data inspection site always close to a *focus point* where all actions occur. This can be done by continually shifting the program or the data, to keep the *active program blob* (APB) and *active data blob* (ADB) always in reach of the focus. The picture illustrates this idea for direct program execution.

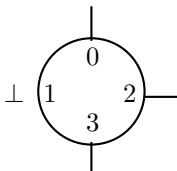
Running program p, i.e., computing $\llbracket p \rrbracket(d)$



The data view of blobs: A blob in our model have *four bond sites*, identified by numbers 0, 1, 2, 3. At any instant during execution, each can hold a bond – that is, a link to a (different) blob; or a bond can hold \perp , indicating unbound.

In addition each blob has 8 *cargo bits* of local storage containing Boolean values, and also identified by numerical positions: 0, 1, 2, ..., 7. (A biological analog to bits 1 or 0 is “phosphorylated” or “unphosphorylated”.)

A blob with 3 bond sites bound and one unbound:



Since bonds are in essence two-way pointers, they have a “fan-in” restriction: a given bond site can contain at most one bond (if not \perp).

The program view of blobs: Blob programs are sequential. There is no structural distinction between blobs used as data and blobs used as program. A single, fixed set of instructions is available for moving and rearranging the cursors, and for testing or setting a cargo bit at the data cursor. Novelties from a computer science viewpoint: there are no explicit program or data addresses,

just adjacent blobs. At any moment there is only a *single program cursor* and a *single data cursor*, connected by a bond written * above.

Instructions, in general. The blob instructions correspond roughly to “four-address code” for a von Neumann-style computer. An essential difference, though, is that a bond is a *two-way link between two blobs*, and is not an address at all. It is not a pointer; there *exists no address space* as in a conventional computer. A blob’s 4 bond sites contain links to other instructions, or to data via the APB-ADB bond.

For program execution, one of the 8 cargo bits is an “activation bit”; if 1, it marks the instruction currently being executed. The remaining 7 cargo bits are interpreted as a 7-bit instruction so there are $2^7 = 128$ possible instructions in all. An instruction has an *operation code* (around 15 possibilities), and 0, 1 or 2 *parameters* that identify single bits, or bond sites, or cargo bits in a blob. See the table below for some current details. For example, SCG v c has 16 different versions since v can be one of 2 values, and c can be one of 8 values.

Why exactly 4 bonds? The reason is that each instruction must have a bond to its predecessor; further, a test or “jump” instruction will have two successor bonds (true and false); and finally, there must be one bond to link the APB and the ADB, i.e., the bond * between the currently executing instruction and the currently visible data blob. The FIN instruction is a device to allow a locally limited fan-in.

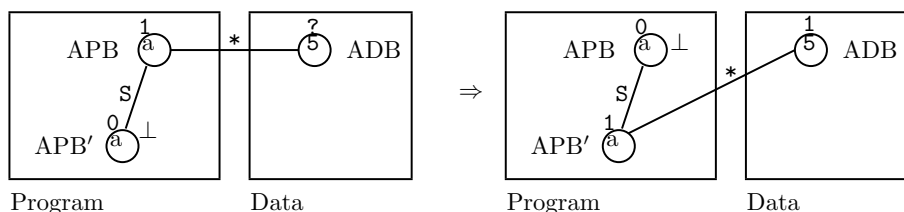
A specific instruction set (a bit arbitrary). In [67,68] we propose a specific instruction set. Here we include a subset of the instructions for illustration.

On the insert instruction INS. This creates a new blob, linked with the current ADB. Analogy: thinking of a blob as a cell or molecule (whichever paradigm seems natural), we are implicitly assuming that blobs are swimming in a “biological soup”, so INS just reconfigures a nearby element. From one viewpoint, this action resembles detaching a new cell from the freelist (the list of available cells used in Lisp/Scheme implementations).

Instruction	Description	Informal semantics
SCG v c	Set CarGo bit	ADB.c := v; APB := APB.2
JCG c	Jump CarGo bit	if ADB.c = 0 then APB := APB.3 else APB := APB.2
JB b	Jump Bond	if ADB.b = ⊥ then APB := APB.3 else APB := APB.2
CHD b	CHange Data	ADB := ADB.b; APB := APB.2
INS b1 b2	INSert new bond	new.b2 <i>bound to</i> ADB.b1; new.b1 <i>bound to</i> ADB.b1.bs; APB := APB.2 “new” is a fresh blob, “bs” is the bond site that ADB.b1 was bound to before INS b1 b2.
FIN	Fan IN	... APB := APB.2

On the need for a fan-in instruction. The point with FIN (short for “fan-in”) is that in blob code, unlike say Scheme, there cannot exist an unbounded number of pointers to a given blob (since every pointer corresponds to a bond site, and every blob has only 4 bond sites). So to achieve the effect of, say, 5 pointers to a blob instruction in a program, one can use a fan-in tree where each blob in the tree has at most 4 bond sites.

An example in detail: the instruction SCG 1 5, as picture and as a rewrite rule. SCG stands for “set cargo bit”. The effect of instruction SCG 1 5 is to change the 5-th cargo bit of the ADB (active data blob) to 1. First, an informal picture to show its effect:



Note: the APB-ADB bond * has moved: Before execution, it connected APB with ADB. After execution, it connects APB' with ADB, where APB' is the next instruction: the successor (via bond S) of the previous APB. Also note that the activation bit has changed: before, it was 1 at APB (indicating that the APB was about to be executed) and 0 at ADB'. Afterwards, those two bit values have been interchanged.

5.1 The Blob World from a Computer Science Perspective

First, an operational image: Any well-formed blob program, while running, is a collection of program blobs that is adjacent to a collection of data blobs, such that there is *one* critical bond (*) that links the APD and the ADB (the active program blob and the active data blob). As the computation proceeds, the program or data may move about, e.g., rotate as needed to keep their contact points adjacent (the APB and the ADB). For now, we shall not worry about the thermodynamic efficiency of moving arbitrarily large program and data in this way; for most realistic programs, we assume them to be sufficiently small (on the order of thousands of blobs) that energy considerations and blob coherence are not an issue.

5.2 The Blob Language

It is certainly small: around 15 operation codes (for a total of 128 instructions if parameters are included). Further, the set is irredundant in that no instruction's effect can be achieved by a combination of other instructions. There are easy computational tasks that simply cannot be performed by any program without, say, SCG or FIN.

There is a close analogy between blob programs and a *rudimentary machine language*. However a bond is not an address, but closer to a two-way pointer. On the other hand, there is *no address space*, and *no address decoding hardware* to move data to and from memory cells. An instruction has an unusual format, with 8 single bits and 4 two-way bonds. There is no fixed word size for data, there are no computed addresses, and there are no registers or indirection.

The blob programs have some similarity to *LISP* or *SCHEME*, but: there are no variables; there is no recursion; and bonds have a “fan-in” restriction.

5.3 What Can Be Done in the Blob World?

In principle the ideas presented and further directions are clearly expressible and testable in Maude or another tool for implementing term rewriting systems, or the kappa-calculus [26,31,40,44]. Recent work involves programming a blob simulator, and execution visualiser. Prototype implementations of both have been made, described in [67,68].

The usual programming tasks (appending two lists, copying, etc.) can be solved straightforwardly, albeit not very elegantly because of the low level of blob code.

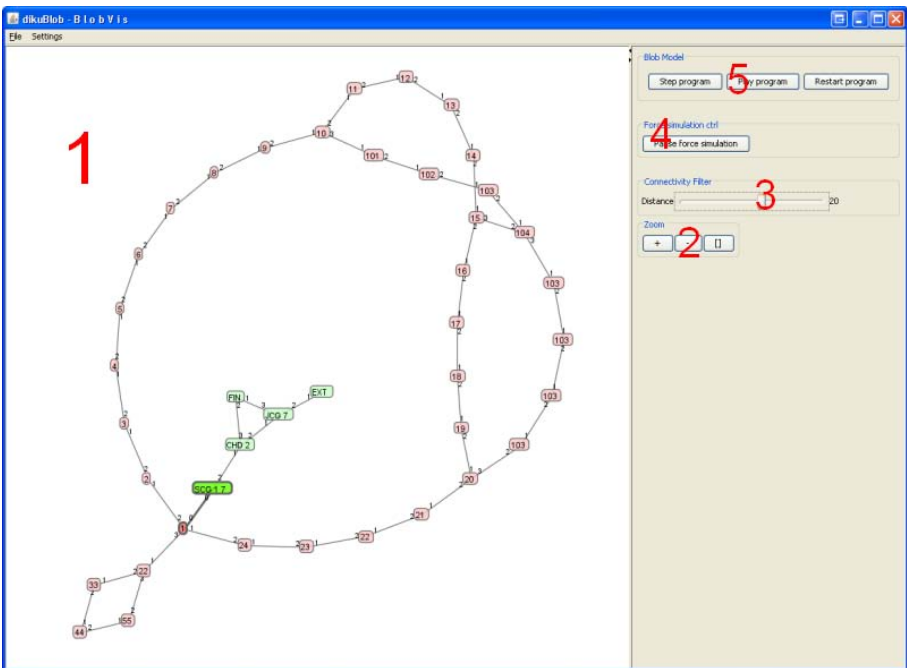


Fig. 8. The main interface of the blob visualiser. 1) visualisation area, 2) Zoom buttons, 3) Connectivity filter, 4) force-directed layout start/pause button, 5) blob program controls. Program blobs are green, data blobs red; the APB and ADB are emphasized by brighter colours and thicker bond lines.

It seems possible to make an analogy between universality and self-reproduction that is tighter than seen in the von Neumann and other cellular automaton approaches. It should now be clear that familiar Computer Science concepts such as interpreters and compilers also make sense also at the biological level, and hold the promise of becoming useful operational and utilitarian tools.

5.4 Blob Instruction Interpreter and Visualization Tool

We have developed an interpreter and visualization tool for the blob instruction set. Figure 8 shows an example of such an illustration. At <http://blobvis.appspot.com> the program and source code can be downloaded as well as several videos and images demonstrating programs and the usage of the visualization tool.

6 Conclusions

Computation via biological devices has been the subject of diverse and close scrutiny for many years. We have given a review of the literature on programming-related biocomputing and briefly identified some strengths and shortcomings from a programming perspective. Given the vast amount of work done in this field to date, the notion of programming the system or devices presented still seems to be ill-defined. Many models are claimed to be computationally universal in some sense. This universality prompted us to ask the question *Where are the programs?* Our recent work [67,68] tries to answer this question, where we presented the outline of a computation model that seems biologically more plausible than existing silicon-inspired models. We hope that it sets a standard for a biological device that can be both universal *and programmable*.

References

1. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. *Science* 266(11), 1021–1024 (1994)
2. Adleman, L.M.: On constructing a molecular computer. DIMACS: series in Discrete Mathematics and Theoretical Computer Science, pp. 1–21. American Mathematical Society (1996)
3. Amos, M., Paun, G., Rozenberg, G., Salomaa, A.: Topics in the theory of DNA computing. *Theor. Comput. Sci.* 287(1), 3–38 (2002)
4. Backofen, R., Clote, P.: Evolution as a computational engine. In: Proceedings of the Annual Conference of the European Association for Computer Science Logic, pp. 35–55. Springer, Heidelberg (1996)
5. Beaver, D.: Computing with dna. *Journal of Computational Biology* 2(1), 1–7 (1995)
6. Beaver, D.: Computing with DNA. *Journal of Computational Biology* 2(1), 1–7 (1995)
7. Benenson, Y., Adar, R., Paz-Elizur, T., Livneh, Z., Shapiro, E.: Dna molecule provides a computing machine with both data and fuel. *Proc Natl Acad Sci U S A* 100(5), 2191–2196 (2003), <http://dx.doi.org/10.1073/pnas.0535624100>

8. Benenson, Y., Adar, R., Paz-Elizur, T., Livneh, Z., Shapiro, E.: DNA molecule provides a computing machine with both data and fuel. In: Noltemeier, H. (ed.) WG 1980. LNCS, vol. 100, pp. 2191–2196. Springer, Heidelberg (1981)
9. Benenson, Y., Paz-Elizur, T., Adar, R., Keinan, E., Livneh, Z., Shapiro, E.: Programmable and autonomous computing machine made of biomolecules. *Nature* 414(1), 430–434 (2001), <http://www.nature.com/nature/links/011122/011122-2.html>
10. Benenson, Y.: Biocomputers: from test tubes to live cells. *Molecular BioSystems* 5(7), 675–685 (2009), <http://dx.doi.org/10.1039/b902484k>
11. Benenson, Y.: RNA-based computation in live cells. *Current Opinion in Biotechnology* 20(4), 471–478 (2009), <http://www.sciencedirect.com/science/article/B6VRV-4X4BR27-2/2/0133dc1fc3a23441b6aa9bab4115fc11>, protein technologies / Systems and synthetic biology
12. Bennett, C.H.: Logical reversibility of computation. *IBM Journal of Research and Development* 17(6), 525–532 (1973)
13. Bennett, C.H.: The thermodynamics of computation – a review. *International Journal of Theoretical Physics* 21(12), 905–940 (1982), <http://dx.doi.org/10.1007/BF02084158>
14. Berger, R.: The undecidability of the domino problem. *Memoirs American Mathematical Society* 66 (1966)
15. Berlekamp, E.R., Conway, J.H., Guy, R.K.: *Winning Ways for your Mathematical Plays*, vol. 2, ch. 25. Academic Press (1982) ISBN 0-12-091152-3
16. Bohringer, K.-F., Paulisch, N.F.: Using constraints to achieve stability in automatic graph layout algorithms. In: *Proceedings of ACM CHI 1990 Conference on Human Factors in Computing Systems*, pp. 43–51. Constraint Based UI Tools (1990)
17. Boneh, D., Dunworth, C., Lipton, R.J.: Breaking DES using a molecular computer. In: Lipton, E.B.B.R.J. (ed.) *DNA Based Computers*. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 27, pp. 37–66. American Mathematical Society (1995)
18. Braich, R.S., Chelyapov, N., Johnson, C., Rothmund, P.W.K., Adleman, L.: Solution of a 20-variable 3-sat problem on a dna computer. *Science* 296, 499–502 (2002)
19. Cardelli, P.: An universality result for a (mem)brane calculus based on mate/drip operations. *IJFCS: International Journal of Foundations of Computer Science* 17 (2006)
20. Cardelli, L.: Brane calculi. In: Danos and Schächter [14], pp. 257–278, <http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3082&page=257>
21. Cardelli, L.: Abstract machines of systems biology. In: Priami, C., Merelli, E., Gonzalez, P., Omicini, A. (eds.) *Transactions on Computational Systems Biology III*. LNCS (LNBI), vol. 3737, pp. 145–168. Springer, Heidelberg (2005)
22. Cardelli, L.: Strand algebras for DNA computing. In: Deaton and Suyama [42], pp. 12–24
23. Cardelli, L.: *Molecular programming tutorial*, microsoft research, cambridge (February 2010), <http://lucacardelli.name/Talks/2010-02-11%20Molecular%20Programming%20Tutorial.pdf>
24. Cardelli, L.: Biocomputers is not a good idea of solving np complete problems. said during presentation of strand algebra, CS2Bio, Amsterdam (2010)

25. Cardelli, L., Zavattaro, G.: On the computational power of biochemistry. In: Horimoto, K., Regensburger, G., Rosenkranz, M., Yoshida, H. (eds.) AB 2008. LNCS, vol. 5147, pp. 65–80. Springer, Heidelberg (2008)
26. Cardelli, L., Zavattaro, G.: Turing universality of the biochemical ground form. *Mathematical Structures in Computer Science* 19 (2009)
27. Chang, W.L., Ho, M.H., Guo, M.: Molecular solutions for the subset-sum problem on DNA-based supercomputing. *Biosystems* 73, 117–130(14) (2004), <http://www.ingentaconnect.com/content/els/03032647/2004/00000073/00000002/art00225>
28. Chapman, P.: Life universal computer (November 2002), <http://www.igblan.free-online.co.uk/igblan/ca/>
29. Chiniforooshan, E., Doty, D., Kari, L., Seki, S.: Scalable, time-responsive, digital, energy-efficient molecular circuits using DNA strand displacement. *CoRR abs/1003.3275* (2010)
30. Ciocchetta, F., Hillston, J.: Bio-PEPA: An extension of the process algebra PEPA for biochemical networks. *Electr. Notes Theor. Comput. Sci.* 194(3), 103–117 (2008), <http://dx.doi.org/10.1016/j.entcs.2007.12.008>
31. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
32. Condon, A., Rozenberg, G. (eds.): DNA 2000. LNCS, vol. 2054. Springer, Heidelberg (2001)
33. Conrad, M.: On design principles for a molecular computer. *Commun. ACM* 28(5), 464–480 (1985)
34. Conrad, M.: Quantum molecular computing: The self-assembly model. *International Journal of Quantum Chemistry. Quantum Biology Symposium: Proceedings of the International Symposium on Quantum Biology and Quantum Pharmacology*, vol. 19, pp. 125–143 (1992)
35. Danchin, A.: Bacteria as computers making computers. *FEMS Microbiology Reviews* 33(1), 3–26 (2008)
36. Danos, V., Feret, J., Fontana, W., Krivine, J.: Abstract interpretation of cellular signalling networks. In: Logozzo, et al [83], pp. 83–97
37. Danos, V., Feret, J., Fontana, W., Krivine, J.: Abstract interpretation of cellular signalling networks. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 83–97. Springer, Heidelberg (2008)
38. Danos, V., Krivine, J.: Formal molecular biology done in CCS-R. *Electr. Notes Theor. Comput. Sci.* 180(3), 31–49 (2007), <http://dx.doi.org/10.1016/j.entcs.2004.01.040>
39. Danos, V., Laneve, C.: Formal molecular biology. *Theor. Comput. Sci.* 325(1), 69–110 (2004)
40. Danos, V., Laneve, C.: Formal molecular biology. *Theor. Comp. Science* 325, 69–110 (2004)
41. Danos, V., Schachter, V. (eds.): CMSB 2004. LNCS (LNBI), vol. 3082. Springer, Heidelberg (2005)
42. Deaton, R., Suyama, A. (eds.): DNA 15. LNCS, vol. 5877. Springer, Heidelberg (2009)
43. Degano, P., Gorrieri, R. (eds.): CMSB 2009. LNCS, vol. 5688. Springer, Heidelberg (2009)

44. Delzanno, G., Giusto, C.D., Gabbriellini, M., Laneve, C., Zavattaro, G.: The *kappa*-lattice: Decidability boundaries for qualitative analysis in biological languages. In: Degano and Gorrieri [43], pp. 158–172
45. Doty, D., Patitz, M.J.: A Domain-Specific Language for Programming in the Tile Assembly Model, pp. 25–34. Springer, Heidelberg (2009)
46. Eades, P.: A heuristic for graph drawing. *Congressus Numerantium* 42, 149–160 (1984)
47. Eades, P., Lai, W., Misue, K., Sugiyama, K.: Preserving the mental map of a diagram. In: *COMPUGRAPHICS 1991*, vol. I, pp. 34–43 (1991)
48. Elbaz, J., Lioubashevski, O., Wang, F., Remacle, F., Levine, R.D., Willner, I.: DNA computing circuits using libraries of DNzyme subunits. *Nat. Nanotechnol.* 5(6), 417–422 (2010), <http://dx.doi.org/10.1038/nnano.2010.88>
49. Fisher, J., Henzinger, T.A.: Executable cell biology. *Nature Biotechnology* 25(11), 1239–1249 (2007), <http://dx.doi.org/10.1038/nbt1356>
50. Fleischer, R., Hirsch, C.: Graph drawing and its applications. In: Kaufmann, M., Wagner, D. (eds.) *Drawing Graphs*. LNCS, vol. 2025, pp. 1–22. Springer, Heidelberg (2001)
51. Fontana, W., Buss, L.: The barrier of objects: From dynamical systems to bounded organizations. Working Papers wp96027, International Institute for Applied Systems Analysis (March 1996), <http://ideas.repec.org/p/wop/iasawp/wp96027.html>
52. Frick, A., Ludwig, A., Mehldau, H.: A fast adaptive layout algorithm for undirected graphs. In: Tamassia, R., Tollis, I.G. (eds.) *GD 1994*. LNCS, vol. 894, pp. 388–403. Springer, Heidelberg (1995), <http://dblp.uni-trier.de/db/conf/gd/gd94.html#FrickLM94>
53. Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. *Software: Practice and Experience* 21(11), 1129–1164 (1991), citeseer.ist.psu.edu/fruchterman91graph.html
54. Fu, P.: Biomolecular computing: Is it ready to take off? *Biotechnology Journal* 2(1), 91–101 (2007), <http://dx.doi.org/10.1002/biot.200600134>
55. Gardner, M.: The fantastic combinations of John Conway’s new solitaire game “life”. *Scientific American* 223, 120–123 (1970)
56. Gardner, M.: *Mathematical recreations*. *Scientific American* (October 1970)
57. Garzon, M.H., Deaton, R.J.: Biomolecular computing and programming. *IEEE Trans. Evolutionary Computation* 3(3), 236–250 (1999)
58. Gilmore, S., Hillston, J.: The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In: Haring, G., Kotsis, G. (eds.) *TOOLS 1994*. LNCS, vol. 794, pp. 353–368. Springer, Heidelberg (1994)
59. Giral, U.D.E., Cetintas, A., Civril, A., Demir, E.: A compound graph layout algorithm for biological pathways. In: Pach [94], pp. 442–447, <http://dblp.uni-trier.de/db/conf/gd/gd2004.html#DogrusozGCCD04>
60. Goel, A., Ibrahim, M.: Renewable, time-responsive DNA logic gates for scalable digital circuits. In: Deaton and Suyama [42], pp. 67–77
61. Goel, A., Simmel, F.C., Sosik, P. (eds.): *DNA Computing*. LNCS, vol. 5347. Springer, Heidelberg (2009)
62. Guarnieri, F., Fliss, M., Bancroft, C.: Making DNA add. *Science* 273(5272), 220–223 (1996)

63. Guerriero, M.L., Prandi, D., Priami, C., Quaglia, P.: Process calculi abstractions for biology. Tech. rep., CoSBI (Center for Computational and Systems Biology), University of Trento (January 01, 2006), <http://eprints.biblio.unitn.it/archive/00001704/>, <http://eprints.biblio.unitn.it/archive/00001704/01/TR-13-2006.pdf>
64. Guerriero, M.L., Prandi, D., Priami, C., Quaglia, P.: Process calculi abstractions for biology. Tech. rep., University of Trento, Italy (January 01, 2006), <http://eprints.biblio.unitn.it/archive/00001704/>, <http://eprints.biblio.unitn.it/archive/00001704/01/TR-13-2006.pdf>
65. Hagiya, M.: From molecular computing to molecular programming. In: Condon and Rozenberg [32], pp. 89–102, <http://link.springer.de/link/service/series/0558/bibs/2054/20540089.htm>
66. Hagiya, M.: Designing chemical and biological systems. *New Generation Comput.* 26(3), 295 (2008)
67. Hartmann, L., Jones, N., Simonsen, J.: Programming in biomolecular computation. In: CS2BIO 2009: Proceedings of the 1st International Workshop on Interactions between Computer Science and Biology. *Electronic Notes on Theoretical Computer Science series*. Elsevier (2010), <http://dx.doi.org/10.1016/j.entcs.2010.12.008>
68. Hartmann, L., Jones, N., Simonsen, J., Vrist, S.: Programming in biomolecular computation: Programs, self-interpretation and visualisation. To appear in *Scientific Annals of Computer Science*, <http://dk.diku.blo.blovis.s3.amazonaws.com/blobiasi.pdf>
69. Heer, J.: Prefuse: a software framework for interactive information visualization. Master's thesis, University of California, Berkeley (2004), <http://jheer.org/publications/2004-Heer-prefuse-MastersApp.pdf>
70. Hoare, C.A.R.: Communicating sequential processes. *Communications of the ACM* 21(8), 666–677 (1978)
71. Hug, H., Schuler, R.: Strategies for the development of a peptide computer. *Bioinformatics* 17(4), 364–368 (2001), <http://bioinformatics.oxfordjournals.org/content/17/4/364.abstract>
72. Jones, J.E.: On the determination of molecular fields. ii. from the equation of state of a gas. *Proceedings of the Royal Society of London. Series A* 106(738), 463–477 (1924), <http://dx.doi.org/10.1098/rspa.1924.0082>
73. Jones, N., Gomard, C., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall (1993)
74. Jones, N.D.: *Computability and complexity: from a programming perspective*. MIT Press, Cambridge (1997)
75. Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. *Information Processing Letters* 31(1), 7–15 (1989)
76. Kari, J.: Theory of cellular automata: A survey. *Theoretical Computer Science* 334(1-3), 3–33 (2005), <http://www.sciencedirect.com/science/article/B6V1G-4FDS8HM-2/2/7bdf589f505353432c8447e06f491ceb>
77. Kari, L.: *Biological computation: How does nature compute?* Tech. rep., University of Western Ontario (2009)
78. Kari, L., Rozenberg, G.: The many facets of natural computing. *Commun. ACM* 51(10), 72–83 (2008)
79. Khodor, J.: *DNA-based string rewrite computational systems*. Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science (2002), <http://hdl.handle.net/1721.1/8339>

80. Khodor, J., Gifford, D.K.: Programmed mutagenesis is universal. *Theory Comput. Syst.* 35(5), 483–500 (2002), <http://dblp.uni-trier.de/db/journals/mst/mst35.html#KhodorG02>
81. Krämer, M., Pita, M., Zhou, J., Ornatska, M., Poghossian, A., Schöning, M.J., Katz, E.: Coupling of biocomputing systems with electronic chips: Electronic interface for transduction of biochemical information. *The Journal of Physical Chemistry C* 113(6), 2573–2579 (2009), <http://pubs.acs.org/doi/abs/10.1021/jp808320s>
82. Lipton, R.J.: Using DNA to solve NP-complete problems. *Science* 268, 542–545 (1995)
83. Logozzo, F., Peled, D., Zuck, L.D. (eds.): *VMCAI 2008. LNCS, vol. 4905*. Springer, Heidelberg (2008)
84. Lund, K., Manzo, A.J., Dabby, N., Michelotti, N., Johnson-Buck, A., Nangreave, J., Taylor, S., Pei, R., Stojanovic, M.N., Walter, N.G., Winfree, E., Yan, H.: Molecular robots guided by prescriptive landscapes. *Nature* 465(7295), 206–210 (2010), <http://dx.doi.org/10.1038/nature09012>
85. Mao, C., Labean, T.H., Reif, J.H., Seeman, N.C.: Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. *Nature* 407, 493–496 (2000)
86. Milner, R.: *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River (1989)
87. Milner, R.: *Functions as processes*. Research Report 1154, INRIA (1990)
88. Minsky, M.: *Computation: finite and infinite machines*. Prentice-Hall, Englewood Cliffs (1967)
89. Misue, K., Eades, P., Lai, W., Sugiyama, K.: Layout adjustment and the mental map. *Journal of Visual Languages and Computing* 6(2), 183–210 (1995), <http://www.sciencedirect.com/science/article/B6WMM-45PVMS3-13/2/0f1f0f6cf4f49a7892fb6064751b128c>
90. Stojanovic, M.N., Stefanovic, D.: A deoxyribozyme-based molecular automaton. *Nature Biotechnol.* 21(9), 1069–1074 (2003)
91. Murata, S., Stojanovic, M.N.: DNA-based nanosystems. *New Generation Comput.* 26(3), 297–312 (2008)
92. Nehaniv, C.L.: Asynchronous automata networks can emulate any synchronous automata network. *International Journal of Algebra and Computation* 14(5-6), 719–739 (2004)
93. von Neumann, J., Burks, A.W.: *Theory of Self-Reproducing Automata*. Univ. Illinois Press (1966)
94. Pach, J. (ed.): *GD 2004. LNCS, vol. 3383*. Springer, Heidelberg (2005)
95. Parker, J.: Computing with DNA. *EMBO Rep.* 4(7), 7–10 (2003)
96. Phillips, A., Cardelli, L.: A programming language for composable DNA circuits. *Journal of the Royal Society Interface* 6(S4) (2009)
97. Qian, L., Winfree, E.: A simple DNA gate motif for synthesizing large-scale circuits. In: Goel, et al [61], pp. 70–89
98. Ran, T., Kaplan, S., Shapiro, E.: Molecular implementation of simple logic programs. *Nat. Nano.* 4(10), 642–648 (2009), <http://dx.doi.org/10.1038/nnano.2009.203>
99. Regev, A., Shapiro, E.Y.: Cells as computation. In: Priami, C. (ed.) *CMSB 2003. LNCS, vol. 2602*, pp. 1–3. Springer, Heidelberg (2003), <http://link.springer.de/link/service/series/0558/bibs/2602/26020001.htm>

100. Regev, A., Panina, E.M., Silverman, W., Cardelli, L., Shapiro, E.: Bioambients: An abstraction for biological compartments. *TCS: Theoretical Computer Science* 325 (2004)
101. Regev, A., Silverman, W., Shapiro, E.Y.: Representation and simulation of biochemical processes using the pi-calculus process algebra. In: *Pacific Symposium on Biocomputing*, pp. 459–470 (2001), <http://helix-web.stanford.edu/psb01/regev.pdf>
102. Reif, J.H., LaBean, T.H.: Autonomous programmable biomolecular devices using self-assembled DNA nanostructures. *Commun. ACM* 50(9), 46–53 (2007), <http://dblp.uni-trier.de/db/journals/cacm/cacm50.html#ReifL07>
103. Robinson, R.M.: Undecidability and nonperiodicity for tilings of the plane. *Inv. Math.* 12, 117–209 (1971)
104. Rothmund, P.W.K.: A DNA and restriction enzyme implementation of Turing machines. In: Lipton, E.B.B.R.J. (ed.) *DNA Based Computers. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27, pp. 75–120. American Mathematical Society (1995)
105. Rothmund, P.W.K.: Using lateral capillary forces to compute by self-assembly. *Proceedings of the National Academy of Sciences of the United States of America* 97(3), 984–989 (2000), <http://www.pnas.org/content/97/3/984.abstract>
106. Rothmund, P.: Folding DNA to create nanoscale shapes and patterns. *Nature* 440, 297–302 (2006)
107. Roweis, S., Winfree, E., Burgoyne, R., Chelyapov, N.V., Goodman, M.F., Rothmund, P.W.K., Adleman, L.M.: A sticker based model for DNA computation. In: Landweber, L., Baum, E. (eds.) *DNA Based Computers II. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, vol. 44, American Mathematical Society (1996), <ftp://hope.caltech.edu/pub/roweis/DIMACS/stickers.ps>
108. Roweis, S.T., Winfree, E., Burgoyne, R., Chelyapov, N.V., Goodman, M.F., Rothmund, P.W.K., Adleman, L.M.: A sticker-based model for DNA computation. *Journal of Computational Biology* 5(4), 615–630 (1998)
109. Sander, G.: Graph layout for applications in compiler construction. *Theor. Comput. Sci.* 217(2), 175–214 (1999), <http://dblp.uni-trier.de/db/journals/tcs/tcs217.html#Sander99>
110. Sangiorgi, D., Walker, D.: *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press (2001)
111. Seelig, G., Soloveichik, D.: Time-Complexity of Multilayered DNA Strand Displacement Circuits, pp. 144–153. Springer, Heidelberg (2009)
112. Seelig, G., Soloveichik, D., Zhang, D.Y., Winfree, E.: Enzyme-Free Nucleic Acid Logic Circuits. *Science* 314(5805), 1585–1588 (2006), <http://www.sciencemag.org/cgi/content/abstract/314/5805/1585>
113. Shapiro, B.: Bringing DNA computers to life. *SCIAM: Scientific American* 294 (2006)
114. Shapiro, E.: Mechanical Turing machine: Blueprint for a biomolecular computer. Tech. rep., Weizmann Institute of Science (1999)
115. Shapiro, E.: Mechanical Turing machine: Blueprint for a biomolecular computer. Tech. rep., Weizmann Institute of Science (1999)
116. Shapiro, E., Benenson, Y.: Bringing DNA computers to life. *Scientific American* 294, 44–51 (2006)
117. Shlyahovskiy, B., Li, Y., Lioubashevski, O., Elbaz, J., Willner, I.: Logic gates and antisense DNA devices operating on a translator nucleic acid scaffold. *ACS Nano* 3(7), 1831–1843 (2009), <http://dx.doi.org/10.1021/nn900085x>

118. Simpson, M.L., Sayler, G.S., Fleming, J.T., Applegate, B.: Whole-cell biocomputing. *Trends Biotechnol.* 19(8), 317–323 (2001), <http://www.biomedsearch.com/nih/Whole-cell-biocomputing/11451474.html>
119. Smith, W.D.: DNA computers in vitro and vivo. In: Lipton, E.B.B.R.J. (ed.) *DNA Based Computers. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27, pp. 121–186. American Mathematical Society (1995)
120. Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. In: Goel et al [61], pp. 57–69
121. Stefansen, C.: SMAWL: A SMALL workflow language based on CCS. In: Belo, O., Eder, J. (eds.) *CAiSE 2005. CAiSE Forum, Short Paper Proceedings. CEUR Workshop Proceedings*, vol. 161, CEUR-WS.org (2005), http://www.ceur-ws.org/Vol-161/FORUM_10.pdf
122. Storey, M.A.D., Fracchia, F., Müller, H.: Customizing a Fisheye View Algorithm to Preserve the Mental Map. *Journal of Visual Languages and Computing* 10(3), 245–267 (1999)
123. Talcott, C.: Pathway logic. In: Bernardo, M., Degano, P., Tennenholtz, M. (eds.) *SFM 2008. LNCS*, vol. 5016, pp. 21–53. Springer, Heidelberg (2008)
124. Turing, A.: On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 42(2), 230–265 (1936-1937)
125. Wang, H.: Proving theorems by pattern recognition ii. *Bell System Technical Journal* 40, 1–40 (1961)
126. Wang, S., Yang, A.: DNA solution of integer linear programming. *Applied Mathematics and Computation* 170(1), 626–632 (2005)
127. Winfree, E.: Toward molecular programming with DNA. *SIGOPS Oper. Syst. Rev.* 42(2), 1–1 (2008)
128. Winfree, E., Eng, T., Rozenberg, G.: String tile models for DNA computing by self-assembly. In: Condon, A., Rozenberg, G. (eds.) *DNA 2000. LNCS*, vol. 2054, pp. 63–88. Springer, Heidelberg (2001)
129. Winfree, E., Yang, X., Seeman, N.C.: Universal computation via self-assembly of DNA: Some theory and experiments. In: *DNA Based Computers II. DIMACS*, vol. 44, pp. 191–213. American Mathematical Society (1996)
130. Winfree, E., Yang, X., Seeman, N.C.: Universal computation via self-assembly of DNA: Some theory and experiments. In: *DNA Based Computers II. DIMACS*, vol. 44, pp. 191–213. American Mathematical Society (1996)
131. Wolfram, S.: *A New Kind of Science*. Wolfram Media (January 2002), <http://www.amazon.com/exec/obidos/ASIN/1579550088/ref=nosim/rds-20>
132. Yan, H., Park, S.H., Finkelstein, G., Reif, J.H., Labean, T.H.: DNA-templated self-assembly of protein arrays and highly conductive nanowires. *Science* 301(5641), 1882–1884 (2003), <http://dx.doi.org/10.1126/science.1089389>
133. Yin, P., Choi, H.M.T., Calvert, C.R., Pierce, N.A.: Programming biomolecular self-assembly pathways. *Nature* 451(7176), 318–322 (2008), <http://dx.doi.org/10.1038/nature06451>
134. Yin, P., Turberfield, A.J., Sahu, S., Reif, J.H.: Design of an autonomous DNA nanomechanical device capable of universal computation and universal translational motion. In: Ferretti, C., Mauri, G., Zandron, C. (eds.) *DNA 2004. LNCS*, vol. 3384, pp. 426–444. Springer, Heidelberg (2005)

135. Yokomori, T., Kobayashi, S., Ferretti, C.: On the power of circular splicing systems and DNA computability. In: IEEE International Conference on Evolutionary Computation (1997), <http://ylab-gw.cs.uec.ac.jp/..Papers/yokomori/cssfinal.ps.gz>
136. Zhang, D.Y.: Dynamic DNA strand displacement circuits. Ph.D. thesis, California Institute of Technology (2010), <http://resolver.caltech.edu/CaltechTHESIS:05262010-173410602>
137. Zhu, Y., Ding, Y., Li, W., Kemp, G.: A proposed modularized dna computer, based on biochips. In: GEC 2009: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, pp. 773–780. ACM, New York (2009)

Applications of Pathway Logic Modeling to Target Identification

Anupama Panikkar¹, Merrill Knapp³, Huaiyu Mi², Dave Anderson¹,
Krishna Kodukula¹, Amit K. Galande¹, and Carolyn Talcott²

¹ Center for Advanced Drug Research, SRI International, Harrisonburg, Virginia 22802
{anupama.panikkar, david.anderson, krishna.kodukula,
amit.galande}@sri.com

² Computer Science Lab, SRI International, Menlo Park, CA 94025
mi@ai.sri.com, clt@csl.sri.com

³ Biosciences Division, SRI International, Menlo Park, CA 94025
merrill.knapp@sri.com

Abstract. To explore the role of proteases in pathogenesis and as potential drug targets we need to elucidate their function and effect on biological networks. In this paper, we describe the application of Pathway Logic (PL) (<http://pl.csl.sri.com/>) to the *symbolic* modeling of the interaction networks of proteases of Gram-positive bacteria and the use of Pathway Logic Assistant tool (PLA) to browse and query these models. Pathway Logic is a systems biology approach to biological processes as integrated systems rather than isolated parts based on formal methods and rewriting logic. These models are developed using Maude, a formal language and tool set based on rewriting logic. We show how this approach can be used to represent and analyze systems at multiple levels of details. The Pathway Logic Assistant tool enables us to identify key proteases and regulatory molecules – ‘choke points’ by comparing different pathways or networks within and across species and to predict how these molecules, if inhibited or avoided would affect the pathway or network.

1 Introduction

The emergence of Gram-positive drug resistant bacteria such as methicillin-resistant *Staphylococcus aureus* (MRSA), *Streptococcus pneumoniae* and Enterococcus represent a serious public health problem. This might be resolved by using a new generation of antibiotics with a different mechanism of action. Proteases (also termed peptidases or proteinases) are enzymes that break down proteins by hydrolysis of peptide bonds in the proteins. Bacterial proteases are implicated in virtually every important biological process related to colonization and evasion of host immune defenses, acquisition of nutrients for growth and proliferation, or tissue damage during infection. Thus, bacterial proteases represent suitable drug targets and inhibition of these enzymes would retard the growth and proliferation of invading pathogens [1–3]. It is critical to understand the role of proteases by modeling them in the context of multiple components such as protein signaling networks and complex biochemical pathways that can influence their activity. In this study, we use the Pathway Logic

(PL) [4–6] framework to *symbolically* model the protease networks and pathway interconnectivity of multiple Gram-positive bacteria including pathogenic and non-pathogenic species in an effort to develop a comprehensive computational model.

Pathway Logic is a *symbolic* systems approach to the modeling and analysis of molecular and cellular processes based on rewriting logic [7]. *Symbolic* systems biology is the *qualitative* and *quantitative* study of biological processes as integrated systems rather than as isolated parts. An important objective of Pathway Logic is to reflect the ways that biologists think about problems using informal models, and to provide bench biologists with tools for computing with and analyzing these models. Symbolic/logical models allow one to represent partial information and to model and analyze systems at multiple levels of details, depending on information available and questions to be studied.

Pathway Logic models are curated from the literature, and written and analyzed using Maude [8], [9] (<http://maude.cs.uiuc.edu/>), a rewriting-logic-based formalism. The Rewriting logic formalism is based on states of a system represented as elements of an algebraic data type and the behavior of a system given by local transitions between states described by *rewrite rules*. A rewrite rule has the form $t \Rightarrow t' \text{ if } c$ where t and t' are patterns (terms possibly containing place holder variables) and c is a condition (a boolean term). Such a rule applies to a system in state s if t can be matched to a part of s by supplying the right values for the place holders, and if the condition c holds when supplied with those values. The process of application of rewrite rules generates computations (also thought of as deductions) and in case of biological processes these computations correspond to pathways. In Pathway Logic, algebraic data types are used to represent concepts from cell biology needed to model signaling processes, including intracellular proteins, biochemicals such as second messengers, extracellular stimuli, biochemical modification of proteins, protein association, and cellular compartmentalization of proteins. Rewrite rules describe the behavior of proteins and other components depending on modification state and biological context. Each rule represents a step in a biological process such as metabolism or intra/inter-cellular signaling. A specific model is assembled by specifying an initial state (called a dish): the cells, their components, and entities such as ligands in the supernatant. Pathway Logic models are executable – hence they can be used for simulation. In addition, the Maude system provides search and model-checking capabilities. Using the search capability all possible future states of a system can be computed to show its evolution from a given initial state (specified by the states of individual components) in response to a stimulus or perturbation. Using model-checking a system in a given initial state can be shown to never exhibit pathways with certain properties, or the model-checker can be used to produce a pathway with a given property (by trying to show that no such pathway exists).

A Pathway Logic knowledge base includes data types representing cellular components such as proteins, small molecules, complexes, compartments/locations protein state, and post-translational modifications. Modifications can be as being activated, inhibited, phosphorylated, degraded or anchored. It also enables one to collect, store and retrieve curated information represented as metadata so that it can be understood and shared by a community of experimental biologists.

The Pathway Logic Assistant (PLA) [10] provides an interactive visual representation of PL models. Using the Pathway Logic Assistant (PLA) one can display pathways of interest, compare two pathways, search for cross talk between subsystems by exploring subnets, map gene expression data onto signaling networks and compute the effects of system perturbations by single or double knockouts (omission of individual or pairs of proteins that prevents reaching a specified state).

The remainder of the paper is organized as follows. The basic ideas of Pathway Logic are presented in §2, and illustrated with fragments from a model of heme transport in *Staphylococcus aureus*. Use of the Pathway Logic Assistant tool to browse and query models is discussed in §3. Applications of Pathway Logic in target discovery are shown by few examples in §4. The paper concludes with a discussion of future directions in §5.

2 Pathway Logic Basics

Pathway Logic models are structured in four layers: (1) sorts and operations, (2) components, (3) rules, and (4) dishes and queries. The *sorts and operations* layer declares the main sorts and subsort relations, the logical analog to ontology or class hierarchy. The sorts of entities include Chemical, Protein, Gene, Complex, Location (cellular compartments), and Cell. These are all subsorts of the sort, Soup that represents ‘liquid’ mixtures, as multisets (unordered collections) of entities. The sort Modification is used to represent post-translational protein modifications and gene regulations including up-regulation and down-regulation in the bacteria. They can be abstract, to specify that a protein is activated, inhibited, bound, anchored, degraded, phosphorylated, dephosphorylated, or more specific, for example, phosphorylation at a particular site. Gene up-regulation specifies increased expression of genes and their encoded protein and gene down-regulation indicates decreased gene and corresponding protein expression. Modifications are applied using the operator [-]. For example the term [IsdA - anchored] represents the iron-responsive surface determinant A (IsdA) protein in an anchored state and [ClpP-gene - on] represents *ClpP* gene in its ‘on’ state (upregulated).

A cell state is represented by a term of the form

$$[\text{cellType} \mid \text{locs}]$$

where `cellType` specifies the type of cell and `locs` represents the contents of a cell organized by cellular location. Each location is represented by a term of the form { `locName` | `components` } where `locName` identifies the location. In gram-positive bacteria the locations defined are

CLm for cell membrane

CLc for cytosol

CLw for cell wall

Xout for outside of the bacterial cell and

`components` stands for the mixture of proteins, genes and other compounds in that location.

The *components* layer specifies particular entities (proteins, genes, chemicals) and introduces additional sorts for grouping proteins in families. The *rules* layer contains rewrite rules specifying individual reaction steps. In the case of signal transduction, rules represent processes such as activation, phosphorylation, complex formation, or translocation. The sorts and operations, components, and rules layers make up a Pathway Logic knowledge base. The *dishes and queries* layer specifies initial states, relative to which queries can be answered, and properties of states to be used in formulating queries. Initial states are in silico Petri dishes containing a cell, with its components, and ligands of interest in the supernatant.

We give a brief overview of the representation in Maude of bacterial intracellular processes, illustrated using a model of heme transport involving the membrane cysteine protease-transpeptidases Sortase A (SrtA) and Sortase B (SrtB) in the pathogenic bacterium *S. aureus* in the following §2.1.

2.1 Modeling Heme Transport Involving SrtA and SrtB Protease-Transpeptidases in Pathway Logic

Pathogenic bacteria require iron as a source of nutrient during the infection process. *S. aureus* utilizes heme (a non-protein chemical compound that contains an iron atom) as a source of iron for its growth during infection. It acquires heme from the host environment and transports it across the cell wall into the cytoplasm by the heme-binding Isd proteins. The passage of heme also requires two sortases namely SrtA and SrtB that anchor these heme-binding Isd proteins to the cell wall. SrtA anchors IsdA, IsdB and IsdH proteins and SrtB anchors IsdC protein to the cell wall. As shown in Fig. 2, the heme binds to IsdA, IsdB, IsdC and IsdH proteins, which is then transported to the membrane transport system composed of IsdDEF into the cytoplasm. In the cytoplasm the heme is degraded by the IsdG and IsdI heme monooxygenases, releasing the free iron for use by the bacterium as a nutrient source. The Pathway Logic model of heme transport in *S. aureus* was curated based on papers from Olaf Schneewind's lab [11–13] and many other references (cited as metadata associated with individual rules). In the following we show an initial state for study of heme transport and examples of rules and briefly sketch some of the ways one can compute with the model. The initial state (called Hemetransport) is a dish PD(. . .) with a single cell represented by the following

```
Dish: PD([Cell |
  {XOut | Heme}
  {CLm | IsdD IsdE IsdF SrtA SrtB}
  {CLc | IsdG IsdI}
  {CLw | IsdA IsdB IsdC IsdH}])
```

Here, the dish contains Heme in the outside environment (location tag XOut). The cell membrane (tag CLm) has proteins IsdD, IsdE, IsdF, SrtA and SrtB. The cell wall (tag CLw) contains IsdA, IsdB, IsdC, and IsdH. The cytosol (tag CLc) contains proteins IsdG and IsdI.

The following is an example of the rule representing heme uptake by IsdB.

```
r1[5]:
  {XOut | xout Heme }
  {CLw | clw [IsdB - anchored] }
=>
  {XOut | xout }
  {CLw | clw ([IsdB - anchored] : Heme) } .
```

As shown in Fig. 2, applying rules 1-4 to the initial dish results in a dish with IsdA, IsdB, IsdC and IsdH anchored, and applying rule 5 to this dish results in

```
Dish: PD([Cell |
  {XOut | empty}
  {CLm | IsdD IsdE IsdF SrtA SrtB}
  {CLc | IsdG IsdI}
  {CLw | (Heme : [IsdB - anchored]) [IsdA - an
chored] [IsdC - anchored][IsdH - an
chored]})])
```

Maude [8], [9] can be used to find some execution, or to search for a state; for example, a state with Fe3+ in the CLc. However, the textual representation of cell states and pathways quickly becomes difficult to use as the size of a model grows, and an intuitive graphical representation becomes increasingly important. In addition, it becomes important to take advantage of the simple structure of PL models when searching for paths and carrying out other analyses. In the next section we show how the Pathway Logic Assistant can be used to visualize a model as a network of reaction rules, to browse the network, and to specify and execute queries.

3 The Pathway Logic Assistant

The Pathway Logic Assistant (PLA) [10] provides an interactive graphical view of a PL knowledge base. A PL knowledge base uses the Petri net transition representation of the Maude rules. A model is then generated by specifying a dish (initial state). Petri nets have a natural graphical representation, and additionally, there are very efficient tools for analyzing the Petri net models generated by PLA. Our Petri net models are a special case of Place-Transition Nets given by a set of occurrences (places in Petri net terminology) and a set of transitions [14]. Occurrences can be thought of as atomic propositions asserting that a protein (in a given state) or other component occurs in a given compartment. For example Heme outside a bacterial cell is represented by the occurrence <Heme, Xout> and IsdB anchored in the cell wall is represented by <[IsdB - anchored], CLw>. A system state is represented as a set of occurrences (called a marking in Petri net terminology), giving the propositions that are true. A transition is a pair of sets of occurrences. A transition can fire if the state contains the first set of occurrences. In which case the first set of occurrences is replaced by the second set. For example the rule labeled [5] shown in §2 becomes the transition

```
pnTrans[5]:
  <Heme, XOut> <[IsdB - anchored], CLw>
=>
  <Heme : [IsdB - anchored], CLw > .
```

In PLA goal properties are Petri net properties expressed as occurrences that must be present (places to be marked) and avoids properties are occurrences that must not appear (places not to be marked) in a computation. Paths leading from an initial state to a state satisfying a set of goals can be represented compactly as a Petri net consisting of the transitions fired in the path, thus giving query results a natural graphical representation. Execution of the path net starting with the initial state, leads to a state satisfying the goals, and the net representation makes explicit the dependency relations between transitions: some can fire concurrently (order doesn't matter), and some require the output of other transitions to be enabled.

Fig. 1 shows a screen shot of the Petri net model of the protease network generated by PLA. Ovals are occurrences, with initial occurrences darker. Rectangles are transitions. Dashed arrows indicate an occurrence that is both input and output. The thumb nail sketch in the upper right shows the full network. The main frame shows a magnified version of the portion of the network in the red rectangle. The Finder in the lower right allows one to locate occurrences and rules by name, and center the view on the selected node. To make a query, goals and avoids can be specified either by clicking

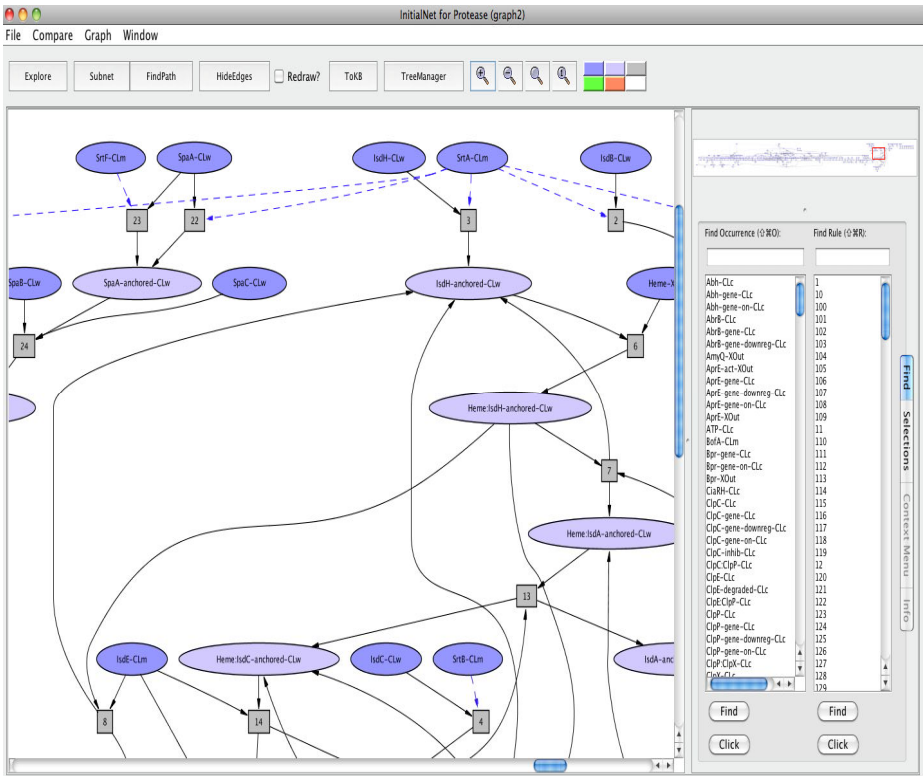


Fig. 1. Protease network model as Petri net viewed in PLA. Ovals are occurrences, with initial occurrences darker. Rectangles are transitions. Dashed arrows indicate an occurrence that is both input and output. The full protease net is shown in the upper right thumbnail. A magnified view of the portion in the red rectangle is shown in the main view.

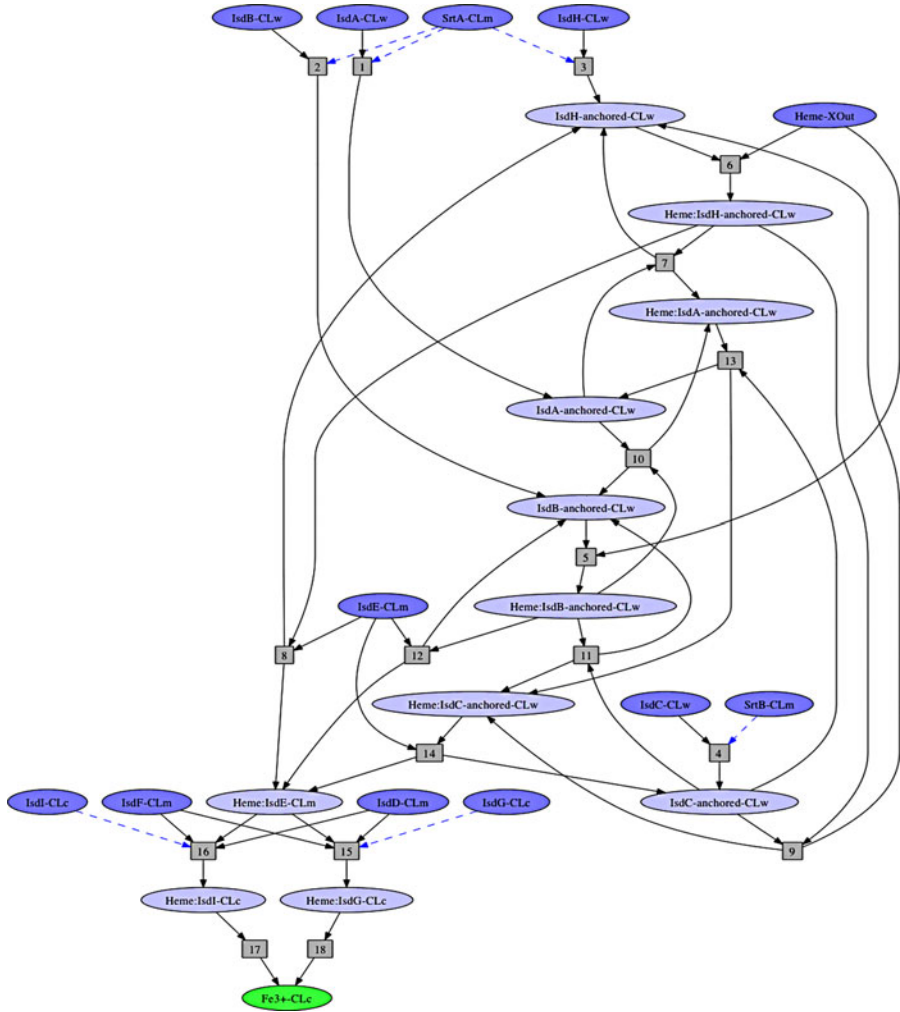


Fig. 2. Heme transport pathway in *S. aureus*. This pathway shows the transfer of heme by the Sortase anchored Isd proteins from outside of the bacterial cell to the cytosol and subsequent release of Fe³⁺ in the cytosol for use by the bacterium as a nutrient source.

on the occurrence and selecting goal or avoid in the selection window that appears, or by using the selection window directly. Once goals and avoids have been specified the user can ask to see the relevant subnet or to find a path. The relevant subnet contains all of the rules needed for any (minimal) pathway satisfying the query, while the path is just the first path found by the analysis tool. Fig. 2 shows the path found in response to the query in which the goal is Fe³⁺ in the cytosol (<Fe³⁺, CLc>) and there are no avoids.

4 Applications of Pathway Logic in Target Discovery

We describe here a few applications of Pathway Logic in target discovery. Identifying choke points or key molecules in the pathogens can accelerate the process of drug discovery. Choke points are critical points in a network and inactivation of choke points may lead to an organism's failure to produce or consume particular metabolites, which could cause serious problems for fitness or survival of the organism [15]. Potential drug targets are proposed based on the analysis of these choke points in the bacterial network.

4.1 Comparative Network Analysis and Target Inhibition

In Pathway Logic, one can identify choke points or key molecules by comparing different pathways or networks within and across species and target these molecules

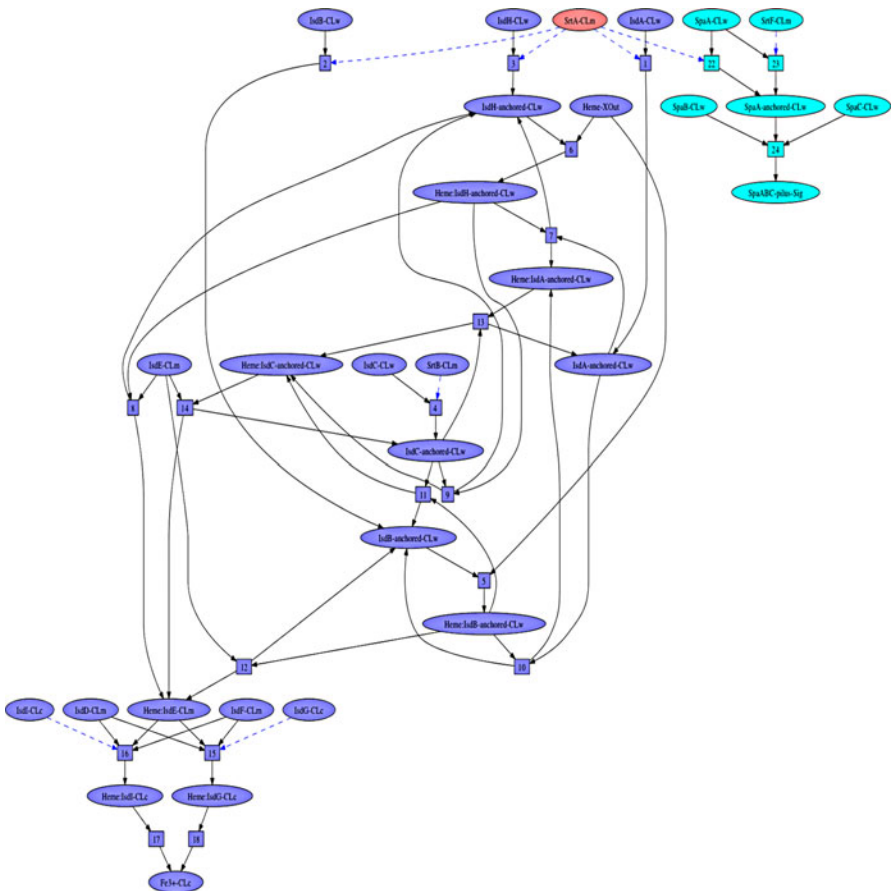


Fig. 3. Comparison of two pathways from *S. aureus* and *C. diphtheriae* - the heme transport (purple/darker color), and the pilus assembly (blue-green/lighter color). The common part (SrtA) is peach colored.

participating in pathogenesis. In addition to generating subnets and pathways, two subnets and/or pathways can be compared. For this, the two networks are merged into one and color-coded. Fig. 3 shows the result of comparing two pathways from two-bacterial species - the heme transport in *S. aureus* [12], [13] and the pilus assembly in *Corynebacterium diphtheriae* [16]: the heme transport (purple/darker color), and pilus assembly (blue-green/lighter color). The common part (SrtA) is peach colored.

The observed sharing of SrtA suggests asking: what happens if it is removed? Fig. 4 shows the subnet with SrtA avoided and viewing the result in the context of the heme-pilus subnet. The model shows inhibition of SrtA affects the heme transport pathway in *S. aureus*. This inhibition model corroborated with an earlier study showing that inactivation of SrtA lead to a decrease in the amount of iron associated with *S. aureus* cells. Biological studies have shown that the knockout mutation of *SrtA* gene in *S. aureus* greatly reduces the capacity of the pathogen to establish an acute infection in mice [17], [18]. Thus, in the growing antibiotic resistant scenario, SrtA may prove exciting new target of anti-infective therapy [19]. SrtA did not inhibit the pili formation due to the occurrence of another redundant pathway in which SrtF, a house keeping sortase, is involved in the Spa pili assembly.

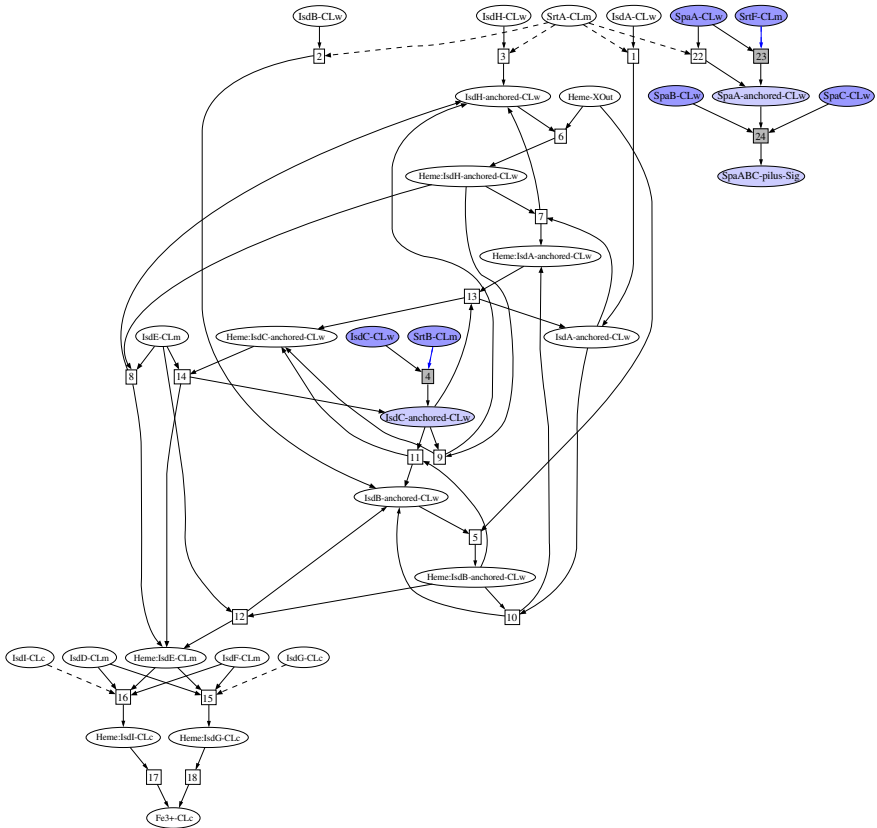


Fig. 4. Avoids query demonstrating single knockout of SrtA and its inhibitory effect on the heme transport in *S. aureus*.

4.2 Protease Interconnectivity Network

Pathway Logic models can be used to explore possible cross talks between proteases and identify key regulatory proteases. Insights into possible interactions among multiple processes involving proteases will give a better understanding to the underlying mechanisms and help develop specific inhibitors.

We show in Fig. 5, the interconnectivity between three proteases namely membrane serine protease HtrA, Group A streptococcus exotoxin B, SpeB (a cysteine protease) and membrane Type II SPase Lsp by setting these proteases as goals and generating the subnet (all the three protease are green/lighter color). We are able to show the direct and indirect influence they may have on each other. We show the influence of HtrA and Lsp in the processing and activation of SpeB [20–25]. The network also shows CiaRH and RopB upregulating HtrA and proSpeB (inactive precursor form) expression respectively.

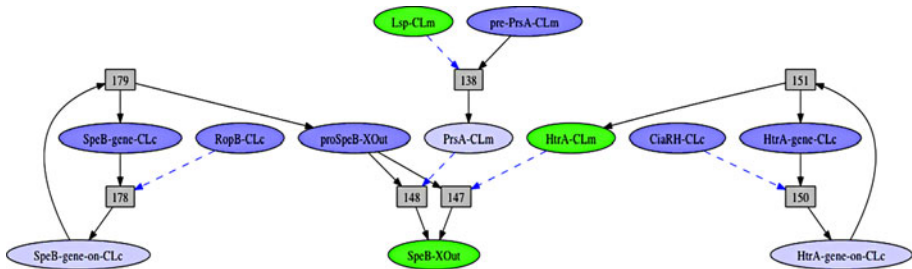


Fig. 5. The protease interconnectivity network. The three proteases (green/lighter color) within a larger protease network are shown to be interconnected in the process of SpeB activation.

5 Conclusions

We have described the Pathway Logic approach to modeling protease networks based on rewriting logic and the use of the Pathway Logic Assistant to browse and analyze these models. An important feature of PLA is the ability to generate pathways as query results. The current state of Pathway Logic is one step towards the grander vision of symbolic systems biology. Future challenges include developing such executable models of pathway interaction networks of proteases from both Gram-positive and Gram-negative bacteria to understand the underlying regulatory mechanisms for cell survival and pathogenicity leading to drug discovery. Another direction is to apply the basic approach to different types of systems, such as gene-regulation networks, or multi-cellular systems, and to integrate models of different types of systems to develop a systems level view.

Acknowledgements. This work is supported by funding to SRI International from the Commonwealth of Virginia. The authors would like to thank Festschrift organizers for inviting this paper; and members of the Pathway Logic team for their contributions to the development of modeling techniques and the analysis and visualization tools. Encouragement and guidance from Dr. Walter Moos, Vice President of Biosciences at SRI, are gratefully acknowledged.

References

1. Supuran, C.T., Scozzafava, A., Clare, B.W.: Bacterial Protease Inhibitors. *Med. Res. Rev.* 22(4), 329–372 (2002)
2. Travis, J., Potempa, J.: Bacterial Proteinases as Targets for the Development of Second-Generation Antibiotics. *Biochim. Biophys. Acta* 1477, 35–50 (2000)
3. Miyoshi, S.I., Shinoda, S.: Bacterial Metalloprotease as the Toxic Factor in Infection. *J. Toxicol. Toxin. Rev.* 16, 177–194 (1997)
4. Talcott, C., Eker, S., Knapp, M., Lincoln, P., Laderoute, K.: Pathway Logic Modeling of Protein Functional Domains in Signal Transduction. In: *Proceedings of the Pacific Symposium on Biocomputing*, vol. 9, pp. 568–580 (2004)
5. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., Sonmez, K.: Pathway Logic: Symbolic Analysis of Biological Signaling. In: *Proceedings of the Pacific Symposium on Biocomputing*, vol. 7, pp. 400–412 (2002)
6. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Talcott, C.: Pathway Logic: Executable Models of Biological Networks. In: *Fourth International Workshop on Rewriting Logic and Its Applications. Electronic Notes in Theoretical Computer Science*, vol. 71 (2002)
7. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: The Maude 2.0 system. In: Nieuwenhuis, R. (ed.) *RTA 2003. LNCS*, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
10. Talcott, C., Dill, D.L.: The Pathway Logic Assistant. *Proceedings of Computational Methods in Systems Biology*, 228–239 (2005)
11. Marraffini, L.A., DeDent, A.C., Schneewind, O.: Sortases and the Art of Anchoring Proteins to the Envelopes of Gram-Positive Bacteria. *Microbiol. Mol. Biol. Rev.* 70(1), 192–221 (2006)
12. Skaar, E.P., Gaspar, A.H., Schneewind, O.: IsdG and IsdI, Heme-Degrading Enzymes in the Cytoplasm of *Staphylococcus aureus*. *J. Biol. Chem.* 279(1), 436–443 (2004)
13. Skaar, E.P., Schneewind, O.: Iron-Regulated Surface Determinants (Isd) of *Staphylococcus aureus*: Stealing Iron from Heme. *Microbes. Infect.* 6(4), 390–397 (2004)
14. Stehr, M.O.: A Rewriting Semantics for Algebraic Nets. In: Girault, C., Valk, R. (eds.) *Petri Nets for System Engineering - A Guide to Modelling, Verification, and Applications* (2000)
15. Yeh, I., Hanekamp, T., Tsoka, S., Karp, P.D., Altman, R.B.: Computational Analysis of *Plasmodium falciparum* metabolism: Organizing Genomic Information to Facilitate Drug Discovery. *Genome Res.* 14(5), 917–924 (2004)
16. Ton-That, H., Schneewind, O.: Assembly of Pili on the Surface of *Corynebacterium diphtheriae*. *Mol. Microbiol.* 50(4), 1429–1438 (2003)
17. Mazmanian, S.K., Liu, G., Jensen, E.R., Lenoy, E., Schneewind, O.: *Staphylococcus aureus* Sortase Mutants Defective in the Display of Surface Proteins and in the Pathogenesis of Animal Infections. *Proc. Natl. Acad. Sci. U S A* 97(10), 5510–5515 (2000)
18. Mazmanian, S.K., Ton-That, H., Su, K., Schneewind, O.: An Iron-Regulated Sortase Anchors a Class of Surface Protein During *Staphylococcus aureus* Pathogenesis. *Proc. Natl. Acad. Sci. U S A* 99(4), 2293–2298 (2002)

19. Maresso, A.W., Schneewind, O.: Sortase as a Target of Anti-infective Therapy. *Pharmacol. Rev.* 60(1), 128–141 (2008)
20. Cole, J.N., Aquilina, J.A., Hains, P.G., Henningham, A., Sriprakash, K.S., Caparon, M.G., Nizet, V., Kotb, M., Cordwell, S.J., Djordjevic, S.P., Walker, M.J.: Role of Group A Streptococcus HtrA in the Maturation of SpeB Protease. *Proteomics* 7(24), 4488–4498 (2007)
21. Ma, Y., Bryant, A.E., Salmi, D.B., Hayes-Schroer, S.M., McIndoo, E., Aldape, M.J., Stevens, D.L.: Identification and Characterization of Bicistronic speB and prsA Gene Expression in the Group A Streptococcus. *J. Bacteriol.* 188(21), 7626–7634 (2006)
22. Tjalsma, H., Kontinen, V.P., Pragai, Z., Wu, H., Meima, R., Venema, G., Bron, S., Sarvas, M., van Dijl, J.M.: The Role of Lipoprotein Processing by Signal Peptidase II in the Gram-positive Eubacterium *Bacillus subtilis*. Signal peptidase II is Required for the Efficient Secretion of Alpha-amylase, a Non-lipoprotein. *J. Biol. Chem.* 274(3), 1698–1707 (1999)
23. Sutcliffe, I.C., Harrington, D.J.: Pattern Searches for the Identification of Putative Lipoprotein Genes in Gram-positive Bacterial Genomes. *Microbiology* 148(7), 2065–2077 (2002)
24. Sebert, M.E., Palmer, L.M., Rosenberg, M., Weiser, J.N.: Microarray-Based Identification of htrA, a *Streptococcus pneumoniae* Gene that is Regulated by the CiaRH Two-Component System and Contributes to Nasopharyngeal Colonization. *Infect. Immun.* 70(8), 4059–4067 (2002)
25. Hollands, A., Aziz, R.K., Kansal, R., Kotb, M., Nizet, V., Walker, M.J.: A Naturally Occurring Mutation in ropB Suppresses SpeB Expression and Reduces MIT1 Group A Streptococcal Systemic Virulence. *PLoS One* 3(12), e4102 (2008)

Author Index

- Álvarez, José María 329
Anderson, Dave 434
Arbab, Farhad 169

Bianchini, Devis 223

Cheung, Steven 110
Clavel, Manuel 277

Doh, Kyung-Goo 90
Duarte, Carlos Henrique C. 57
Durán, Francisco 329
Dutt, Nikil 207

Eker, Steven 299

Feferman, Solomon 1

Galande, Amit K. 434
Gehani, Ashish 262

Hartmann, Lars 403
Hölzl, Matthias 241

Jaghoori, Mohammad Mahdi 20
Jalali, Leila 352
Jones, Neil D. 403

Kim, Hyunha 90
Kim, Minyoung 110, 207
Knapp, Merrill 434
Kodukula, Krishna 434
Kreiker, Jörg 74

Lincoln, Pat 110

Martí-Oliet, Narciso 277
Mehrotra, Sharad 352
Mi, Huaiyu 434

Nielson, Flemming 74
Nielson, Hanne Riis 74

Ölveczky, Peter Csaba 368
Owre, Sam 315

Pagliarecci, Francesco 223
Palomino, Miguel 277
Panikkar, Anupama 434
Pilegaard, Henrik 74
Pinsky, Sylvan 4
Poggio, Andy 110

Ren, Shangping 144
Rocha, Camilo 329
Rushby, John 110

Schmidt, David A. 90
Shankar, Natarajan 315
Simonsen, Jakob Grue 403
Sirjani, Marjan 20
Song, Miao 144
Spalazzi, Luca 223
Stehr, Mark-Oliver 110, 207
Subramani, K. 262

Talcott, Carolyn 110, 207, 434

Venkatasubramanian, Nalini 207, 352
Vrist, Søren Bjerregaard 403

Wirsing, Martin 241

Yu, Yue 144

Zaniewski, Lee 262