AUTONOMOUS, MODEL-BASED DIAGNOSIS AGENTS

THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE

AUTONOMOUS, MODEL-BASED DIAGNOSIS AGENTS

by

Michael Schroeder
Universität Hannover
Hannover, GERMANY

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4613-7629-3 ISBN 978-1-4615-5739-5 (eBook)
DOI 10.1007/978-1-4615-5739-5

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Printed on acid-free paper.

To Nicola

Contents

List of Figures

Acknowledgments

First of all, I'd like to thank Prof. Dr. Wolfgang Nejdl who supervised my master's thesis and encouraged me to continue research as a PhD student. I am grateful for our stimulating discussions and the deep insights he gave me concerning model-based diagnosis. Our work led to a number of publications and his continuous support allowed me to attend international workshops and conferences and to spend a year abroad at our partner institute in Lisbon.

I am thankful to Prof. Dr. Lipeck, my second supervisor, for the helpful comments on a first draft of my thesis.

Next, I'd like to thank all my collaborators and co-authors. My special thanks go to my dear colleague Peter Fröhlich with whom I always worked closely together since writing my master's thesis. Togehter we managed the ups and downs of University life. Then there are various Portuguese scientists I am thankful to: Prof. Dr. Luis Moniz Pereira, who supported my stay in Portugal and taught me about logic programming; Dr. Carlos Dámasio, to whom I owe my knowledge on revision of logic programs; Iara Móra, with whom I had wonderful discussions on fault-tolerant diagnosis and argumentation of agents; Prof. Dr. Alferes, from whom I learned some secrets about extended logic programs, and Rui Marques and Prof. Dr. Cunha, who told me all about distributed computing.

In Lisbon and back in Germany, I worked intensively with Dr. habil. Gerd Wagner who critically read my writings and gave me very profound insights into logic and agents. He has always been a shining example to me with his scientifically founded and courageous ways of thinking. Furthermore, I'd like to thank Daniela Plewe, whom I also met in Lisbon. She showed me another approach to science: arts. Together with Andreas Raab, we had Hamlet argue automatically on the internet. Another thanks to Ralf Schweimeier and Matthias Berger for their hints to improve my style of writing and clarifying the relation of the axiom of choice and Zorn's lemma.

I'd like to thank all my colleagues and staff of the *Departamento da Informatica* of *Universidade Nova de Lisboa* and the *Institut für Rechnergestützte Wissensverarbeitung* at *Universität Hannover*. I will always remember the pleasant environment and the fun we had while working.

During the last three years, I was supported by several grants which made this work possible: My initial visits to Portugal were part of an INIDA project funded by the german DAAD and the portuguese JNICT. DAAD and Instituto Camões financed my one-year-stay in Portugal and the Minna-James-Heinemann-Stiftung funded my research in Hannover.

My warmest thanks go to my parents, sisters, and Nicola.

Preface

This book emerged from a PhD project carried out at University of Hannover, Germany and New University of Lisbon, Portugal. After getting in contact with our Portuguese partners during my master's thesis, I spent a year in Lisbon and finished my PhD two years later in Hannover. It was a great experience to be involved in several projects carried out at the institutes. In the *Proloppe* and *Padipro* projects, the Portuguese side, led by Prof. Pereira, focused on logic programming including issues such as new semantics, distribution, and applications. The German side, led by Prof. Nejdl, has been dedicated to model-based diagnosis and participated in the European project *ModelAge* to develop a common model of agents. A tedious task. Both institutes cooperated on logic programming for model-based diagnosis over a period of four years.

Based on this background, the objective of this work is the definition and implementation of an architecture for autonomous, model-based diagnosis agents. In this book, we first develop a logic programming approach for model-based diagnosis and introduce strategies to deal with more complex diagnosis problems. Then we embed the diagnosis framework into the agent architecture of vivid agents.

First, we survey extended logic programming and show how this expressive language is used to model diagnosis problems stemming from applications such as digital circuits, traffic control, integrity checking of a chemical database, alarm-correlation in cellular phone networks, diagnosis of an automatic mirror furnace, and diagnosis of communication protocols. To compute diagnoses we review a bottom-up algorithm to remove contradictions from extended logic programs and substantially improve it by top-down evaluation of extended logic programs. Both algorithms are evaluated in the circuit domain including some of the ISCAS85 benchmark circuits.

To deal with complex diagnosis problems we lift the idea of model-based diagnosis to the meta-level of the diagnostic process and define a strategy language that allows a declarative description of the diagnostic process. Taking into account both

practical needs and rigorous formal treatment, we define syntax and declarative and operational semantics of the strategy language. With the concept of deterministic and non-deterministic as well as monotonic and non-monotonic strategies, we design a strategy knowledge base for circuit diagnosis with strategies for structural refinement, choice of models, measurements, and preferences. We evaluate the knowledge base and the algorithm on a voter circuit which is part of the benchmark circuits.

Based on the inference engine lined out above we turn to the autonomous agent's behaviour specification. We present the concept of vivid agents which comprise a vivid knowledge system and reaction and action rules to specify the agent's reactive and pro-active behaviour. To realise vivid agents we develop an architecture for concurrent action and planning. For implementation we use PVM-Prolog that provides coarse-grain parallelism to spawn agents in a network and fine-grain parallelism to run action and planning component concurrently. The interpreter is evaluated in distributed diagnosis where we implement fault-tolerant diagnosis and diagnosis of a communication protocol. The agent interpreter satisfies the requirements for a state-of-the-art multi-agent programming language: it supports reactive and pro-active behaviour specification; the specifications are executable; the language has a formal semantics; the modular design facilitates plug and play according to the problem domain; the system is open to heterogeneous agents based on other concepts and languages.

This comprehensive in-depth study of concepts, architecture, and implementation of autonomous, model-based diagnosis agents will be of great value for researchers, engineers, and graduate students with a background in artificial intelligence. For the practitioners it provides three main contributions: first, many examples from diverse areas such as alarm correlation in phone networks to inconsistency checking in databases; second, an architecture to develop agents; and third, a sophisticated and declarative implementation of the concepts and architectures introduced. The theorist can benefit from the three contributions of a novel approach to diagnosis based on logic programming, a newly devised modal strategy language to model diagnostic processes, and a practical and formally under-pinned concept of agents.

Michael Schroeder, February 1998

1 INTRODUCTION

Intelligent agents and multi-agent systems are one of the most important emerging technologies in computer science today. Agents have high level interaction capabilities and are capable of rational autonomous action in dynamic, unpredictable, typically multi-agent environments. Agents are being applied in domains as different as telecommunication networks, air-traffic control, space crafts, etc.

For example, NASA's Deep Space One Mission challenges information technology to achieve a paradigm shift in space missions [PBC+97]. While previous missions have been large, long-term projects being controlled from earth, NASA now aims at down-sizing. The challenge of NASA's New Millennium Program is the realization of deep space missions with minimal costs and maximal autonomy of the space craft. Williams and Nayak argue that such systems "will need to be programmable purely through high level compositional models. Self modeling and self configuration, coordinating autonomic functions through symbolic reasoning, and compositional, model-based programming are the three key elements of a *model-based autonomous systems* architecture that is taking us into the *New Millennium*" [WN96]. Pell et al. also emphasise that model-based diagnosis and agent technology are a necessary prerequisite for the success of such missions [PBC+97].

M. Schroeder, *Autonomous, Model-Based Diagnosis Agents*

The reason is twofold. First, model-based reasoning, and in particular model-based diagnosis, enjoy the advantage of being declarative so that they facilitate easy maintenance and extendibility of the systems at minimal development costs [DH88]; second, agent technology guarantees maximal autonomy of the systems which is essential for operation in unpredictable environments [WJ95].

In this book, we will develop the theoretical framework of autonomous, model-based diagnosis agents along with an implementation. The work will be motivated and accompanied by examples stemming from practical problems including domains such as digital circuits, fault management in cellular phone networks, traffic control, unreliable communication protocols, inconsistent databases, and fault-tolerant diagnosis. We will commence by defining a layered architecture for diagnostic agents comprising a knowledge base, an inference engine and a layer for communication and control. The core of the inference engine is REVISE, a diagnosis system for extended logic programs. Next, there is a component for diagnosis strategies which allows to describe the diagnosis process declaratively and which is vital to deal with complex problems. Finally, we introduce the new concept of vivid agents which caters for high-level executable specifications of agents' reactive and pro-active behaviours.

1.1 MOTIVATION

Before we go into detail, let us motivate the three important techniques of model-based diagnosis, diagnosis strategies, and autonomous agents.

1.1.1 Model-based Diagnosis

Model-based diagnosis is a prominent area within Artificial Intelligence and emerged in the last ten years [HCd92]. The basic principle in model-based diagnosis is the description of a system as a causal model. With the model at hand, the behaviour predicted by the model is compared to the actually observed behaviour. Since the predictions of the model are based on the assumption that the components work correctly, these assumptions may be partially dropped to accommodate for a detected behaviour difference and thus diagnose faulty behaviour.

Example 1.1 *Automatic Mirror Furnace [TD95, The95]*
Consider an automatic mirror furnace [TD95, The95] whose structure is depicted in Figure 1.1. It is used in space for material science research of crystal growth under micro-gravity conditions. About 20 crystal samples are placed on a disc from where they can be put into the focus of an ellipsoid mirror.

Have a look at the disc drive unit of the sample exchange mechanism in Figure 1.2. The unit consists of five components: A control unit that operates the plate and the motor and two switches. Switch 4 indicates whether a sample is at a position to be put into mirror focus and switch 5 shows whether this sample has the number 0.

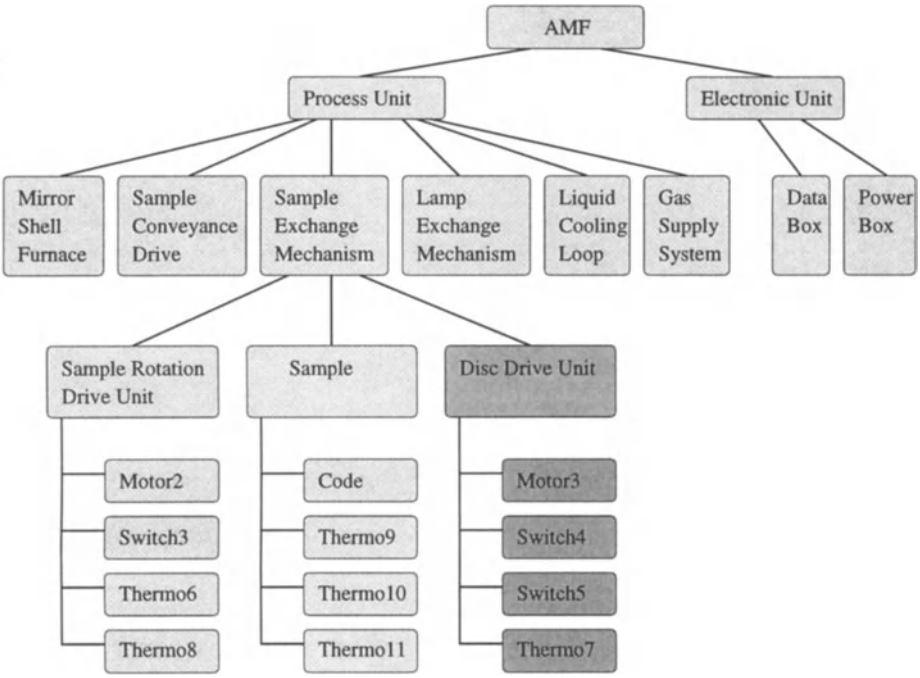


Figure 1.1. Schematics of an automated mirror furnace (AMF) including disc drive unit [TD95].

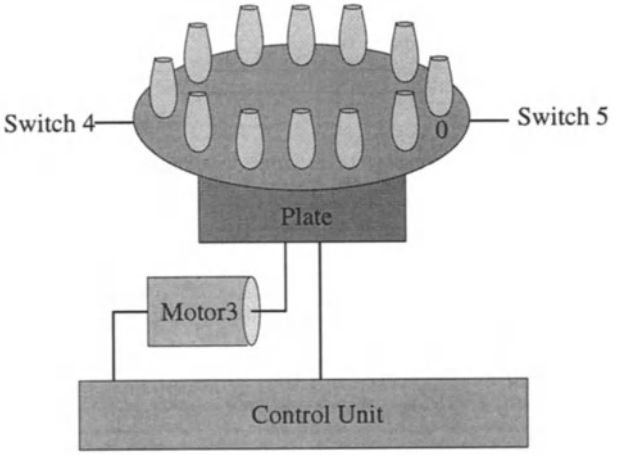


Figure 1.2. Model of the sample exchange disc drive unit.

Imagine we observe that the control unit enables the plate and starts the motor with half a position per time unit. Furthermore, we know that initially position 0 is in place so that both switch 4 and 5 should be active. And indeed, if we look at the observed behaviour of the switches in Figure 1.3 everything seems to be fine. Based on the assumption that all components work correctly, the model predicts that sample number 1 is in place after two time units and that therefore switch 4 is active and switch 5 is off. But both are observed to be off.

A possible diagnosis to explain the behaviour difference is that switch 4 is faulty. The alternative solutions of a disabled plate or stopped motor are no valid diagnoses, since then sample 0 would be in position and subsequently both switches would be active. But assuming both of them faulty contradicts the principle of parsimony [Rei87], e.g. to prefer minimal solutions, as switch 4 by itself is already a diagnosis. However, there is another explanation. The motor may suffer from friction so that it did move the samples but not at half a position per time unit. In this case, the position is somewhere between 0 and 1 which perfectly explains both switches being off and thus is a diagnosis.

In contrast to other diagnosis approaches such as heuristic or case-based diagnosis, model-based diagnosis enjoys the following four important properties [DH88]. First, it is declarative, so that the model can be described at an abstract level hiding unnecessary details. Second, it allows easy maintenance and extension of the model. Further components can be added to the model without re-implementing the diagnoser from scratch. Third, the model gives valuable insight into a system and supports prediction

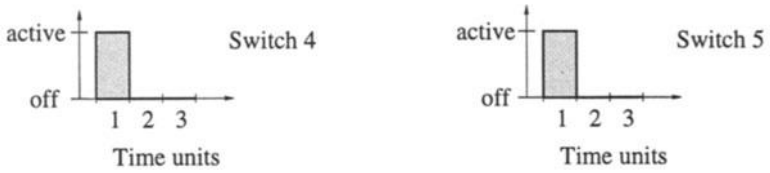


Figure 1.3. Observations of switch 4 and 5 of the disc drive unit.

of expected behaviour for future states. Fourth, the model features explanation of its findings which is vital to judge the diagnosis system's quality and to trust in it. We will further elaborate background, techniques, and algorithms of model-based diagnosis in chapter 2 and 3. We will demonstrate how to employ extended logic programming, an expressive language that provides two forms of negation and integrity constraints, in model-based diagnosis.

1.1.2 Strategies in Model-based Diagnosis

In the last years, model-based diagnosis has been extended by the introduction of the new concept of using different diagnostic assumptions which can be activated during the diagnostic process [Str92b]. The selection of the appropriate diagnostic assumptions and system models during the diagnostic process is controlled by a set of working hypotheses [Str92b, BD93, BD94]. Diagnostic strategies are rules defining which working hypotheses should be used in a given situation during the diagnostic process.

Example 1.2 *Structural Refinement of a Voter*
Consider the digital circuit in Figure 1.4. The circuit realises a voter with input a, b, c which outputs b if $a = b$ or $b = c$ and otherwise c. Initially, we view the voter as a whole unit and detect a malfunctioning. A first refinement step reveals that the selection component sel and the component voter_ab are involved whereas the component voter_bc is found ok. By adopting an according working hypothesis, the suspicious devices are further refined and again two candidates, sel0 and equ_ab0, are detected, while the other components are discarded. A last refinement step leads to the final diagnosis of or-gate or0_sel and and-gate andA being abnormal.

The use of hierarchies and abstractions allows to reduce the complexity of diagnosing. In the above example, it has only been necessary to consider half the components of the flat gate-level of the voter. If we take into account that computing all diagnoses is an NP-complete problem then such a use of hierarchies facilitates the diagnosis of large systems. Besides the use of hierarchies, a diagnosis process can use further strategies, such as preference of single over double faults, measurements to further dis-

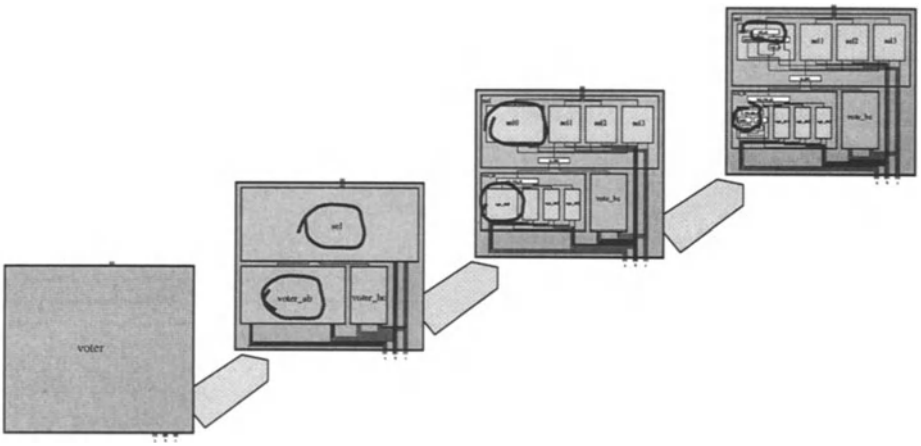


Figure 1.4. A diagnostic process.

tinguish among diagnoses, choice of another model in case of implausible diagnoses, and refinement of a component's behaviour to further elaborate diagnoses.

1.1.3 Autonomous Agents

Agents and multi-agent systems have been an increasingly active area throughout the last five years. It attracted many researchers with different backgrounds and subsequently, a variety of notions of agenthood have been defined. Our notion is based on the ideas introduced by Shoham [Sho93] and follows the traditional notion of a logic-based rational agent. Shoham extends the concept of object-orientation and introduces the corresponding agent-oriented programming. He defines an agent as "an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments" and he pinpoints: "What makes any hardware or software component an agent is precisely the fact that one has chosen to analyze and control it in these mental terms" [Sho93]. Apart from this weak notion of agenthood, we favour the approach of logic-based rational agents coming from logic-programming. Kowalski promotes such agents and attempts to reconcile this notion "with the contrary notion of a reactive agent which acts 'instinctively' in response to conditions that arise in its environment" [Kow95]. Wagner's concept of a vivid agent points to a similar direction emphasising that beliefs, tasks, and goals are represented by a knowledge base and are subject to manipulation via update and inference operations. The agent's behaviour is represented by means of *action* and *reaction rules* [Wag96]. Wooldridge and Jennings summarise that agenthood is determined by four properties [WJ95]:

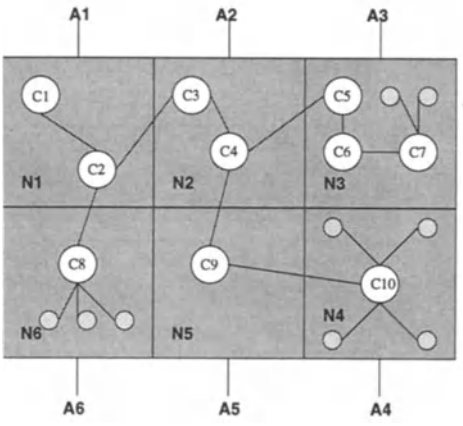


Figure 1.5. A communication network.

1. autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;

2. social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;

3. reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;

4. pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.

Example 1.3 *Distributed Diagnosis of a Computer Network*
As an example of agents for distributed diagnosis, have a look at the communication network in Figure 1.5. Due to size and complexity, large communication networks cannot be diagnosed by a central system, so that diagnosis has to be performed locally whenever possible.

A solution is to introduce agents which autonomously monitor and diagnose a subnet of feasible complexity. An agent has detailed knowledge about its own subnet and an abstract view of the rest of the network. Agent A_1, for example, knows its components C_1 and C_2 and has routing information to send messages. Being social, agent A_1 passes the message to agent A_2 in order to send a message to C_7. Thus, A_1 has only

an abstract view of the network including agents A_2 and A_6, but excluding A_3,A_4,A_5. Now in case A_1 notices that a message was lost, it reacts immediately and diagnoses itself and comes either to the conclusion that it is down itself or that A_2 is suspicious. In the latter case it pro-actively contacts A_2 to further investigate the cause of failure.

1.2 MAIN CONTRIBUTIONS

The main goal of this book is to define and implement a framework for autonomous, model-based diagnosis agents. The contributions of this work are fourfold. First, we show how to use extended logic programming for model-based diagnosis. We identify modelling techniques for diagnosis problems based on a variety of practical applications. To compute diagnoses we develop a top-down diagnosis algorithm which is implemented in the REVISE system. The algorithm is evaluated on some benchmark problems.

Second, the notion of a dynamically developing diagnosis process has been coined recently. We devise a new strategy language to specify such diagnosis processes. We define syntax and semantics of the language and identify principles to design strategies for applications. We define an operational semantics and an efficient algorithm which is evaluated in the domain of circuit diagnosis.

Third, to deal with distributed diagnosis we introduce and implement the concept of vivid agents. For the implementation we line out an architecture for concurrent action and planning and we implement the architecture using a distributed Prolog. The complete agent interpreter is evaluated in the domain of distributed diagnosis. It satisfies the main criteria for a state-of-the-art multi-agent programming language: It allows to specify reactive and pro-active behaviour, it is under-pinned by a formal semantics, the high-level agent specifications are executable, the agents are hardware independent, and heterogeneous agents are supported.

Forth, we introduce the novel architecture for autonomous, model-based diagnosis agents shown in Figure 1.6. It consists of three layers: a knowledge base, an inference engine, and a top-layer for communication and control. Besides the system description, the knowledge base contains the strategic knowledge, and action- and reaction rules which specify the agent behaviour. The core of the inference engine is the diagnosis system REVISE. A strategy algorithm that computes the diagnostic process operates on top of the diagnosis engine. Next, there is an optional test component to perform hardware tests and evaluate sensors. The planner takes the action rules and a goal as input and computes action sequences that achieve the desired goal. Finally, the top-layer for communication and control realises a perception-reaction-cycle which checks incoming events and reacts according to the reaction rules. It also exhibits pro-active behaviour as it communicates goals to the planner and executes the generated plans.

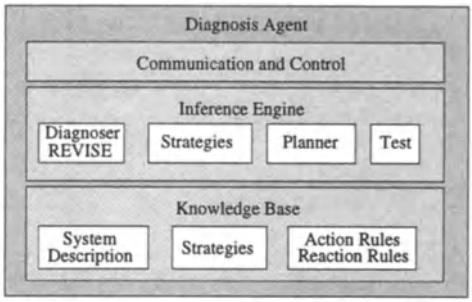


Figure 1.6. Components of a new architecture for autonomous model-based diagnosis agents.

1.3 ORGANISATION

Chapter 2 surveys model-based diagnosis including terminology, basic definitions, modelling techniques, and diagnosis engines.

Chapter 3 deals with the use of logic programming for model-based diagnosis. We introduce extended logic programs that provide two kinds of negation and integrity constraints. Using this expressive language, we further elaborate modelling techniques for the system description based on practical examples such as digital circuits, fault management in cellular phone networks, an automated mirror furnace, traffic control, unreliable communication protocols, and inconsistent databases. With the examples in mind, we turn towards computing diagnoses. We review a bottom-up algorithm for contradiction removal and develop a dramatically improved top-down algorithm using top-down evaluation of extended logic programs. The algorithms are implemented in the REVISE system and we conclude this chapter by comparing the algorithms' efficiency. Parts of chapter 3 are based on [DPS97b] co-authored by C. V. Damásio and L. M. Pereira.

Chapter 4 discusses a framework which allows to express diagnosis strategies as formulas of a meta–language. We define syntax and semantics of the language and present a method for designing strategy knowledge bases as well as an efficient, straightforward operational semantics for exploiting them. We evaluate the results by a comprehensive example for the diagnosis process of digital circuits. Parts of chapter 4 are based on [NFS95, FNS96, FNS97] co-authored by P. Fröhlich and W. Nejdl.

Chapter 5 introduces vivid agents to develop distributed monitoring and diagnosis systems consisting of a variety of scalable knowledge- and perception-based agents. We develop an execution model for vivid agents which is based on an architecture for concurrent action and planning. We implement vivid agents in PVM-Prolog which is

based on the Parallel Virtual Machine and provides coarse-grain parallelism to spawn agents in a network, and fine-grain parallelism to run an agent's perception-reaction-cycle and planning facility concurrently. Next, we evaluate the concept of vivid agents in distributed diagnosis including fault-tolerant diagnosis and the diagnosis of an unreliable communication protocol. Parts of chapter 5 are based on [SW97, SMWC97] co-authored by G. Wagner, R. Marques, and J. Cunha.

Chapter 6 rounds out the picture by comparing to related work, evaluating the previous chapters, and pointing to future work.

2 MODEL-BASED DIAGNOSIS

In this chapter we give an overview over model-based diagnosis including terminology, basic definitions, modelling techniques, and diagnosis engines.

2.1 HEURISTIC DIAGNOSIS VS. MODEL-BASED DIAGNOSIS

In the past, two different approaches to diagnosis emerged, namely heuristic diagnosis and model-based diagnosis. In heuristic diagnosis expert knowledge and large data samples are used to define rules of thumb to relate symptoms to diagnoses. Heuristic diagnosis has been widely applied in the medical domain and the most prominent system, called MYCIN, attempts to recommend appropriate therapies for patients with bacterial infections. The first principle in systems like MYCIN [BS84] are certainty factors to indicate how relevant a symptom is for a diagnosis. The factors range from -1 to 1, where -1 means uncertain, 0 unknown, and 1 certain. Given a set of symptoms the corresponding certainty factors are combined to give an overall estimation for the certainty of a diagnosis.

M. Schroeder, *Autonomous, Model-Based Diagnosis Agents*

Symptom		Diagnoses		
		Morbus Paget	Lipom	...
a-priori		+0.0300	+0.0012	
localisation:	hand	-0.0875	-0.1000	
x-ray:	homogeneous concentration	-1.0000	+0.0300	
age:	51-60	+0.0583	+0.0833	
...				

Figure 2.1. Certainty factors for bone tumors.

Example 2.1 *CARAT, Diagnosis of Bone Tumors*
CARAT [PBT$^+$93, PBT$^+$94] is a heuristic diagnosis system for bone tumors. Some certainty factors are shown in Figure 2.1. Morbus Paget and a Lipom have a-priori certainty of 0.03 and 0.0012, respectively. With homogeneous concentrations on the X-ray Morbus Paget turns out to be highly uncertain, while a Lipom is slightly positive. Given a patient of 55 years with a tumor on his/her hand and an X-ray with homogeneous concentrations we can apply the following formula to obtain the overall rating for the diagnoses:

$$cf(d, s_1 \wedge s_2) = \begin{cases} cf_1 + cf_2 - cf_1 * cf_2, & \text{if } cf_1, cf_2 \geq 0 \\ cf_1 + cf_2 + cf_1 * cf_2, & \text{if } cf_1, cf_2 < 0 \\ \frac{cf_1 + cf_2}{1 - min(|cf_1|, |cf_2|)}, & \text{otherwise} \end{cases}$$

where cf_1 and cf_2 are the certainty factors $cf(d, s_1)$ and $cf(d, s_2)$ for a diagnosis d and symptoms s_1, s_2. With the symptoms of the 55 year-old patient Morbus Paget is rated -1 and a Lipom 0.0132. Thus, we can definitely rule out Morbus Paget, whereas the Lipom is still unknown.

Though heuristic diagnoses are easily computable, the approach suffers from some severe weaknesses. It is difficult to get good heuristics, because experts often do not have the technical skills to define them. It is, for example, crucial to fully use the interval of possible certainties to get the best discrimination between the diagnoses. Once the knowledge about certainties is given, it is difficult to extend. For example, in the CARAT project 43 tumors have been classified with around 65.000 factors. If a new tumor is found one day, it will be difficult to relate the new certainty factors to the old ones. And last, not least, the approach is not theoretically founded so that it is

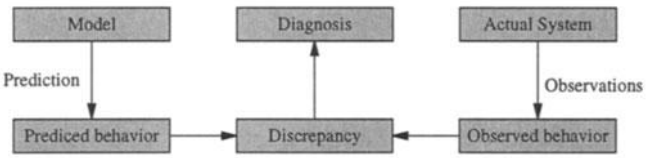


Figure 2.2. Model-based diagnosis.

not possible to estimate the quality of the systems' findings which is essential for its acceptance by medical staff and patients.

Model-based diagnosis differs significantly from heuristic diagnosis. The first principle of model-based diagnosis is a model or artifact of the system to be diagnosed. As shown in Figure 2.2 the model's predicted behaviour is compared to the observed behaviour of the actual system. If they contradict each other, the model's default assumption of working components has to be revised to remove the discrepancy.

2.2 THE MODEL

Initially, research in model-based diagnosis concentrated on algorithms to compute diagnoses, but soon it turned out that modelling is a hard problem, too [HCd92]. Basically, the model contains the system's components and their structure. When troubleshooting digital circuits, a domain to which model-based diagnosis has been tightly connected [Dav84, dKW87, Rei87, Ham91, dKW89], the model consists of gates and their connections.The components have input and output ports and transform a given input according to their functionality. The functionality depends on the behaviour mode of the component. Two modes are always present, namely ok and abnormal where the former is usually assumed by default. The abnormal mode may be refined by fault modes [SD89] that describe the malfunctioning more precisely.

In the following example we use first order logic as notation. Variables begin with upper-case letters and constants with lower-case letters. Variables are assumed to be universally quantified.

Example 2.2 *Modelling a Full Adder*
Consider the eight-bit-full-adder in Figure 2.3. The components of the circuit are classified by predicates

$$type(x_{11}, xor), \quad type(a_{11}, and), \ldots$$

The components are connected which is captured by the predicate conn:

$$conn(o_{11}, in_1, a_{12}, out_1), \quad conn(o_{11}, in_2, a_{11}, out_1), \ldots$$

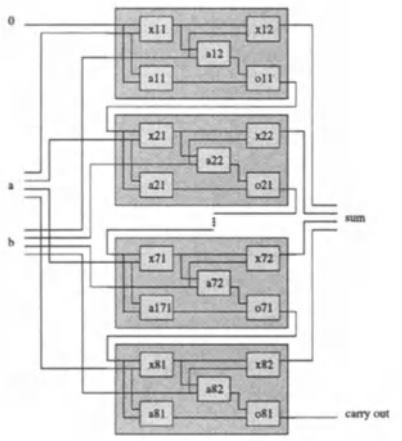


Figure 2.3. Eight-bit-adder as functional model.

Given a component, values may be either observed, passed via connections, or computed.

$$val(C,P,V) \leftarrow obs(C,P,V)$$
$$val(C,P,V) \leftarrow conn(C,P,C',P'),val(C',P',V)$$
$$val(C,P,V) \leftarrow \neg obs(C,P,V'),comp\_val(C,P,V)$$

Observations are given as facts, while the computed values are calculated:

$$comp\_val(C,out_1,V) \leftarrow \neg ab(C),type(C,and),\ val(C,in_1,V_1),val(C,in_2,V_2),\ and\_table(V_1,V_2,V)$$

To constrain the behaviour difference we write

$$\bot \leftarrow obs(C,P,V),\neg comp\_val(C,P,V)$$

where $\bot$ denotes the bottom symbol for false.

To introduce fault modes we could express that gates may be stuck at 0 or 1 denoted by s_0,s_1. So if we detect a component as abnormal we require it to adopt a fault mode by

$$ab(C) \rightarrow s_0(C) \vee s_1(C)$$

For the fault modes we have to define how to compute output values. For example, does s_0 always yield the output value 0.

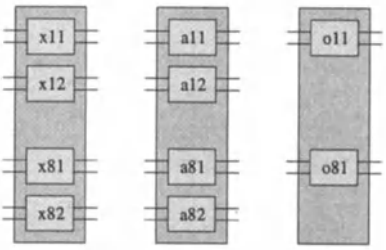


Figure 2.4. Eight-bit-adder as physical model.

$$comp\_val(C, out_1, 0) \leftarrow type(C, and), s_0(C)$$

Besides the functional view of the full adder we may want to use a physical view as shown in Figure 2.4, as well. In contrast to the functional view, the physical view takes the location of the gates additionally into account. To put both views in the same system description we add to all rules of a view a working hypothesis which has to be set to activate the rules in the model. To compute values in the physical view we have rules whose body contains the working hypothesis physical_view.

$$\begin{aligned} comp\_val(C, out_i, V_i') \leftarrow \quad & \neg ab(C), type(C, and\_chip), \\ & physical\_view, \\ & val(C, in_1, V_1), \ldots, val(C, in_8, V_8), \\ & and\_chip\_table(V_1, \ldots, V_8, V_1', \ldots, V_8') \end{aligned}$$

2.3 CONSISTENCY-BASED AND ABDUCTIVE DIAGNOSIS

The birth of model-based diagnosis can be traced back to de Kleer's and Reiter's seminal papers "Diagnosing Multiple Faults" [dKW87] and "A Theory of Diagnosis from First Principles" [Rei87], respectively. Reiter's paper introduces the first principles of model-based diagnosis, the terminology, and an algorithm which was corrected in [GSW89]. Reiter's initial proposal is consistency-based in contrast to abductive diagnosis which was coined later. In consistency-based diagnosis only the two behaviour modes *ab* and $\neg ab$, i.e. ok, are present. Using these modes, the system description *SD* is modelled as a first-order formula. Given a set *OBS* of observations and the components *COMPS* the triple $(SD, COMPS, OBS)$ is dubbed a diagnosis problem. Adding the observations may lead to inconsistencies, i.e. $SD \cup OBS \models \bot$, which can be removed by adding abnormality assumptions to $SD \cup OBS$. There is always the trivial solution to gain consistency back by assuming all components to be abnormal. But the principle of parsimony requires that subsets of diagnoses should not be diag-

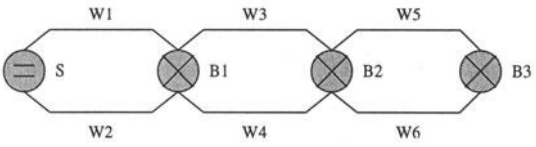


Figure 2.5. Three bulbs and one voltage supply in parallel.

noses, as well. Hence, a minimal set of components for which assuming abnormality re-establishes consistency is called a consistency-based diagnosis.

Definition 2.3 *Consistency-based Diagnosis*
A consistency-based diagnosis *for a diagnostic problem* $(SD, COMPS, OBS)$ *is a minimal set* $\Delta \subseteq COMPS$ *such that*

$$SD \cup OBS \cup \{ab(c) \mid c \in \Delta\} \cup \{\neg ab(c) \mid c \in COMPS \setminus \Delta\}$$

is consistent.

Note, that other authors deviate from the term consistency-based diagnosis. Konolige [Kon92], e.g., uses the term excusing diagnosis, which comes from the fact that the assumption about the malfunction of a component excuses the predicted correct behaviour. Console and Torasso [CT91] use the term explanation as consistency.

Example 2.4 *Diagnosing the Full Adder*
Consider the full adder in Figure 2.3 and its system description SD as developed above. We observe for input $a = b = 0$ *that the sum output is 0 as expected, but that the carry out is set 1. Therefore* $SD \cup OBS \models \bot$, *where* $OBS = \{val(a_i, out_1, 0), val(b_i, out_1, 0), val(sum_i, in_1, 0 \mid 1 \leq i \leq 8\} \cup \{val(carry\_out, in_1, 1)\}$. *The three diagnoses* $\{a_{81}\}, \{a_{82}\}, \{o_{81}\}$ *remove the contradiction.*

Consistency-based diagnosis is a weak notion of diagnosis. Rather than explaining the observed behaviour by the model, only contradictions are removed. The advantage is twofold: The model is easier to construct as only the intended functionality has to be modelled and consistency-based diagnoses are easier to compute. Consistency-based diagnosis is sufficient for many applications, but a variety of practical problems occur so that the basic approach has been further developed. The initial proposals [Dav84, Gen84, Rei87, dKW87, FN90] did, e.g., not deal with fault modes. De Kleer and Williams [dKW89] as well as Struss and Dressler [SD89] tackled this issue. The latter by means of the following example of physical impossibility [SD89].

Example 2.5 *Physical Impossibility*
A supply is connected to three bulbs b_1, b_2, b_3. *The bulbs* b_1, b_2 *are broken,* b_3 *is*

lit. Besides the diagnosis of b_1, b_2 being faulty, consistency-based diagnosis comes up with the unintended diagnosis of the supply and b_3 being faulty. The reason is that the faulty supply explains the broken bulbs, while assuming b_3 faulty accommodates for the light with no information on the supply.

Struss and Dressler solved this problem by introducing fault models: They model that a faulty supply does not supply energy so that the bulb b_3 cannot burn. Friedrich et al. [FGN90] propose rather to add impossibility axioms to the system description that state that bulbs cannot burn without supply which is less expressive but computationally more efficient. With the notion of fault modes a strong notion of diagnosis can be defined. Instead of only removing contradictions between predictions and observations, we require that the observed behaviour is explained. Poole's abductive diagnosis [Poo89] and Console and Torasso's explanation as covering [CT91] is defined as follows:

Definition 2.6 *Abductive Diagnosis*
An abductive diagnosis *for a diagnostic problem $(SD, COMPS, OBS)$ is a minimal set Δ of behavioural mode assumptions such that*

$$SD \cup \Delta \models OBS \text{ and } SD \cup \Delta \cup OBS \text{ is consistent}$$

2.4 DIAGNOSIS ENGINES

Based on this theory several diagnosis engines have been implemented.

GDE and Sherlock. One of the first and most influential diagnosis systems is GDE, the General Diagnostic Engine, by de Kleer and Williams [dKW87]. The initial GDE computes consistency-based diagnoses and thus there have been various extensions. A subsequent system, called Sherlock [dKW89] extends the GDE to include fault models and probabilistic information.

IMPLODE. To improve efficiency de Kleer and Brown introduced sensitivity analysis in the diagnosis engine SOPHIE III [dKB92]. Assumptions are judged wrt. their influence on output values and are classified as primary or secondary. SOPHIE III is specialised on analog circuits and computing single-fault diagnoses. Raiman, de Kleer, and Saraswat extend the approach of sensitivity analysis in their theory of critical reasoning [RdKS93]. Rather than assumptions they classify conflicts, i.e. sets of assumptions. Their diagnoser IMPLODE which incorporates critical reasoning on top of a diagnoser described in [dK91] performs very well even for large diagnosis problems such as the ISCAS benchmark circuits [BPH85] which have not been solved efficiently by previous engines.

MOMO. MOMO, a system developed by Friedrich and Nejdl [FN90], follows a different line of research. It generates consistent logical models for the inconsistent, underlying theory of system description and observations. MOMO is implemented in Prolog and uses hyper-resolution. The core is a one-page Prolog program.

DRUM. The same idea of working directly on logical models is the key of DRUM and DRUM II [NG94, NF97, FN97]. Unlike most other systems DRUM does not record justifications which can lead to an explosion of labels when solving complicated problems. It uses static pre-compiled information on the structure of the underlying theory and does no further recording during search. DRUM solves large complicated problems with attractive time and space complexity and outperforms the IMPLODE system on almost all examples.

REVISE. The non-monotonic reasoning system REVISE [DNP94, DNPS95, DPS96, DPS97b] that revises extended logic programs is based on logic programming with explicit negation and integrity constraints. It provides a two-valued revision of assumptions to remove contradictions from the knowledge base. It has been tested on a spate of examples with emphasis on model-based diagnosis.

2.5 SUMMARY

The first principle in model-based diagnosis is a model of the system to be diagnosed. Assuming that the components work correctly we can predict the behaviour of the system. If the predictions differ from the observations the default assumptions are revised to remove the contradiction. In contrast to other approaches model-based diagnosis enjoys several advantages:

1. Due to their declarative nature model-based diagnosis systems are easy to extend and to maintain.
2. No heuristics are needed and diagnoses can be explained without the recourse to rules of thumb.
3. Models are often already available from a constructor.
4. The model can predict the system's future states in advance and also explain unforeseen and unknown failures.
5. Model-based diagnosis is theoretically founded, so that the diagnoser's capability and quality can be tuned and judged beforehand.

3 LOGIC PROGRAMMING AND DIAGNOSIS

First we give a survey of the historical development of logic and logic programming. After introducing extended logic programming which comprises explicit as well as implicit negation and integrity constraints, we show how to model diagnosis problems. The domains range from digital circuits through traffic control, integrity checking of a chemical database, fault-management in mobile telecommunication, an automated mirror furnace to communication protocols. All these problems are solved by revising contradictory extended logic programs. Therefore we investigate algorithms for contradiction removal. We sketch a first algorithm which is based on bottom-up evaluation of extended logic programs and discuss in detail a second one which makes use of a top-down proof procedure. The algorithms are implemented in the REVISE system and we conclude by comparing them and evaluating their efficiency.

M. Schroeder, *Autonomous, Model-Based Diagnosis Agents*

3.1 EXTENDED LOGIC PROGRAMMING

3.1.1 A Brief History of the Predicate Calculus

Modelling diagnosis problems in logic is part of a much more profound question: Can human thought be automated? Is it possible to model the real world mathematically and then automate problem solving? The question was raised early. Leibniz (1646-1716) already formulated the two goals of a *calculus ratiocinator* and a *lingua universalis*. The calculus should free humans from computing (until the sixties "computer" was a *human* profession) while the universal language should be so powerful to express anything humans can think of [Eco94]. Of course, Leibniz did not achieve these two ambitious goals. But a hundred years later Boole (1815-1864) entered the stage and tied logic and mathematics in an algebra for logic that he developed in the *Laws of Thought* [Sch97]. Later Frege (1848-1925) leaped forward towards a calculus of human thought by defining a theory of quantification, now known as the predicate calculus, in his *Begriffsschrift* [Dav83]. Though he defined the first formal language his work was not recognised by his peers.

In fact, the formal treatment of logic and language and its use for mathematics started a controversy that came to a head in particular with Cantor's (1845-1918) new definition of infinity and transfinite numbers. The critics being famous mathematicians such as Kronecker (1823-1891), Poincaré (1854-1912), and Brouwer (1881-1966) founded the doctrine of intuitionism which views mathematics as the formulation of mental constructions that are governed by self-evident laws and therefore allows only constructive proofs. Counter-arguing intuitionism, Hilbert (1862-1943) set out the goal to define a formal calculus of classical mathematics based on Frege's and Peano's (1858-1932) work. Whitehead (1861-1947) and Russell (1872- 1970) accomplished a great part of the goal in their *Principia Mathematica* [BM92].

A critical point is whether such a theory is consistent. Hilbert demanded a constructive proof of consistency to accommodate with the intuitionists. In 1931, Gödel (1906-1978) answered the question when he published his incompleteness theorem. He proved that no formal system comprising arithmetics is expressive enough to prove its own consistency. Though this is a fundamental theoretical problem, logic developed further in practice. After World War II, the first theorem provers were developed together with the first computers, so that finally Leibniz' dream seemed to become reality. But the performance was poor since no optimisations were used. With the introduction of the resolution principle in the sixties and the use of Horn clauses which do not allow for negation, the situation changed and logic programming languages such as Prolog came up.

3.1.2 Why do we need more Expressiveness than Horn Clauses?

The main reason why logic programming was limited initially to Horn clauses is that most mathematical problems can be formulated as Horn clause programs and that they can be solved efficiently in linear time wrt. the instantiated program. But in the meantime logic programming has been explored in other domains such as deductive databases and knowledge-based systems, as well. In such domains the user does not have knowledge about internal implementation details, so that logic programs need an intuitive semantics. An essential characteristics of Horn clause programs is their monotonicity. Since only positive facts can be added to a program, previous conclusions are never abandoned. They persist, so that adding new facts always increases the overall conclusions. Furthermore, we cannot deduce negative conclusions from a Horn clause program. The only possibility is to assume a literal false if its proof fails in finite time. Such a treatment avoids by definition contradictions. Several problems cannot be solved by Horn clauses:

- Implicit Negation *not*:
  The qualification problem states that birds usually fly, but that there are exceptions. Listing all exceptions is tedious and requires that there is only a finite number. Therefore we want to assume by default that birds fly and possibly revise some conclusions in the light of new facts. We can solve this problem by introducing implicit negation *not*, which states that we assume a literal false if there is no evidence for the contrary. So we may state that a bird which is not abnormal flies, where we implicitly assume not-abnormality for any bird. For a penguin we could add the fact that it is abnormal and therefore block the conclusion of the rule.

- Explicit negation $\neg$:
  To formalise when a bus is allowed to cross a railway we may say that it should drive if there is no train. If the driver takes "no train" as an implicit negation he/she acts grossly negligent. Because with lack of evidence for the train he/she would cross the railway. But the lack of evidence for the train may be due to his/her shortsightedness. Being careful, he/she would prove explicitly that there is no train which would fail without glasses. So he/she has to put on his/her glasses and prove the absence of a train. Explicit negation $\neg$ copes with this problem. To relate it to implicit negation we demand a coherence principle that explicit negation implies implicit negation.

- Integrity Constraints and Revision:
  Diagnosing a system we require that predictions of the model may not differ from actual observations of the system. To put it in other words, there is a contradiction $\perp$ if predictions and observations differ. In general, we want besides rules that span a solution space for a program, integrity constraints that limit the space (see Figure 3.1). Once a constraint is violated we want to revise the program. If the contradiction is based on assumptions we can revise these assumptions until no constraint is violated anymore.

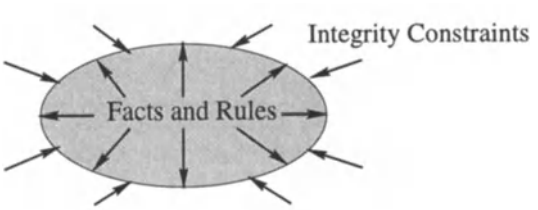


Figure 3.1. The rules of a program span the solution space, whereas the integrity constraints limit it.

Since Prolog became a standard in logic programming much research has been devoted to the semantics of logic programs which are enhanced by extensions such as implicit and explicit negation, disjunction, and integrity constraints [GL90, Wag91, PA92, Wag94]. Well-founded semantics [GRS88, Gel89, GRS91] turned out to be a promising approach to cope with negation by default. Subsequent work extended well-founded semantics with a form of explicit negation and constraints [PA92, AP96] and showed that the richer language, called WFSX, is appropriate for a spate of knowledge representation and reasoning forms [PAA93, PDA93, ADP95]. In particular, the technique of contradiction removal from extended logic programs [PAA91] opens up many avenues in model-based diagnosis [PDA93, DNP94, DNPS95, MA95].

3.1.3 Syntax of Extended Logic Programs

To define the syntax of extended logic programs we need some basic definitions:

Definition 3.1 *Term, Ground*
Let C, V, F, P be disjunctive sets of constants, variables, functors, and predicates, respectively. The set T of terms is defined as follows. If $t \in C$ *or* $t \in V$, *then* $t \in T$ *is a* term, *i.e. constants and variables are terms. Let* $t_i \in T$ *be terms and* $f \in F$ *be an n-ary function symbol, then* $f(t_1, \ldots, t_n) \in T$ *is a* term. *Nothing else is a term. A term is called* ground *if it does not contain any variables.*

Definition 3.2 *Herbrand Universe, Atom, Default and Objective Literal*
Let $p \in P$ *be a predicate symbol and* $t_i \in T$ *terms, then* $p(t_1, \ldots, t_n)$ *is an* atom. *Nothing else is an atom. The set of all ground atoms is called* Herbrand universe $\mathcal{H}$. *A* literal *is either an atom A or its explicit negation* $\neg A$ *or its implicit negation not A. The latter is also called* default literal, *whereas the two former are* objective literals.

Now we can define the syntax of extended logic programs following [PA92, AP96].

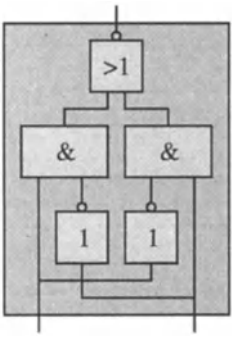


Figure 3.2. Part of a voter circuit.

Definition 3.3 *Extended Logic Program*
An extended logic program is a possibly infinite set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n \quad (0 \leq m \leq n)$$

where each L_i is an objective literal.

How can we describe systems to be diagnosed as extended logic programs? Imagine a digital circuit as depicted in Figure 3.2. We have to describe the behaviour of the components and the topology of the circuit. To express the assumption that the components work correctly by default we use implicit negation.

Example 3.4 *Computing Output Values of a Component*
In order to compute output values of a component the system description contains rule instances of the following form:

$$\begin{aligned} value(out(Comp,k),Out_k) \leftarrow\ & mode(Comp,Mode), \\ & value(in(Comp,1),In_1),\ldots,value(in(Comp,n),In_n), \\ & table(Comp,Mode,In_1,\ldots,In_n,Out_1,\ldots,Out_k,\ldots,Out_m) \end{aligned}$$

where table consists of facts that describe the input/output relation of the component wrt. to the current mode. The behaviour of an and-gate with two input ports and one output port is captured by

$$\begin{aligned} &table(and,ok,0,0,0) \\ &table(and,ok,0,1,0) \\ &table(and,ok,1,0,0) \\ &table(and,ok,1,1,1) \end{aligned}$$

Faulty behaviour is expressed similarly. The mode ab is a general fault mode, whereas s_0 and s_1 are additional fault modes indicating that the gate is stuck at 0 and 1, respectively.

$$
\begin{array}{lll}
table(and,ab,0,0,1) & table(and,s_0,0,0,0) & table(and,s_1,0,0,1) \\
table(and,ab,0,1,1) & table(and,s_0,0,1,0) & table(and,s_1,0,1,1) \\
table(and,ab,1,0,1) & table(and,s_0,1,0,0) & table(and,s_1,1,0,1) \\
table(and,ab,1,1,0) & table(and,s_0,1,1,0) & table(and,s_1,1,1,1)
\end{array}
$$

To express the default assumption that the components are working fine we use negation by default:

$$mode(Comp,ok) \leftarrow not\ mode(Comp,ab)$$

The above schema captures the behaviour of a single component. Additionally, we have to code the propagation of values through the system. Given the causal connection of the system as relation conn(N,M), where N and M are nodes, we express that a value is propagated through the causal connection or else observed:

$$
\begin{array}{l}
value(N,V) \leftarrow conn(N,M), value(M,V) \\
value(N,V) \leftarrow obs(N,V)
\end{array}
$$

These modelling concepts allow predicting the behaviour of the system in case it is working fine. To express that normality assumptions may lead to contradiction between predictions and actual observations we introduce integrity constraints.

Definition 3.5 *Integrity Constraint*
An integrity constraint has the form

$$\bot \leftarrow L_1,\ldots,L_m,not\ L_{m+1},\ldots,not\ L_n \quad (0 \leq m \leq n)$$

where each L_i is an objective literal and $\bot$ stands for false.

Syntactically, the only difference between the program rules and the integrity constraints is the head. A rule's head is an objective literal, whereas the constraint's head is $\bot$, the symbol for false. Semantically, the difference is that program rules open the solution space, whereas constraints limit it.

Example 3.6 *Integrity Constraint*
Now we can express that a contradiction arises if predictions and observations differ. In the setting of digital circuits we use the constraints:

$$\bot \leftarrow value(N,0), obs(N,1) \qquad \bot \leftarrow value(N,1), obs(N,0)$$

A contradiction is always based on the assumption that the components work fine, i.e. the default literals are false. In general, we can remove a contradiction by partially dropping some closed world assumptions. Technically, we achieve this by adding a minimal set of revisable facts to the initially contradictory program:

Definition 3.7 *Revisable, Revision*
The revisables *R of a program P are a subset of the default literals which do not occur as rule heads in P. The set $R' \subseteq$ not R is called a* revision *if it is a minimal set such that $P \cup R'$ is free of contradiction.*

The limitation of revisability to default literals which do not occur as rule heads is adopted for efficiency reasons, but without loss of generality. We want to guarantee that the truth value of revisables is independent of any rule. Thus we can change the truth value of a revisable whenever necessary without considering an expensive derivation of the default literal's truth value.

Before we show how to compute revisions we further motivate the need for model-based diagnosis by sketching some practical problems solved with model-based diagnosis and extended logic programming.

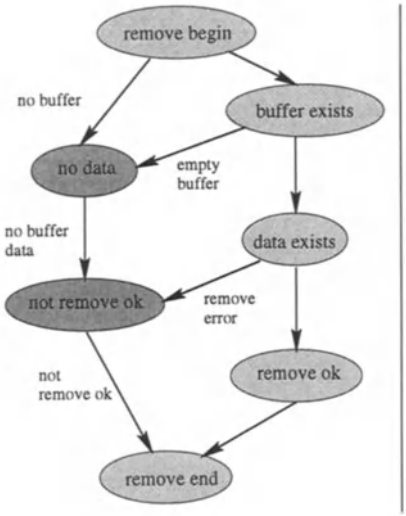
3.2 MODELLING DIAGNOSIS PROBLEMS

In this section we briefly describe applications for model-based diagnosis in traffic control [SNH94, Sch94], integrity checking of a chemical database [Ham94], alarm-correlation in mobile telecommunication [FNJW97], diagnosis of the automated mirror furnace [TD95, The95], and diagnosis of an unreliable communication protocol [FdAMNS97]. The code of the examples is available online and can be run over the internet (see [DPS97a]).

3.2.1 Traffic Control

Traffic increases faster than road networks do [SNH94]. Therefore much research is devoted to develop traffic information networks to increase safety, efficiency, the driver's comfort and to reduce pollution. A typical problem in such networks relates to storms of notification: An initial message causes a lot of subsequent messages which are irrelevant to fix the fault. Fault management of infrastructure networks can be solved by model-based diagnosis techniques [SNH94, Sch94].

The following problem is taken from [SNH94, Sch94]. The information system is described in terms of processes exchanging messages. The behaviour of a process can be described by its state transitions. Figure 3.3 shows a process that reads data from a buffer. In absence of errors the process runs through the following sequence of states: *remove begin, buffer exists, data exists, remove ok, remove end.* The states *no data* and *not remove ok* indicate faulty behaviour which is accompanied by error messages such as *no buffer* or *empty buffer.*



1. $revisable(\neg ok(\_,\_,\_))$.
2. $revisable(in(\_,\_,\_))$.
3. $proc\_state(trfc\_snd, remove, buffer\_exists)$.
4. $\bot \leftarrow proc\_state(P, remove, no\_data)$,
   $in(P, remove, no\_data)$,
   $proc\_state(P, remove, remove\_begin)$,
   $proc\_state(P, remove, buffer\_exists)$,
   $not\ in(P, remove, remove\_begin)$,
   $not\ in(P, remove, buffer\_exists)$.
5. $\bot \leftarrow in(P, remove, remove\_begin)$,
   $not\ \neg ok(P, remove, remove\_begin)$.
6. $\bot \leftarrow proc\_state(P, remove, remove\_begin)$,
   $signal(P, no\_buffer)$,
   $not\ in(P, remove, remove\_begin)$.

Figure 3.3. Remove automaton and part of its system description.

The process is described by four predicates:

1. $proc\_state(P,B,Q)$ states that automaton B belongs to process P and that the status Q is allowed for B.

2. $in(P,B,Q)$ indicates that P has actually reached status Q of automaton B after a malfunctioning is detected.

3. $ok(P,B,Q)$ means that P has actually reached status Q of B and Q belongs to the default path taken if no faults occured.

4. $signal(P,S)$ represents an observation: a signal with message S was sent by process P.

By default the *ok*-path is taken and the process has not yet reached the automaton's status after a malfunctioning, i.e. we assume $\neg ok$ and *in* (line 1/2) false. Line 3 is an example for a state of an automaton. The constraint in line 4 contains the possible transitions: If the process has reached the state *no data* then it must have previously been in *remove_begin* or *buffer_exists*. Line 5 relates *in* and *ok* such that a malfunctioning which lead to *in* contradicts the assumption that the process ran through the *ok*-path. Line 6 states that an observed signal such as *no buffer* contradicts the assumption not to be in state *remove_begin*.

$material(vestolen\_a\_3512)$.
$group(vestolen\_a\_3512, pe)$.
$medium(isoproponal\_koh)$.
$comp(isoproponal\_koh, isopropanol)$.
$comp(isoproponal\_koh, koh)$.
$group(isoproponal\_koh, alcohol)$.

$revisable(ab(\_))$.
$\bot \leftarrow group(X, weak\_base)$,
$\quad group(X, strong\_base)$.
$\bot \leftarrow resistent(X, Y, N1)$,
$\quad resistent(X, Y, N2), N1 \neq N2$.

Figure 3.4. Some database entries and integrity constraints of MEDEX.

If we add the system description and the observation $signal(trfc\_snd, remove, no\_buffer\_data)$ there is a contradiction. It can be resolved by either assuming

$$\neg ok(traffic\_snd, remove, buffer\_exists),$$
$$in(traffic\_snd, remove, buffer\_exists),$$
$$in(traffic\_snd, remove, no\_data)$$
or
$$\neg ok(traffic\_snd, remove, remove\_begin),$$
$$in(traffic\_snd, remove, remove\_begin)$$
$$in(traffic\_snd, remove, no\_data),$$

In both cases the process ends up in the abnormal state *no_data*.

3.2.2 *Integrity Checking for the Chemical Database MEDEX*

Assuring integrity of databases is a difficult problem. Especially, deductive databases need additional effort to be consistent since rules allow to infer new data indirectly. For example, the chemical database MEDEX is maintained by various individuals who update data and add inference rules without coordination [Ham94].

The database contains, for example, entries for Vestolen a 3512 and Isopropanol KOH (see Figure 3.4). Assume two experts add different rules r_1, r_2 and r_3, r_4 to the database (see Figure 3.5). The first knows that materials of the pe-group have a resistance factor 2 against strong bases (rule r_1) and that media containing KOH are strong bases (r_2). The second expert knows that materials of the pe-group have a resistance factor 5 against alcoholic media (r_3) and that Isopropanol is a weak base (r_4). Database and rules are contradictory wrt. the integrity constraints (see Figure 3.4) that a substance is either a weak or a strong base and that a material cannot have two different resistance factors for a medium. The contradiction can be removed by the diagnoses $\{ab(r_2)\}$, $\{ab(r_1), ab(r_4)\}$, or $\{ab(r_3), ab(r_4)\}$. So the experts have to discuss the validity of the respective rules and modify them to guarantee overall consistency.

$resistent(Material, Medium, 2) \leftarrow$
$\quad not\ ab(r_1),$
$\quad medium(Medium),$
$\quad material(Material),$
$\quad group(Medium, strong\_base),$
$\quad group(Material, pe).$
$group(Medium, strong\_base) \leftarrow$
$\quad not\ ab(r_2),$
$\quad medium(Medium),$
$\quad comp(Medium, koh).$

$resistent(Material, Medium, 5) \leftarrow$
$\quad not\ ab(r_3),$
$\quad medium(Medium),$
$\quad material(Material),$
$\quad comp(Medium, koh),$
$\quad group(Medium, alcohol),$
$\quad group(Material, pe).$
$group(Medium, weak\_base) \leftarrow$
$\quad not\ ab(r_4),$
$\quad medium(Medium),$
$\quad comp(Medium, isopropanol).$

Figure 3.5. Some rules of MEDEX.

3.2.3 Alarm-Correlation in Cellular Phone Networks

Though cellular phone networks already contain intelligent network elements diagnosing local faults, alarm bursts caused by network element failures cannot be handled properly by this technology [FNJW97]. To deal with alarm bursts alarm correlation systems are required to filter and condense the incoming alarms to meaningful high-level alarms and diagnoses. We review the application described in [FNJW97] and show how the problem is solved with extended logic programming.

> Mobile networks can be divided into three parts (see Figure 3.6): the mobile station (MS), the access network with the base station transceiver (BTS) consisting of antennas, radio transceivers, cross connect systems (CC) and microwave (ML) or cable links (CL) and the base station controller (BSC), and the switched network, which is connected to the access network by the BSC's. The BSC provides the radio resource management , which serves the control and selection of appropriate radio channels to interconnect the MS and the switched network. The switched network interconnects the MS to the communication partner, which might be another MS or an ISDN subscriber.[FNJW97]

The networks are configured in a star topology (see Figure 3.7) with exactly one path from a BTS to the BSC. Since such networks are highly dynamic an explicit model of the network is a necessary prerequisite for alarm correlation. Consider the path depicted in Figure 3.8. Its topology is modelled by facts of the components' types and connections (see Figure 3.9). The fact $conn(ml16, up, bsc, down)$ states, for example, that the up-stream port of microwave link $ml16$ is connected to the down-stream port of the BSC.

The network elements are intelligent and perform local diagnosis resulting in alarm messages which are sent to the BSC. The detailed messages such as e.g. *bts_omu_link_fail* or *bcch_missing* are divided into four classes:

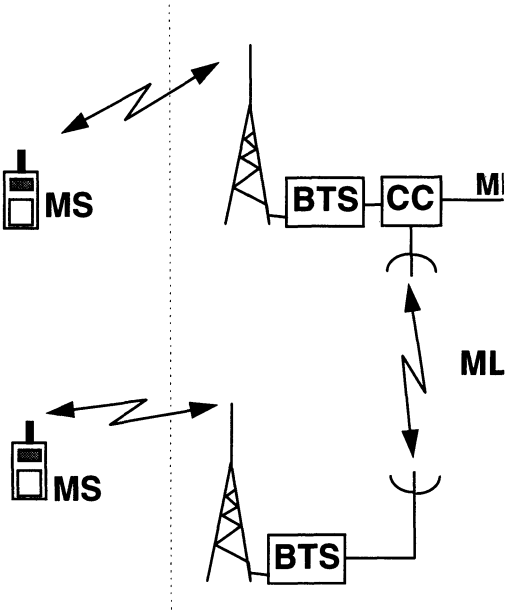


Figure 3.6. Alarm messages in the base station subsystem [FNJW97].

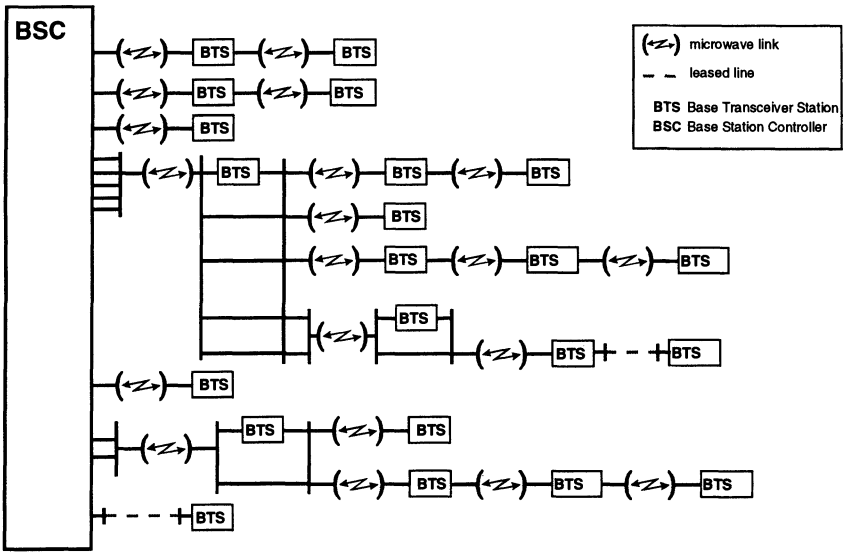


Figure 3.7. Star-configuration of a base station subsystem [FNJW97].

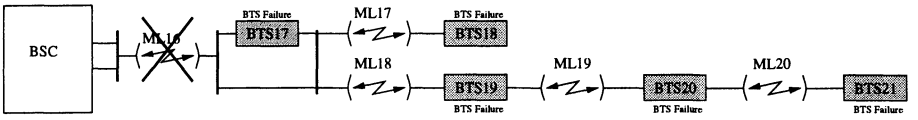


Figure 3.8. Network's topology where microwave link $ml16$ is faulty [FNJW97].

$type(ml16, ml).$	$conn(ml16, up, bsc, down).$
$type(ml17, ml).$	$conn(ml18, up, ml16, down).$
$type(ml18, ml).$	$conn(bts18, up, ml17, down).$
$type(ml19, ml).$	$conn(bts17, up, ml16, down).$
$type(ml20, ml).$	$conn(ml17, up, bts17, down).$
$type(bts17, bts).$	$conn(ml19, up, bts19, down).$
$type(bts18, bts).$	$conn(ml20, up, bts20, down).$
$type(bts19, bts).$	$conn(bts19, up, ml18, down).$
$type(bts20, bts).$	$conn(bts20, up, ml19, down).$
$type(bts21, bts).$	$conn(bts21, up, ml20, down).$

Figure 3.9. Rules for the network's topology.

```
class(bts_omu_link_fail, bts_failure_signal).
class(bcch_missing, bts_failure_signal).
class(available_traffic, bts_failure_signal).

  bts_failure_alarm(Sender) ←
    alarm(Sender, bts_omu_link_fail),
    type(Sender, bts).
  bts_failure_alarm(Sender) ←
    alarm(Sender, bcch_missing),
    type(Sender, bts).
```

```
class(lapd_link_failure, bts_failure_signal).
class(farend_alarm_1, farend_signal).
class(alive, status_signal).

  bts_failure_alarm(Sender) ←
    alarm(Sender, available_traffic),
    type(Sender, bts).
  bts_failure_alarm(Sender) ←
    alarm(Sender, lapd_link_failure),
    type(Sender, bts).
```

Figure 3.10. Alarm classes.

- Farend alarms are generated by a BTS if the components farther away from the BSC are not reachable anymore.
- BTS-failure alarms are generated directly in the BSC, when it detects that a BTS is not reachable anymore.
- The status signal *alive* is sent from each BTS on a periodical polling of the BSC. It is used to generate appropriate alarms in the BSC in case it gets lost on the way from the BTS to the BSC. The message is not physically present.
- There are other alarm messages which are not necessary to track down faults and therefore they are omitted in the model.

The alarms are classified by the facts *class* and *bts_failure_alarm* as shown in Figure 3.10. The generation and suppression of alarms is captured by the rules shown in Figure 3.11. The rules on the lefthandside state that each BTS sends an alive message and an alarm message in case of a farend alarm, respectively. The rules on the righthandside express that there cannot be an alive message at the BSC if there is a BTS-failure or a farend alarm. Here explicit negation ($\neg signal$) proves to be very useful to get a compact model. The model of [FNJW97] is more complicated due to missing explicit negation and abduction.

Farend and the status signal *alive* are propagated from up-stream to down-stream ports, through a BTS and through a microwave link (see Figure 3.12). While the BTS cannot fail, the microwave link may be faulty, though we assume by default that it is not abnormal ($not\ ab(NE)$).

Finally, we have to express that it is contradictory to have and not to have an alive message at the BSC and that there has to be either an alive message, or a BTS failure alarm, or the message was lost by the BTS (see Figure 3.13). The latter is not very probable, so that we additionally specify probabilities for the revisables. Abnormal microwave links are much more probable than lost messages.

$$signal(Sender, up, Sender, alive) \leftarrow type(Sender, bts).$$

$$signal(Sender, up, Sender, Signal) \leftarrow alarm(Sender, Signal), class(Signal, farend\_signal), type(Sender, bts).$$

$$\neg signal(bsc, down, Sender, alive) \leftarrow bts\_failure\_alarm(Sender), type(Sender, bts).$$

$$\neg signal(bsc, down, Sender, alive) \leftarrow alarm(Sender, Signal), class(Signal, farend\_signal), type(Sender, bts).$$

Figure 3.11. Signal generation and suppression.

$$signal(NE_2, down, Sender, Signal) \leftarrow type(Sender, bts), class(Signal, farend\_or\_status\_signal), conn(NE_1, up, NE_2, down), signal(NE_1, up, Sender, Signal).$$

$$signal(NE, up, Sender, Signal) \leftarrow class(Signal, farend\_or\_status\_signal), type(NE, bts), type(Sender, bts), NE \neq Sender, signal(NE, down, Sender, Signal).$$

$$signal(NE, up, Sender, Signal) \leftarrow not\ ab(NE), type(NE, ml), type(Sender, bts), class(Signal, farend\_or\_status\_signal), signal(NE, down, Sender, Signal).$$

Figure 3.12. Signal propagation.

$$\bot \leftarrow type(Sender, bts), not\ signal(bsc, down, Sender, alive), not\ bts\_failure\_alarm(Sender), not\ message\_lost(Sender).$$

$$\bot \leftarrow \neg signal(bsc, down, Sender, alive), signal(bsc, down, Sender, alive).$$

$$probability(ab(\_), 0.001).$$

$$probability(message\_lost(\_), 0.1).$$

Figure 3.13. Constraints for signals and a-priori probabilities of revisables.

alarm(*bts17*, *bcch_missing*).	*alarm*(*bts19*, *pcm_failure*).
alarm(*bts17*, *bcf_bie_alarm_in*).	*alarm*(*bts19*, *bts_omu_link_fail*).
alarm(*bts17*, *pcm_fail*).	*alarm*(*bts20*, *bcch_missing*).
alarm(*bts17*, *bts_omu_link_fail*).	*alarm*(*bts20*, *pcm_fail*).
alarm(*bts18*, *bcch_missing*).	*alarm*(*bts20*, *bts_omu_link_fail*).
alarm(*bts18*, *pcm_failure*).	*alarm*(*bts21*, *bcch_missing*).
alarm(*bts18*, *bts_omu_link_fail*).	*alarm*(*bts21*, *pcm_fail*).
alarm(*bts19*, *bcf_bie_alarm_in*).	*alarm*(*bts21*, *bts_omu_link_fail*).

Figure 3.14. Some alarms.

As long as there are no alarms the model is consistent. If alarms are generated and received by the BSC the constraints are violated and removing the contradictions yields the correct diagnoses of the problem. Consider for example the alarms in Figure 3.14. The burst of messages is difficult to survey for a human operator, though the most probable explanation by revision is fairly easy: microwave link *ml*16 is abnormal. For further test cases which are given in [FNJW97] see [DPS97a]. Concerning efficiency, both systems are comparable. This does not hold in general (see section 6.1.1), but for this particular application REVISE's negations allow a compact and very efficient representation of the problems.

3.2.4 Automatic Mirror Furnace

Have a look at the disc unit of the automated mirror furnace [TD95, The95] as discussed in the introduction. Figure 1.2 in the introduction shows the control unit, the motor, the plate, and the two switches. As the system description has a similar structure as digital circuits which we discussed in detail in section 3.1.3 we will focus on the behaviour of the components. Figure 3.15 shows *table* predicates for the components' behaviour and the computation of values for the motor. The plate and the switches are fairly easily modelled. In case the plate's second input parameter is not *enable* the plate stays in the same sample position P. If *enable* is set by the control unit then the new sample P' is computed by adding the old position P and the velocity V which is given in positions per time units. The result is taken modulo 23, the number of samples on the plate. The table for switch 4 checks whether position P is an integer which means that a sample is in position to be taken to the mirror focus. In case it is, switch 4 is active, otherwise it is off. Switch 5 works similarly: If position P is 0, then it is active or else off. The motor table is a bit more complicated. Besides the normal behaviour the motor can adopt fault modes defect and friction. In the former mode the motor stands still and therefore velocity is set to 0 and in the latter mode we assume a fixed velocity of 0.25 positions per time unit, in contrast to the 0.5 positions

```
motor_table(ok, start_pos, 0.5).
motor_table(ok, start_neg, −0.5).
motor_table(ok, stop, 0).
motor_table(defect, _, 0).
motor_table(fric'n, start_pos, 0.25).
motor_table(fric'n, start_neg, 0.25).
motor_table(friction, stop, 0).
comp_val(out(M, motor), W, T) ←
    state(M, motor, S),
    val(in(M, motor, 1), V, T),
    motor_table(S, V, W).
state(M, motor, ok) ←
    not fm(M, defect),
    not fm(M, friction).
state(M, motor, FM) ←
    fm(M, FM).

plate_table(ok, P, _, E, P) ←
    E ≠ enable.
plate_table(ok, P, V, enable, P') ←
    P' is P + V mod 23.

switch4_table(ok, P, active) ←
    integer(P).
switch4_table(ok, P, off) ←
    not integer(P).

switch5_table(ok, 0, active).
switch5_table(ok, P, off) ←
    P ≠ 0.
```

Figure 3.15. Behaviour of the disc drive units.

in the ok-mode. The table contains additionally a parameter for the control units commands, namely moving in positive direction, *start_pos*, moving in negative direction, *start_neg*, and not moving at all, *stop*. By implicit negation the motor is assumed to be in the ok-mode; if contradictions occur this may be changed to one of the fault modes *defect* or *friction*. Since the behaviour of these modes is modelled new predictions about velocity can be computed.

Now assume we observe that the control unit enables the plate and sends *start_pos* to the motor. Furthermore, the switches are observed to be active initially and then off (see Figure 1.3). As we know that the plate is initially in position 0, a contradiction occurs: Since the motor is supposed to move with 0.5 positions per time unit and the plate is enabled the position should be 1 after two time units. Subsequently, switch 4 should be active; but it is not. Assuming switch 4 abnormal removes the contradiction. Another solution is friction of the motor, because the fault model predicts position 0.5 after two time unit with friction which explains the observations. The defective motor or abnormal plate do not yield diagnoses. In both cases position 0 would be predicted after two time units. But this implies both switches abnormal which is not a minimal solution as switch 4 is already by itself an explanation.

3.2.5 Communication Protocol

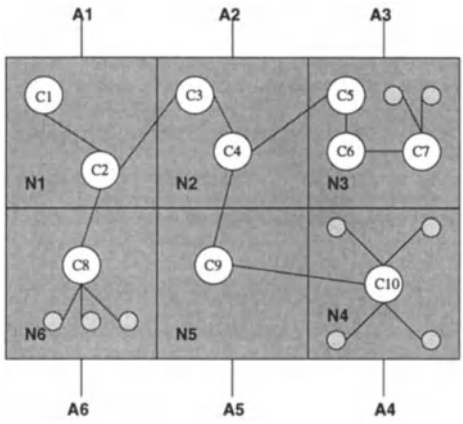


Figure 3.16. A communication network.

Centralised solutions for the diagnosis of spatially distributed systems such as computer and telecommunication networks are unnecessary complex and lead to a large communication overhead [FdAMNS97]. Consider a distributed system with n nodes, e.g. a computer network as depicted in Figure 3.16 consisting of n machines. When using a centralised diagnosis system the size of the system description (i.e. number of ground formulas) is linear in n. Diagnosis time will usually be worse than linear in n [MH93]. Also all observations have to be transmitted to the central diagnosis machine, causing a large communication overhead.

An agent–based approach decomposes a system into a set of subsystems. Each subsystem is diagnosed by an agent which has detailed knowledge over its subsystem and an abstract view of the neighbouring subsystems. Most failures can be diagnosed locally within one subsystem. This decreases diagnosis time dramatically in large systems. In the case of the computer network most machines in a subnet can usually fail without affecting machines in other subnets. Only those computers in other subnets can be affected which have sent messages to the faulty machine. Moreover, the local computation of diagnoses avoids the communication overhead which would be needed to forward all observations to the central diagnosis engine. Failures which affect more than one subsystem are diagnosed by the agents cooperating with each other. The cooperation process is triggered locally by an agent, when it realises that it cannot explain the observations by a failure in its own subsystem. The cooperation process is guided by a small amount of topological information. To demonstrate the power of our approach we formalise the domain of an unreliable protocol (like UDP) in a computer network and diagnose an example scenario.

$on\_route(n_1, c_3, n_2) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_1, n_1) \leftarrow i\_am(n_2).$
$on\_route(n_1, c_4, n_2) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_2, n_1) \leftarrow i\_am(n_2).$
$on\_route(n_1, c_5, n_2) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_5, n_3) \leftarrow i\_am(n_2).$
$on\_route(n_1, c_6, n_2) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_6, n_3) \leftarrow i\_am(n_2).$
$on\_route(n_1, c_7, n_2) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_7, n_3) \leftarrow i\_am(n_2).$
$on\_route(n_1, c_8, n_6) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_8, n_1) \leftarrow i\_am(n_2).$
$on\_route(n_1, c_9, n_2) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_9, n_5) \leftarrow i\_am(n_2).$
$on\_route(n_1, c_{10}, n_2) \leftarrow i\_am(n_1).$	$on\_route(n_2, c_{10}, n_5) \leftarrow i\_am(n_2)....$

Figure 3.17. Facts for the routing table.

$area\_component(n_1, c_1) \leftarrow i\_am(n_1).$	$area\_component(n_3, c_6) \leftarrow i\_am(n_3).$
$area\_component(n_1, c_2) \leftarrow i\_am(n_1).$	$area\_component(n_3, c_7) \leftarrow i\_am(n_3).$
$area\_component(n_2, c_3) \leftarrow i\_am(n_2).$	$area\_component(n_6, c_8) \leftarrow i\_am(n_6).$
$area\_component(n_2, c_4) \leftarrow i\_am(n_2).$	$area\_component(n_5, c_9) \leftarrow i\_am(n_5).$
$area\_component(n_3, c_5) \leftarrow i\_am(n_3).$	$area\_component(n_4, c_{10}) \leftarrow i\_am(n_4).$

Figure 3.18. Facts for the component hierarchy.

The routing tables comprise facts stating to which neighbour node a message addressed to a component has to be sent. The knowledge is local since each agent only knows its neighbours. In order to keep the facts in a single knowledge base which is the same for all agents the facts hold only for the respective agent (*i_am*). For example, for n_1 and n_2 we get the routing tables shown in Figure 3.17. Additionally, each agent knows its components (see Figure 3.18). Since this knowledge is local it is only derivable for the respective agent (*i_am*).

In the implementation we model only two modes, abnormality (*ab*) and being ok (*not ab*). The predicate *ab* is revisable. The default truth value of the predicate *ab* is false, which means that by default we assume components to work fine. Possible contradictions to these assumption are caused by violation of consistency of abstraction (see Figure 3.19) and existence of a cause for a lost message (see Figure 3.20). Consistency of abstraction means than an abnormal area contains at least one abnormal component. A contradiction arises if the area is detected to be abnormal but no faulty component is abduced. This constraint has only local character (*i_am*), since an agent cannot detect abnormal components of other areas (see Figure 3.19). The basic integrity constraint to start the diagnostic process states that it is contradictory to observe a lost message from node N to component C and not to have lost it on the route from N to C. The message is lost somewhere on this route if at least one of

$$\bot \leftarrow i\_am(N), ab(N), \; not\, has\_ab\_component(N). \quad \Big| \quad has\_ab\_component(N) \leftarrow area\_component(N,C), ab(C).$$

Figure 3.19. Rules for consistency of abstraction.

$$\bot \leftarrow message\_lost(N,C), \; not\, lost\_on\_route(N,C).$$
$$lost\_on\_route(N,C) \leftarrow ab(N).$$
$$lost\_on\_route(N,C) \leftarrow on\_route(N,C,M), ab(M). \quad \Big| \quad \bot \leftarrow message\_lost(N,C), on\_route(N,C,M), ab(M), \; not\, new\_message\_lost(M,C).$$

Figure 3.20. Rules for existence of a cause for a lost message and refined observation.

the involved nodes is abnormal (see Figure 3.20). The second constraint allows us to abduce new observations. If a message is lost from N to C and M is a neighbour of N which is assumed to be abnormal by N, then N abduces the new observation that the message was lost on the way from M to C (see Figure 3.20).

Now have a look at Figure 3.16 and assume that the agent assigned to n_1 wants to send a message to component c_7. A timeout mechanism informs n_1 which sent the message that there was no acknowledgment for the message. Subsequently, n_1's knowledge base is updated and the observation $message\_lost(n_1, c_7)$ is added. Now the constraint for existence of a cause for a lost message (Figure 3.20) is violated. To repair the violation *lost_on_route* has to be abduced. There are two candidates $ab(n_1)$ and $ab(n_2)$, i.e. n_1's neighbour is faulty. Since we assume that the agents have locally perfect knowledge, agent n_1 knows that $\neg ab(n_1)$ holds so that only the latter candidate is valid. Assuming $ab(n_2)$ also satisfies the constraints for consistency of abstraction (Figure 3.19). Due to the fact $i\_am(n_1)$ only the local components, c_1 and c_2, of the component hierarchy (Figure 3.18) are visible so that the constraints are trivially satisfied for $ab(n_2)$. Assuming $ab(n_2)$ the second constraint in Figure 3.20 leads to the abduction of the refined observation $new\_message\_lost(n_2, c_7)$ indicating that n_1 believes that the message was lost on the way from n_2 to c_7. So the final diagnosis for the observation $message\_lost(n_1, c_7)$ is $\{ab(n_2), new\_message\_lost(n_2, c_7)\}$.

3.3 REVISE - A SYSTEM FOR PROGRAM REVISION

With this variety of applications in mind we turn to the computation of revisions and diagnoses. Before we discuss two algorithms to compute revisions, one being based on bottom-up evaluation of extended logic programs, the other being based on top-down

evaluation, we briefly review the theory of diagnosis from first principles as initially proposed by Reiter [Rei87] and corrected in [GSW89].

3.3.1 Definitions

To compute revisions, we define conflicts which are sets of default assumptions that lead to a contradiction and show how to solve the conflicts by changing the assumptions so that all conflicts are covered. Such a cover is called hitting set, since all conflicts involved are hit.

Definition 3.8 *Conflict*
Let P be an extended logic program with default literals D. Then $C \subset D$ is a conflict *iff*

$$P \cup \{\neg c \mid not\ c \in C\} \models \bot$$

Definition 3.9 *Hitting Set*
A hitting set *for a collection of sets C is a set $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S \neq \{\}$ for each $S \in C$. A hitting set is minimal iff no proper subset of it is a hitting set for C.*

Conflicts and hitting sets allow to compute revisions:

Theorem 3.10 *Adopted from [Rei87]*
Let P be a program. Then R is a revision of P iff R is a minimal hitting set for the collection of conflicts for P.

To compute hitting sets Reiter proposed the algorithm shown in Figure 3.21 which was corrected in [GSW89]. The algorithm computes a hitting set tree which is defined as follows:

Definition 3.11 *Hitting Set Tree*
Let C be a collection of sets. An HS-tree for C, call it T, is a smallest edge-labeled and node-labeled tree with the following properties:

1. *The root is labeled $\surd$ if C is empty. Otherwise the root is labeled by an arbitrary set of C.*

2. *For each node n of T, let $H(n)$ be the set of edge labels on the path in T from the root node to n. The label for n is any set $\Sigma \in C$ such that $\Sigma \cap H(n) = \{\}$, if such a set Σ exists. Otherwise, the label for n is $\surd$. If n is labeled by the set Σ, then for each $\sigma \in \Sigma$, n has a successor, n_σ, joined to n by an edge labeled by σ.*

1. Let D represent a growing dag. Generate a node which will be the root of the dag. This node will be processed by step 2 below.

2. Process the nodes in D breadth first order. To process a node n:

   (a) Define $H(n)$ to be the set of edge labels on the path in D from the root down to node n.

   (b) If there is a node n' which is labeled by $\surd$ and $H(n') \subset H(n)$ then close node n and mark it $\times$.
   Otherwise,

   - if for all $x \in C$, $x \cap H(n) \neq \{\}$ then label n by $\surd$.
   - else let Σ be the first member of C for which $\Sigma \cap H(n) = \{\}$.
     - If there is a node n' which has been labeled by the set S' of C where $\Sigma \subset S'$, then relabel n' with Σ. For any α in $S' - \Sigma$, the α-edge under n' is no longer allowed. The node connected by this edge and all of its descendants are removed, except for those nodes with another ancestor which is not being removed.
       Interchange the sets S' and Σ in the collection.
     - Else label n by Σ.

   (c) If n is labeled by a set $\Sigma \in C$, then for each $\sigma \in \Sigma$,

   - either reuse a node, i.e. if there is a node n' in D such that $H(n') = H(n) \cup \{\sigma\}$, then let the σ-arc under n point to this existing node n'.
   - or generate a new downward arc labeled by σ. This arc leads to a new node m with $H(m) = H(n) \cup \sigma$. The new node m will be processed (labeled and expanded) after all nodes in the same generation as n have been processed.

3. Return the resulting dag, D.

Figure 3.21. Algorithm to compute minimal hitting sets [Rei87, GSW89].

3.3.2 A Bottom-Up Algorithm

The first algorithm to revise contradictory extended logic programs described in [DNP94] adopts Reiter's hitting set algorithm and refines it by taking into account preferences for revisables. For this refinement several pruning and caching methods have been implemented to avoid recomputations on different preference levels. To compute the conflicts sets the support sets for $\bot$ are generated. A support set indicates which default literals are involved in the derivation of a literal. To compute a support set we need a model of the program P which is denoted by $WFSX(P)$. The semantics $WFSX$ is explained later in section 3.3.3, for details see [PA92, PAA94, ADP94a, ADP94b, ADP95, AP96]. For the purpose of defining support sets it is sufficient that $WFSX(P)$ is a paraconsistent model of P, i.e. it may contain $\bot$, and that it is computed in square time wrt. the number of rules of the instantiated program.

Definition 3.12 *Support Set*
Let P be a program, $WFSX(P)$ its model and $L \in WFSX(P)$. A support set $SS(L)$ of L *is obtained as follows:*

1. *If L is positive or $\bot$ and there is a rule $L \leftarrow L_1, \ldots, L_n$ in P such that $L_i \in WFSX(P)$, then $SS(L) = \{L\} \cup SS(L_i)$*
2. *If L is a default literal not A then*
   (a) *If there are no rules for A then $SS(L) = \{not\ A\}$*
   (b) *If there is a rule $A \leftarrow L_1, \ldots, L_n$ in P and a L_i whose complement $\bar{L}_i \in WFSX(P)$ then $SS(L) = \{L\} \cup SS(\bar{L}_i)$*
   (c) *If $\neg A \in WFSX(P)$ then $SS(L) = SS(\neg A)$*

Example 3.13 *Let*

$$P = \{\bot \leftarrow not\ a,\ not\ b. \quad \bot \leftarrow not\ a,\ not\ c,\ d. \quad d.\}$$

then $SS_1(\bot) = \{not\ a, not\ b\}$ and $SS_2(\bot) = \{not\ a, not\ c, d\}$

To obtain a conflict from a support set for contradiction we just have to intersect the support set with the revisables and their negation to get rid of the objective literals and the default literals whose truth values cannot be changed.

Lemma 3.14 *Let P be a program with revisables R then $SS(\bot) \cap (R \cup not\ R)$ is a conflict.*

With the hitting set algorithm the revision of extended logic programs is straight forward as depicted in Figure 3.22. After computing a model, support sets, and conflicts, the hitting set algorithm can be applied. The algorithm works bottom-up, since a model of the program has to be computed by bottom-up evaluation in the first step.

1. Compute the model $WFSX(P)$ of program P
2. Compute all support sets $SS(\bot)$
3. Compute all conflicts from the support sets
4. Compute general hitting set tree
5. Complement all nodes marked $\surd$ and output them

Figure 3.22. Bottom-up algorithm to compute revisions [DNP94].

Example 3.15 *Let*

$$P = \{\bot \leftarrow not\ a, not\ b. \qquad \bot \leftarrow not\ a, not\ c,\ d. \qquad d.\}$$

P is contradictory and we expect the minimal revisions $\{a\}$ *and* $\{b,c\}$ *to remove the contradiction. The bottom-up algorithm returns these solution by first computing the model of P by bottom-up evaluation*

$$WFSX(P) = \{\bot,\ not\ a,\ not\ b,\ not\ c,\ d\}$$

There are two support sets for $\bot$

$$SS_1(\bot) = \{not\ a,\ not\ b\}\ and\ SS_2(\bot) = \{not\ a,\ not\ c,\ d\}$$

being reduced to the conflicts

$$C_1 = \{not\ a,\ not\ b\}\ and\ C_2 = \{not\ a,\ not\ c\}$$

The hitting set tree computed by the algorithm 3.21 is shown in Figure 3.23. It is constructed as follows:

In step 1 of 3.21 the directed-acyclic graph is initialised with root node labeled $\{not\ a,\ not\ b\}$. *With* $H(1) = \{\}$ *(2a) and* $\Sigma = \{not\ a,\ not\ b\}$ *(2b). In step 2c the successor nodes 2. and 3. are created. In the next iteration, node 2. is processed. With* $H(2) = \{not\ a\}$ *(2a) node 2. turns out as a solution (2b) and is marked* $\surd$. *For node 3.* $H(3) = \{not\ b\}$ *(2a) and in step 2b the node is labeled by* $\Sigma = \{not\ a,\ not\ c\}$. *In step 2c, the down arc to nodes 4. and 5. are created. For node 4. we get* $H(4) = \{not\ a,\ not\ b\}$ *and subsequently the node is closed (2b) since it is not minimal. Finally, node 5. is identified as solution and marked* $\surd$ *and with all nodes being processed the algorithm obtains the correct solutions from nodes 2. and 5.*

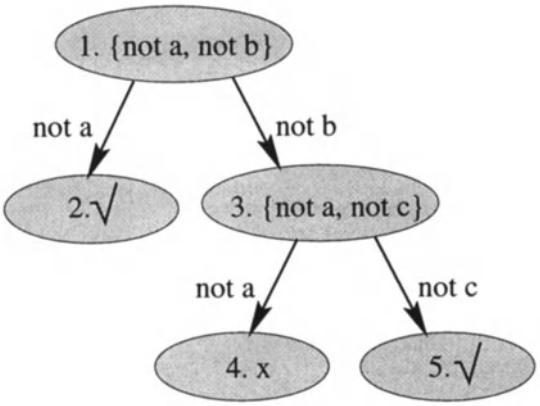


Figure 3.23. A hitting set tree.

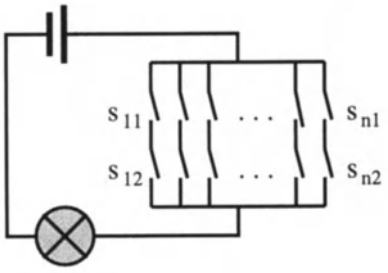


Figure 3.24. Power supply, switches s_{ij} and a bulb

The algorithm sketched in Figure 3.22 enhanced by pruning rules and a cache mechanism is implemented in the non-monotonic reasoning system REVISE 1.0 [DNP94]. The system shows nicely that extended logic programming is a suitable technique for model-based diagnosis. But in practice the algorithm faces a severe efficiency problem. Since the model is computed bottom-up it has to be done beforehand and in the second step all conflicts are computed. But in most cases this is not necessary. In fact, there are examples where 2^n conflicts are computed to come up with n diagnoses.

Example 3.16 *Bulb example*

Have a look at the circuit in Figure 3.24. A power supply is connected to a bulb via n parallel wires, each with two switches. Though all switches are supposed to be open,

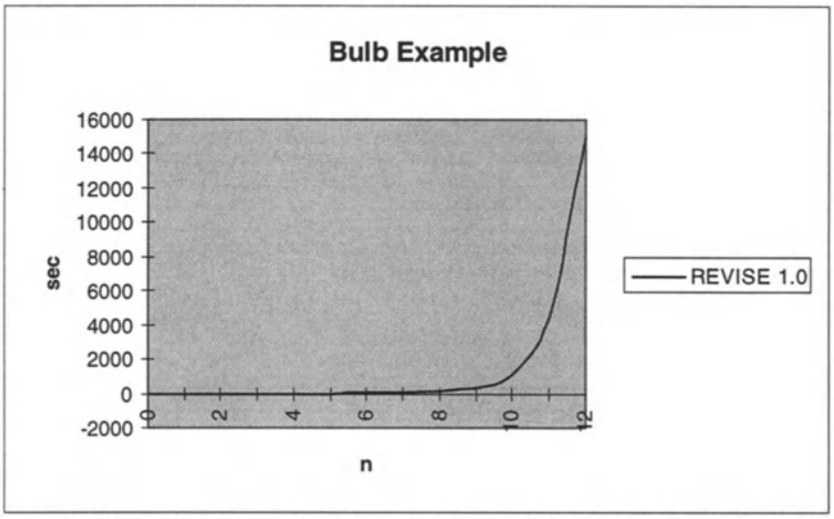


Figure 3.25. Timings of bottom-up algorithm for bulb example. A DEC5000/240 was used.

we observe that the bulb is burning. A conflict is for example that the n switches s_{i1} *are open. All in all, there are* 2^n *different conflicts, namely* $\{s_{ij} \mid 1 \leq i \leq n, j \in \{1,2\}\}$. *But all these conflicts lead finally to only n different diagnoses that the switches* s_{i1} *and* s_{i2} *are closed* $(1 \leq i \leq n)$.

We ran the algorithm of REVISE 1.0 for the bulb example and as expected it performs exponentially. While five wires in parallel are still solved in a reasonable time of 5 sec, ten wires already consume 19 minutes and twelve 4 hours and 10 minutes (see Figure 3.25). The bad performance of the algorithm is due to its bottom-up evaluation of the model. Rather than producing conflicts when necessary all of them have to be computed beforehand. With a top-down proof procedure the basic algorithm can be improved. Rather than doing one after the other, conflict generation and construction of the tree are merged. Before we refine the algorithm and describe a top-down revision algorithm we develop a top-down proof procedure for WFSX, the well-founded semantics with explicit negation.

3.3.3 Top-Down Proof Procedure

WFSX enjoys the properties of simplicity, cumulativity, rationality, relevance, and partial evaluation that other semantics do not fully enjoy [AP96, ADP94a, ADP94b]. Simplicity means that the semantics can be characterised by two iterative operators, without recourse to three-valued logic. Cumulativity means that the addition of lemmata, i.e. true propositions, to the program does not change the semantics. Rationality refers to the ability to add the negation of a non-provable conclusion without affecting the semantics. The issue of relevance is of particular importance for the top-down algorithm. Relevance means that the top-down inference of a literal requires nothing but the predicate call-graph below the literal. Partial evaluation means that partially evaluated programs do not change the semantics of the program.

The above properties are important as the following two examples show. Consider the programs

$$P_1 = \{p \leftarrow p\} \qquad P_2 = \{p \leftarrow not\ p\}$$

In classical logic we obtain the models $M_1 = \{\}$ and $M_2 = \{p\}$ for P_1 and P_2, respectively. Thus, as expected intuitively, we cannot conclude p from P_1, but we do from P_2. In Prolog, however, both cases would lead to no inference due to infinite loops. Trying to prove p in P_1 Prolog applies the rule $p \leftarrow p$ over and over again. A similar reason stops P_2 from terminating in Prolog. The problems occur in general due to positive or negative loops through recursion. To deal with this problem we define T- and TU-trees, to prove verity and non-falsity [ADP94a, ADP94b, ADP95, AP96].

Definition 3.17 *T-tree, TU-tree*
Let P be a ground extended logic program. A T-tree (resp. TU-tree) for a literal L is an and-tree with root labeled L and nodes labeled by literals. T-trees (resp. TU-trees) are constructed top-down starting from the root by successively expanding new nodes using the following rules:

1. *If n is a node labeled with an objective literal L then if there are no rules for L in P then n is a leaf else select a rule*

$$L \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$

from P. In a T-tree the successors of n are nodes labeled

$$L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$

while in a TU-tree there are, additionally, the successor nodes labeled

$$not\ \neg L_1, \ldots, not\ \neg L_m$$

2. *Nodes labeled with default literals are leaves.*

   In figures TU-trees are inside a box.

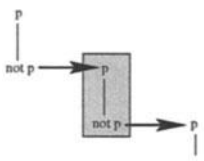
Definition 3.18 *Successful or failed tree*
A T- or TU-tree is either successful or it fails. All infinite trees are failed. A tree is successful/failed if its root is successful/failed. Nodes are marked as follows:

1. *A leaf labeled true is successful.*
2. *A leaf labeled with an objective literal distinct from true is failed.*
3. *A leaf labeled with a default literal not L is successful in T-tree (TU-tree) if*
   (a) all TU-trees (T-trees) for L are failed or
   (b) if there is a successful T-tree for $\neg L$.

   Otherwise it is labeled as failed.
4. *An intermediate node n of a T-tree (TU-tree) is successful if all its children are successful and otherwise failed.*

   All remaining nodes are labeled failed in T-trees and successful in TU-trees.

Example 3.19
Consider the program p ← not p. To prove verity of p a T-tree for p is created. In order to prove not p true, all TU-trees for p have to fail. Thus, an infinite loop occurs and therefore the nodes in the TU-tree are eventually classified as failed and the nodes in the T-tree as successful, so that p is proved true.



Theorem 3.20 *Correctness [ADP94a, ADP94b, AP96]*
Let P be a ground, possibly infinite, extended logic program, M its well-founded model according to WFSX, and let L be an arbitrary fixed literal.

- *If there is a successful T-tree with root L then $L \in M$ (soundness)*
- *If $L \in M$ then there is a successful T-tree with root L (completeness)*

The main issues in defining top-down procedures for well-founded semantics are infinite positive recursion, and infinite recursion through negation by default. The former results in failure to prove verity, while the latter results in failure to prove verity and falsity, i.e. the literal is undefined. Cyclic infinite positive recursion is detected locally in T-trees and TU-trees by checking if a literal L depends on itself. A list of local ancestors is maintained to implement this pruning rule. For cyclic

1. $demo(L) : -demo(L,t,[],[]).$
2. $demo(true,\_,\_,\_).$
3. $demo((L,Cont),M,LA,GA) : -$
   $demo(L,M,LA,GA),$
   $demo(Cont,M,LA,GA).$
4. $demo(not\ L,t,\_,GA) : -$
   $not\ demo(L,tu,[],GA).$
5. $demo(not\ L,tu,\_,GA) : -$
   $not\ demo(L,t,GA,GA).$
6. $demo(L,\_,\_,\_) : -$
   $revisable(L),!,assumed(L,\_).$
7. $demo(L,\_,Ans,\_) : -$
   $loop\_detect(L,Ans),!,fail.$
8. $demo(L,t,LA,GA) : -$
   $rule(L,Body),$
   $demo(Body,t,[L|LA],[L|GA]).$
9. $demo(L,tu,LA,GA) : -$
   $complement\_neg(L,NL),$
   $rule(L,Body),$
   $demo((Body,notNL),tu,$
   $[L|LA],[L|GA]).$

Figure 3.26. Top-down proof procedure for WFSX.

infinite negative recursion detection, a set of global ancestors is kept. A T-tree for a literal L that already appears in an ancestor T-tree is failed. A TU-tree for a literal L which already appears in an ancestor TU-tree is successful. The *demo* predicate in Figure 3.26 implements top-down inference using T-trees and TU-trees and pruning of cycles. The predicate *demo* has as parameters the goal to be proven, the mode t or tu, the local and the global ancestor list. The top goal (1) initialises the ancestor lists as empty. Item 2 states that a node marked *true* is successful. A conjunction is inferred if the conjuncts hold (3). Item 4 and 5 implement that *not* L is successful in a T-tree (TU-tree) if L fails for TU-trees (T-trees), where the ancestor lists are kept accordingly. A revisable is proven if it is assumed, either by default or after flipping its truth value (6). Loop checking is implemented by (7) and rule application by (8) and (9). For revision, the *demo* predicate is extended to contain a fifth parameter that returns the revisables involved in the proof.

The top-down proof procedure is of tremendous importance for the revision algorithm described in the previous section. Rather than computing the model and support sets before constructing the hitting-set tree we can merge these steps now. The top-down inference allows to construct one conflict at a time. Therefore we do not evaluate the whole program anymore, but start by computing a successful T-tree for $\bot$. If there is one, we extract a conflict of the tree and with the conflict we expand the hitting set tree.

3.3.4 A Top-Down Algorithm

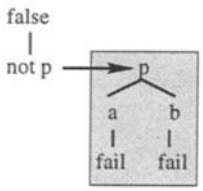
To compute revisions we can adopt the top-down proof procedure to generate conflicts. We know that there are conflicts if and only if $demo(\bot)$ succeeds. Furthermore, the

successful T-tree of $\perp$ contains the default literals forming the conflict. Thus we can employ *demo* to generate conflicts by collecting the encountered default literals in a successful proof for $\perp$.

Proposition 3.21 *demo and conflicts*
There is a conflict for P iff demo($\perp$) succeeds. If T is a successful T-tree for $\perp$, then the union of all default literals in T is a conflict.

Example 3.22 *Consider the extended logic program with revisables a, b, and c initially false:*

$$\perp \leftarrow not\ p. \quad \perp \leftarrow a, not\ c. \quad \perp \leftarrow c. \quad p \leftarrow a. \quad p \leftarrow b.$$



Have a look at the T-tree for $\perp$ on the left. The top-down proof procedure creates a T-tree with root $\perp$ and child not p. To prove verity of not p falsity of p must be proven. A TU-tree rooted with p is created. Since both children a and b fail, p fails and not p succeeds. Finally, $\perp$ is proved based on the falsity of the revisables a and b. Thus, the proof procedure returns the conflict {not a, not b}, which means that at least a or b must be true.

The above proposition allows to replace support sets for $\perp$ by iterative conflict generation. Rather than computing all conflicts beforehand, they are generated iteratively. When we expand a node in the hitting-set tree we do not take an arbitrary conflict, but one which is distinct from the partial solution $H(n)$ of the node found on the path from root to node. Therefore we add the partial solution $\{L \mid not\ L \in H(n)\}$ to the program and generate by *demo* a conflict. The following lemma ensures that for intermediate nodes n there is a conflict of P that contains $H(n)$.

Lemma 3.23
Let P be a program and n a node in a hitting-set tree such that n is not a leaf. Then there is a conflict C of P such that $H(n) \subset C$.

To put it in other words, we can revise the literals of $H(n)$ and still obtain another conflict. Thus we add $\{L \mid not\ L \in H(n)\}$ to P and generate a conflict to expand the current node. Based on proposition 3.21 and lemma 3.23 we can change the bottom-up algorithm in Figure 3.22 that uses the hitting-set algorithm in Figure 3.21 by modifying the latter to contain conflict generation as an integral part. We only have to work on the part of the algorithm where n is either identified as solution and marked $\surd$ or else a new conflict Σ is chosen:

- if for all $x \in C$, $x \cap H(n) \neq \{\}$ then label n by $\surd$.
- else let Σ be the first member of C for which $\Sigma \cap H(n) = \{\}$.

Instead we pick up an unmarked node with minimal $H(n)$. Choosing a minimal node is crucial to guarantee minimal solutions. By lemma 3.23 we can safely replace the check for solution by testing whether there are conflicts for $P \cup \{L \mid not\ L \in H(n)\}$ and in case there are taking one:

- let n be an unmarked node such that $H(n)$ is minimal; mark n
- If there is no conflict for $P \cup \{L \mid not\ L \in H(n)\}$ then label n by $\surd$.
- else let Σ be a conflict for $P \cup \{L \mid not\ L \in H(n)\}$.

The overall algorithm is depicted in Figure 3.27.

Proposition 3.24 *Correctness of Algorithm 3.27*
The top-down algorithm in Figure 3.27 terminates. $H(n)$ is a hitting-set iff n is a leaf marked $\surd$.

Corollary 3.25 *Computing Revisions*
The top-down algorithm in Figure 3.27 computes the revisions of P. If a leaf n is marked $\surd$, then $\{L \mid not\ L \in H(n)\}$ is a revision.

The corollary follows by Reiter's theorem 3.10 and the correctness of the top-down algorithm.

Since minimality is an abstract criterion the algorithm allows for a wide range such as minimality by cardinality, by set-inclusion, or by probability. All of them are implemented and have been used. Minimality by cardinality is essential for large problems such as the ISCAS85 benchmark circuits (see code in section 3.1.3 and Figure 3.30); minimality by set-inclusion is used in the examples of traffic control (section 3.2.1), integrity of databases (see section 3.2.2), the automated mirror furnace (see section 3.2.4), and communication protocols (see section 3.2.5); minimality by probability is used throughout the example of alarm correlation in cellular phone networks (see section 3.2.3).

Example 3.26 *Consider the extended logic program with revisables a, b, and c initially false and have a look at Figure 3.28.*

$$\bot \leftarrow not\ p. \quad \bot \leftarrow a, not\ c. \quad \bot \leftarrow c. \quad p \leftarrow a. \quad p \leftarrow b.$$

The algorithm starts with the empty graph and passes $H(n) = \{\}$ to the inference engine (1). As explained in example 3.22, the proof procedure returns the conflicts $\{not\ a, not\ b\}$ (2) which can be satisfied by adding a or b to the program. Thus, the downward arcs of the root node now labeled by the conflict $\{not\ a, not\ b\}$ are labeled by $\{not\ a\}$ and $\{not\ b\}$ (3). Assume that arc $\{not\ a\}$ is selected for expansion

1. Let D represent a growing dag. Generate a node which will be the root of the dag. This node will be processed by step 2 below.

2. While there is an unmarked node:

   (a) Define $H(n)$ to be the set of edge labels on the path in D from the root down to node n.

   (b) If there is a node n' which is labeled by $\surd$ and $H(n') \subset H(n)$ then close node n and mark it $\times$.
   Otherwise,

   - let n be an unmarked node such that $H(n)$ is minimal; mark n
   - If there is no conflict for $P \cup \{L \mid not\ L \in H(n)\}$ then label n by $\surd$.
   - else let Σ be a conflict for $P \cup \{L \mid not\ L \in H(n)\}$.
     - If there is a node n' which has been labeled by the set S' of C where $\Sigma \subset S'$, then relabel n' with Σ. For any α in $S' - \Sigma$, the α-edge under n' is no longer allowed. The node connected by this edge and all of its descendants are removed, except for those nodes with another ancestor which is not being removed.
       Interchange the sets S' and Σ in the collection.
     - Else label n by Σ.

   (c) If n is labeled by a set $\Sigma \in C$, then for each $\sigma \in \Sigma$,

   - either reuse a node, i.e. if there is a node n' in D such that $H(n') = H(n) \cup \{\sigma\}$, then let the σ-arc under n point to this existing node n'.
   - or generate a new downward arc labeled by σ. This arc leads to a new node m with $H(m) = H(n) \cup \sigma$. The new node m will be processed (labeled and expanded) after all nodes in the same generation as n have been processed.

3. Return the resulting dag, D.

Figure 3.27. Top-down algorithm to compute revisions.

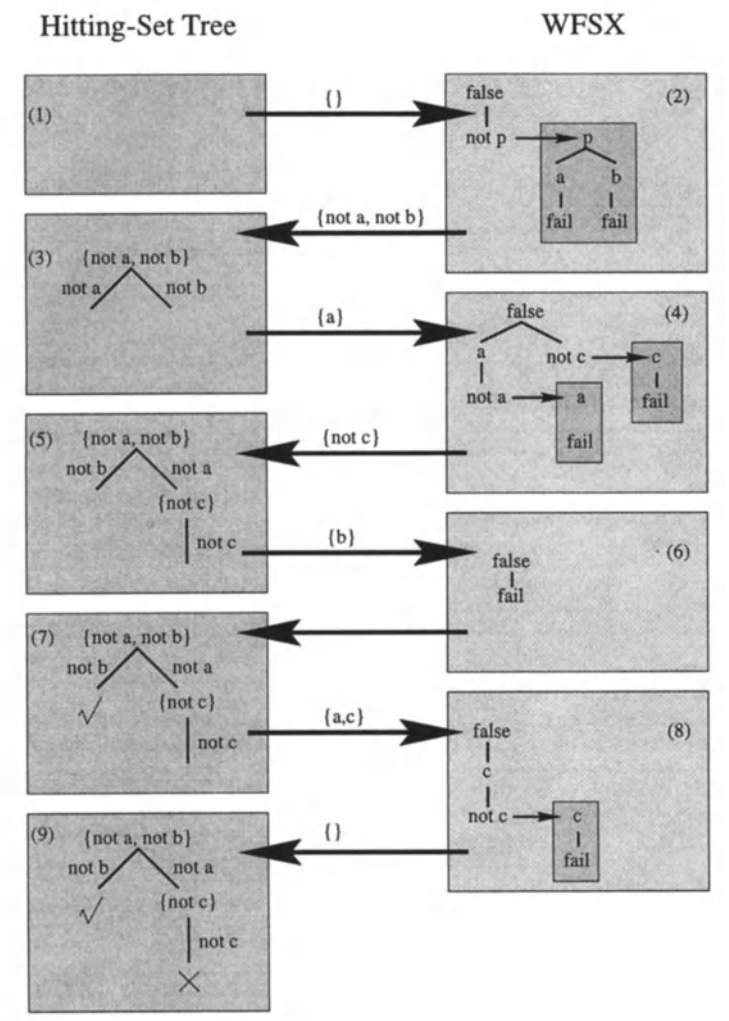


Figure 3.28. Iterative construction of hitting-set tree.

and $H(n) = \{a\}$ is passed to the inference engine. The new conflict $\{not\ c\}$ (4) is found. Thus the node is labeled $\{not\ c\}$ and a new downward arc not c is added (5). Next the node reached by the arc not b is selected for expansion and $\{b\}$ is passed to inference engine. It turns out to be a revision, since $\perp$ cannot be derived. The inference engine returns nothing and the node is marked $\surd$ (6). Note that returning nothing is distinct from returning the empty set. The former means there are no conflicts and we are done, the latter means that there is a conflict which cannot be resolved. Finally, for expansion of the last node, $\{a,c\}$ is send to the inference engine (7). The proof procedure returns the empty conflict $\{\}$ (8), so that the node is marked $\times$ as closed. Thus the overall solution is $\{b\}$ (9).

3.3.5 Comparison of the Algorithms

Three prototypes of REVISE have been implemented. REVISE 1.0 [DNP94] employs the bottom-up algorithm, while REVISE 2.3 and 2.4 [DNPS95, DPS96, SDP96, DPS97b] use the top-down algorithm. For REVISE 2.x the top-down evaluation allows to interleave the steps of computing conflict sets and hitting-sets. The hitting-set tree is computed incrementally, i.e. after one conflict is computed the candidate space is updated. This avoids the explosion of REVISE 1.0 in the bulb example . The computation of all conflicts in REVISE 1.0 is quite tedious as the timings in Figure 3.25 for the bulb example 3.24 show. The exponential runtime behaviour clearly shows that generating conflicts before constructing the hitting-set tree is quite crude and that many conflicts do not provide new information. For the bulb example 3.24 the algorithm of REVISE 1.0 computes first all 2^n conflicts and then reduces them to the n final diagnoses. REVISE 2.3 and 2.4 use the top-down algorithm and need only $n+1$ conflicts, which is the best case for this example. Based on the first conflict that s_{i1} are open, n partial solutions that s_{i1} are not closed are kept in the revision tree. Each of the next n conflicts leads to a diagnosis. For this example the minimality of the paths in the hitting-set tree are of particular importance, as it guarantees that the partial solutions leading to a diagnosis are processed first. The advantage of the incremental method of REVISE 2.3 and 2.4 over REVISE 1.0 is obvious from the timings in Figure 3.29. Both versions of the top-down algorithm perform in quadratic time in comparison to REVISE 1.0's exponential curve (see Figure 3.25). The curves are not linear due to the proof procedure which is quadratic in the number of instantiated clauses [AP96].

It may be puzzling that in the bulb example REVISE 2.3 performs by a constant factor better than REVISE 2.4 does. The difference between them is the representation of conflicts. The former uses difference lists and the latter sets as ordered lists. The difference lists allow a constant append, but may contain duplicates, while the ordered sets are free of duplicates, but a union costs $O(min\{m,n\})$, where m,n is the size of the sets. REVISE 2.3 performs better for the bulb example since in this special case conflicts do not contain any duplicates so that REVISE 2.4's extra check does not pay

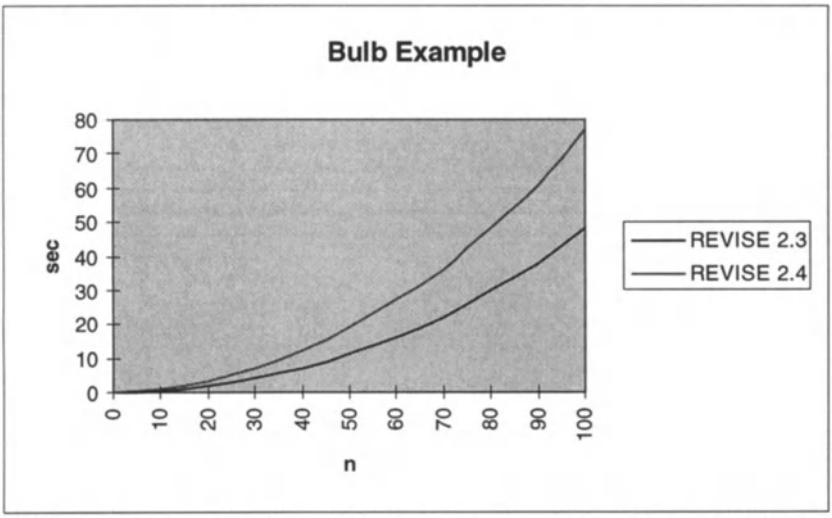


Figure 3.29. Bulb example. Timings in seconds. REVISE in interpreter mode. A DEC-5000/240 was used.

off. For real-world examples such as the ISCAS85 benchmark circuits [BPH85] the situation is different and avoiding recomputation turns out to be vital. REVISE 2.4 copes with this problem through its set representation and memoising in the WFSX interpreter. Depending on the fanout of the circuits' gates, version 2.4 is up to 80 times faster than 2.3 (see Figure 3.30).

3.4 SUMMARY

Extended logic programming is a suitable technique for model-based diagnosis. Integrity constraints and implicit negation are used to constrain predictions and observations and express default assumptions about the components states, respectively. Furthermore, the powerful language allows for a compact representation of diagnosis problems. We have modelled a wide variety of diagnosis problems including digital circuits, traffic control, integrity checking in databases, alarm correlation in cellular phone networks, automated mirror furnace, and communication protocols. The abductive capabilities and explicit negation proved to be especially useful for the problem of alarm correlation in section 3.2.3.

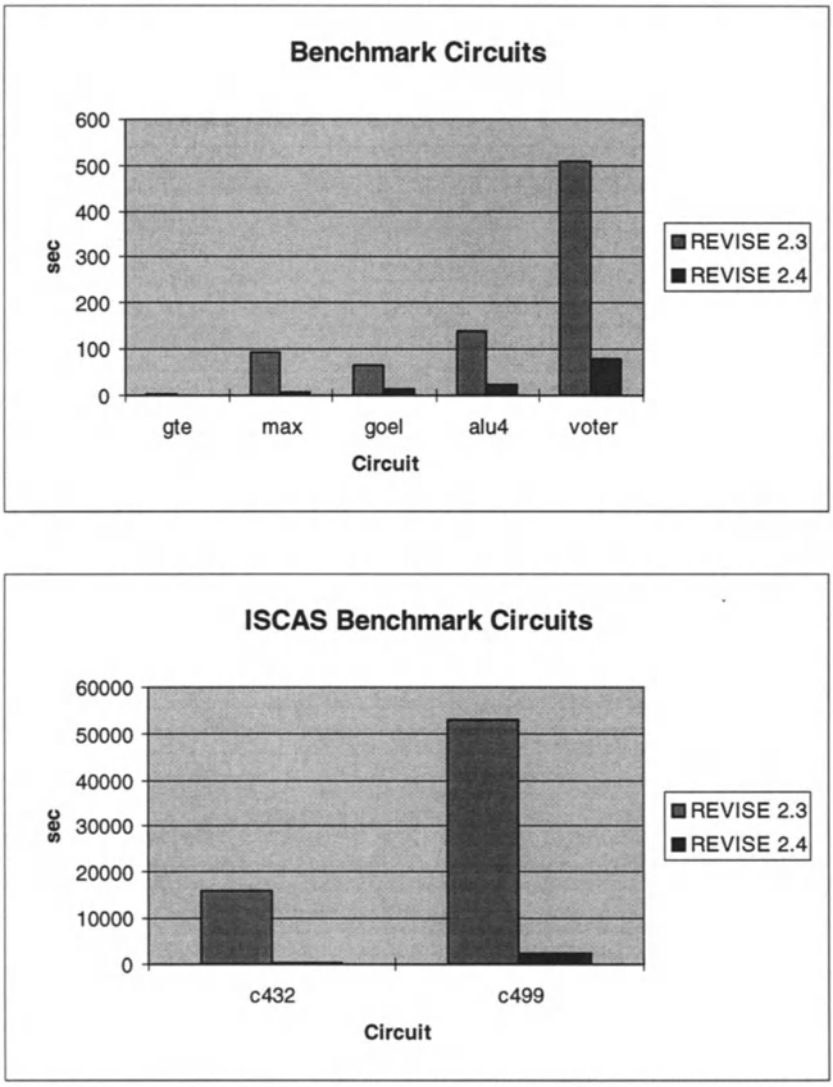


Figure 3.30. ISCAS85 benchmark circuits. Timings in seconds. REVISE computes all single faults in compiler mode. A DEC5000/240 was used.

To compute revisions two algorithms are developed, a bottom-up and a top-down algorithm. The initial bottom-up algorithm is implemented in the REVISE 1.0 system [DNP94] that proved the feasibility of diagnosis and extended logic programming. With the top-down proof procedure for WFSX [ADP94a, ADP94b, AP96] the initial bottom-up algorithm is refined and conflict generation and construction of the hitting-set tree are interleaved. The new top-down algorithm is implemented in REVISE 2.3 and 2.4 [SDP96, DPS97b]. It outperforms REVISE 1.0 by far and shows quadratic runtime behaviour where REVISE 1.0 performs exponentially. For small real-world problems it turns out that REVISE 2.4's conflict representation avoiding duplicates pays off. For the ISCAS85 benchmark circuits REVISE 2.4 runs up to 80 times faster than REVISE 2.3 does. An additional speed-up of ten to fifteen is gained in a compiled version of REVISE in comparison to an interpreted [DPS97b].

4 STRATEGIES IN DIAGNOSIS

The ability to select suitable diagnostic assumptions and models extends the power of model-based diagnosis for complex systems and can explicitly be modelled by diagnostic strategies. We discuss a framework which allows to express these strategies as formulas of a meta–language. We define syntax and semantics of the language and present a method for designing strategy knowledge bases as well as an efficient straightforward operational semantics for exploiting them.

4.1 INTRODUCTION

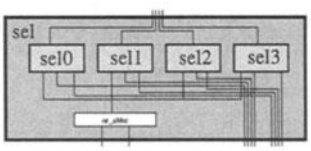
In the last years, model–based diagnosis has been extended by the introduction of the new concept of using different diagnostic assumptions (concerning number of faults, use of fault models and multiple models of the device) which can be activated during the diagnostic process. The selection of the appropriate diagnostic assumptions and system models during the diagnostic process is controlled by a set of predicates in the system description, for which Struss, Böttcher and Dressler [Str92b, BD93, BD94] introduced the term *Working Hypotheses*. Diagnostic strategies

M. Schroeder, *Autonomous, Model-Based Diagnosis Agents*

are rules defining which working hypotheses should be used in a given situation during the diagnostic process. Fröhlich, Nejdl, and Schroeder recently introduced a framework [NFS95, FNS96, FNS97] for controlling the diagnostic process by strategies expressed as sentences in a formal meta–language. Compared to previous approaches such as [BD94] this framework has the advantage of making strategic knowledge explicit and allowing the flexible specification of diagnostic strategies. The framework includes a declarative semantics for deciding whether a diagnostic process obeys the strategic knowledge specified by the strategies. We develop a design method and an operational semantics to efficiently handle strategic knowledge. Within our framework we identify the class of one–step rule–like strategies which is universal in the sense that it is expressive enough to describe every possible diagnostic process. Besides proving this theoretical result we demonstrate that all strategies needed in practice for our application domain can be denoted elegantly by one–step–strategies. Even preference concepts, which were up to now modelled separately [DS92, DNP94, FNS94] will be modelled as one-step strategies. We first show how to develop a strategic knowledge base for a specific application. Then we define an operational semantics which efficiently performs the diagnostic process using the strategies provided. In the course of the chapter we will discuss both deterministic and non–deterministic strategies for hierarchical circuit diagnosis. A major advantage of our method is that the strategies can be designed independently of each other, so that the strategic knowledge can be easily extended without the need to rewrite existing strategies.

Working Hypotheses. Let $\mathcal{L}$ be a first order language with equality. We consider a system and observations described by sets of formulas $SD \subseteq \mathcal{L}$ and $OBS \subseteq \mathcal{L}$. Struss introduced the concept of working hypotheses into model–based diagnosis in order to make diagnostic assumptions explicit [Str92b]. Working hypotheses are denoted by a set *WHYP* of atoms from the language $\mathcal{L}$. They can be used to represent multiple models of the system within one system description as shown in the following example.

Example 4.1 *Use of Working Hypotheses*



A 4-bit-selector is composed of four 1-bit-selectors and an or-gate. The working hypothesis refine(sel) can be used to switch between the abstract model and the detailed model of the selector in which its subcomponents sel_i and or_abbc are visible.

In SD the behaviour of the selector is modelled depending on the working hypothesis refine(sel): The rules of the abstract model contain ¬refine(sel) in their bodies, while the rules of the detailed model contain refine(sel).

To compute the diagnoses under a set of working hypotheses s, we add s and its negated complement $\neg\bar{s} := \{\neg wh | wh \notin s\}$ wrt. *WHYP* to the system description. We do not assume a particular diagnosis definition, but we use a generic function *diag*, which implements the computation of diagnoses. This definition allows for a wide range of diagnosis concepts like minimal diagnoses [Rei87], most probable diagnoses [dKW87], preferred diagnoses [DNP94] and others.

Definition 4.2 *Diagnosis under Working Hypotheses*
Let SD be a system description, OBS a set of observations and s a set of working hypotheses. Then $diag_s(SD \cup OBS) = diag(SD \cup OBS \cup s \cup \neg\bar{s})$ *denotes the* Set of Diagnoses under Working Hypotheses *s.*

Working hypotheses are an important concept for making the current diagnostic assumptions explicit. The selection of the appropriate working hypotheses is controlled by strategies.

4.2 A STRATEGY LANGUAGE

In this section we extend the previously defined strategy language [FNS94, NFS95, FNS96, FNS97] by generic deterministic and non-deterministic strategies, a careful treatment of the case where no diagnoses exist and a general condition for the termination of the process.

4.2.1 Motivation

We will introduce a bimodal language for the description of diagnostic processes. In contrast to previous approaches [Str92b, BD93, BD94] our modal language allows to completely formalise the diagnostic process: Processes are models of formulas in our language. By formalising diagnostic strategies we are able to judge their expressiveness (e.g. every process can be described, see theorem 4.23) and their limitations (e.g. we only look one step into the future). A language with two modalities is suitable for describing processes because two accessibility relations are inherent in a diagnostic process:

- Within each state of the process several hypotheses exist, which together with the system description and the observations induce logical models (worlds). We model this by an S5–operator ($\Box$) and a corresponding accessibility relation.
- A second modality ($\blacksquare$) models accessibility among the process states.

We define a formal semantics for this language. In section 4.4 we will then define an algorithm, which constructs models of our formulas (denoting strategies) with certain desirable properties. Thus, we will use the semantics for model checking and model generation.

4.2.2 *Syntax of the Language*

Diagnostic strategies control the diagnostic process by specifying which working hypotheses to use in a given situation. The state of the diagnostic process is represented by the current set of possible diagnoses. Thus the specification of a diagnostic strategy consists of

- properties characterising the current situation
- working hypotheses which are suitable for handling that situation

Example 4.3 *Using the technique described in example 4.1, we can represent multiple models of a system within a single system description. Suppose we have three system models M_0, M_1 and M_2, where M_0 is active by default. The working hypotheses force_M1 and force_M2 can be used to switch to M_1 and M_2, respectively. The following rule could be used to guide this selection:*

If all diagnoses satisfy a certain formula C which states that model M_0 is not appropriate, then the diagnostic process can continue either using model M_1 or model M_2, i.e. either the working hypothesis force_M1 or the hypothesis force_M2 can be activated.

To capture such strategies we define modalities to specify properties of the current diagnoses as well as to propose working hypotheses

Modalities $\Box$ and $\Diamond$. The preconditions of diagnostic strategies are statements about the current set of possible diagnoses. The atomic statements in these conditions are denoted by S5 modal operators:

$\Box p$: p is true under all diagnoses from $diag_s$.

$\Diamond p$: p is true under at least one diagnosis from $diag_s$.

Modalities $\blacksquare$ and $\blacklozenge$. Strategy formulas specify which working hypotheses should be assumed in a given situation. This is achieved by the modalities:

$\blacksquare\Box wh$: *wh* is a *necessary* working hypotheses in the current situation, i.e. in all successor states of the current diagnostic state we must assume *wh*.

$\blacklozenge\Box wh$: *wh* is a *possible* working hypotheses in the current situation, i.e. there is a successor state of the current diagnostic state in which *wh* holds.

In general, the syntax is defined as follows:

Definition 4.4 *$\mathcal{L}_{Strat}$–Formula*
Let $\mathcal{L}$ be a first-order language with equality, then $L \in \mathcal{L}$ is a $\mathcal{L}_{Strat}$–Formula. Let F, G be $\mathcal{L}_{Strat}$–Formulas and v, v_1, v_2 variables, then $\Box F$ and $\Diamond F$, $\blacklozenge F$ and $\blacksquare F$, $\neg F$ and $F \wedge G$, $v_1 = v_2$, $\forall v : F$ are $\mathcal{L}_{Strat}$–Formulas. Nothing else is an $\mathcal{L}_{Strat}$–Formula.

Note, that $\mathcal{L}_{Strat}$ has the same predicate symbols and constants as the system description language $\mathcal{L}$. The variables denote objects of $\mathcal{L}$. This will be ensured in the formal semantics presented in the next section. The model-theoretic semantics will be given for the full language. In practice we use a subset of the language $\mathcal{L}_{Strat}$, which can be efficiently handled by algorithms. It has turned out that rule-like strategies are best suited to express intuition: Based on properties of the current state of the diagnostic process the strategies propose new working hypotheses. Such strategies are restricted $\mathcal{L}_{Strat}$–formulas, which can be written as a rules $C \rightarrow H$, where C characterises the current diagnostic state and H the immediate successor states. The head H has one level of modalities $\blacklozenge$ and $\blacksquare$, the body C has none. Such *One-Step Strategies* are rules where the body has depth 0 and the head depth 1 wrt. to the bold modalities. Since we deal only with one-step strategies in this paper we simply call them strategies and explicitly refer to $\mathcal{L}_{Strat}$–formulas, when we need the full language in definitions.

Definition 4.5 *Depth*
If $L \in \mathcal{L}$, then $\Box L$ and $\Diamond L$ are formulas of depth 0. Let F, G be both formulas of depth n, then $\Box F$, $\Diamond F$, $\neg F$, $F \wedge G$, $F \vee G$ are formulas of depth n, $\blacksquare F$ and $\blacklozenge F$ are formulas of depth $n+1$.

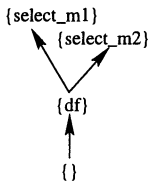
Definition 4.6 *One–Step Strategy*
A One–step Strategy *(in this paper simply called* Strategy*) is a formula of the form $C \rightarrow H$, where C is a formula of depth 0 and H is a formula of depth 1.*

4.2.3 Representation of a Diagnostic Process

The aim of strategies is to control the diagnostic process by proposing suitable working hypotheses. The diagnostic process itself can be non–deterministic, because more than one set of hypotheses can be proposed for a given situation. The *State* of the diagnostic process is characterised by the current set of working hypotheses. For representing diagnostic processes we use the notion of a *State Transition System*:

Definition 4.7 *State Transition System*
Let S be a finite set of states, $t \in S$ an initial state and $\rightarrow \subseteq S \times S$ a transition relation. Then $(S, \rightarrow, t)$ is called a State Transition System. *We assume $S \subseteq 2^{WHYP}$.*

Consider the following diagnostic process. We start the diagnosis with a simple model of a device (M_0) and the single–fault–assumption (see section 4.3.4). It turns out that no single–fault diagnosis exists, thus we allow the assumption of double faults. Again no diagnoses are found. Now we consider the simple model of the device as too abstract and we propose the activation of one of the more detailed models M_1 or M_2. In both of these models we would find single–fault diagnoses.



The transition system for representing this process has the states $\emptyset$ (initial state, no working hypotheses), *df* (then allow double faults), *force_M1* and *force_M2* (select one of the more detailed models).

4.2.4 *Designing Strategies*

The diagnostic process is represented by a transition system. We use strategies to influence the shape of this transition system. Let us first show how to define strategies in order to obtain a certain transition system.

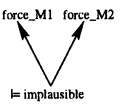
Deterministic Strategies. Often we want to assume one specific hypothesis in a given situation. For example, if we have found that a component C is definitely faulty, we activate the refined model for it to see which of its subcomponents caused the fault. We need a strategy which proposes a state transition leading to a state in which $refine(C)$ holds:

$$\Box ab(C) \rightarrow \blacklozenge\Box refine(C) \wedge \blacksquare\Box refine(C)$$

The formula describes exactly the transition system wrt. the hypothesis $refine(C)$. Other strategies specify transition systems wrt. other hypotheses. These partial transition systems are then combined by the algorithm which computes the diagnostic process. The $\blacklozenge$–operator is necessary in this formula. If we had only used the $\blacksquare$–operator in the conclusion of this formula, the formula would have been satisfied also if there were no successor of the current state. The quantification over all successor states would then be trivially satisfied.

Non–Deterministic Strategies. Sometimes there are several possibilities for continuing the diagnostic process in a given situation. Consider the generic model selection strategy introduced in example 4.3. Again we can first develop a transition system and then describe it by a strategy.

$$\begin{aligned}&\Box implausible \rightarrow \\ &\quad \blacklozenge\Box force\_M1 \wedge \blacklozenge\Box force\_M2 \\ &\quad \wedge \blacksquare(\Box force\_M1 \not\leftrightarrow \Box force\_M2)\end{aligned}$$



In this strategy *implausible* is a predicate in the system description, which indicates that there is no plausible diagnosis under the current system model.

Example 4.8 *implausible*
A diagnosis may be implausible if it contains triple faults including incomplete faults.

Triple faults holds if three distinct components X,Y,Z are abnormal and at least one of them is in an unknown fault mode. We denote the fault mode M of a component C by $fm(C,M)$.

$$\begin{aligned}\forall X,Y,Z: \quad & ab(X) \wedge ab(Y) \wedge ab(Z) \wedge \\ & X \neq Y \wedge X \neq Z \wedge Y \neq Z \wedge \\ & (fm(X,unknown) \vee fm(Y,unknown) \vee fm(Z,unknown)) \\ & \quad \rightarrow implausible\end{aligned}$$

The strategy for model choice proposes two possibilities for continuing the diagnostic process: Either assume M_1 or assume M_2 but do not assume both models at the same time. We will give a more specific account on model selection in section 4.3.1.

The above design method allows to define strategies independently without having to care about interference between different strategies. We will use it to define a complete set of strategies for circuit diagnosis (section 4.3) and describe how the independence can be preserved during the diagnostic process (section 4.4).

4.2.5 *Consistency of Transition Systems*

In order to check if the decisions made during the diagnostic process are consistent with the given diagnosis problem, we define how strategies can be interpreted as statements describing the diagnostic process. First we define logical structures which provide the interpretation for the strategies.

Definition 4.9 *$\mathcal{L}_{Strat}$–Model*
A model for $\mathcal{L}_{Strat}$ is a structure $M = \langle W,D,\rightarrow_1,\rightarrow_2,I\rangle$, where W is a set of individuals (called worlds), D is a domain of individuals, $\rightarrow_1$ and $\rightarrow_2$ are accessibility relations on the worlds, i.e. subsets of $W \times W$ and I is an interpretation function.

The function I provides the interpretation for predicates and ground terms (in our case all ground terms are constants). The values of the variables are given by an assignment:

Definition 4.10 *Assignment*
Given a domain D an Assignment *is a mapping from the set of variables into D. By $\alpha_{(x|d)}$ we denote the assignment that maps variable x to an element $d \in D$ and is defined in the same way as α on the other variables.*

The semantics of an $\mathcal{L}_{Strat}$–Model is defined as follows:

Definition 4.11 *Semantics of $\mathcal{L}_{Strat}$*
Let $M = (W,D,\rightarrow_1,\rightarrow_2,I)$ be an $\mathcal{L}_{Strat}$–model, F,G $\mathcal{L}_{Strat}$–Formulas, α an assignment and $w \in W$. Let

$$Val(t,\alpha,w) := \begin{cases} \alpha(t), & \text{iff } t \text{ is a variable} \\ I(w,t), & \text{iff } t \text{ is a constant} \end{cases}$$

Then

$M \models_{w,\alpha} P(t_1,\ldots t_n)$ *iff* $(Val(t_1,\alpha,w),\ldots Val(t_n,\alpha,w)) \in I(w,P)$

$M \models_{w,\alpha} t_1 = t_2$ *iff* $Val(t_1,\alpha,w) = Val(t_2,\alpha,w)$

$M \models_{w,\alpha} F \wedge G$ *iff* $M \models_{w,\alpha} F$ *and* $M \models_{w,\alpha} G$

$M \models_{w,\alpha} \neg F$ *iff* $M \not\models_{w,\alpha} F$

$M \models_{w,\alpha} \blacksquare F$ *iff f.a.* $w' \in W$ *s.th.* $w \rightarrow_1 w'$: $M \models_{w',\alpha} F$

$M \models_{w,\alpha} \blacklozenge F$ *iff ex.* $w' \in W$ *s.th.* $w \rightarrow_1 w'$ *and* $M \models_{w',\alpha} F$

$M \models_{w,\alpha} \Box F$ *iff f.a.* $w' \in W$ *s.th.* $w \rightarrow_2 w'$: $M \models_{w',\alpha} F$

$M \models_{w,\alpha} \Diamond F$ *iff ex.* $w' \in W$ *s.t.* $w \rightarrow_2 w'$ *and* $M \models_{w',\alpha} F$

$M \models_{w,\alpha} \forall x.F$ *iff f.a.* $d \in D : M \models_{w,\alpha_{(x|d)}} F$

We will use the following abbreviations:

$M \models_w F$ *iff* $M \models_{w,\alpha} F$ *for every assignment* α.

$M \models F$ *iff* $M \models_w F$ *for every* $w \in W$

The connection between the state transition relation and the $\mathcal{L}_{Strat}$–model is established in the following way: The possible diagnoses are interpreted as possible worlds, where all the diagnoses under one set of working hypotheses are connected wrt. the $\Box$–operator (relation $\rightarrow_2$). The accessibility relation for the $\blacksquare$–Operator (relation $\rightarrow_1$) is given by the state transition relation $\rightarrow$, i.e. diagnoses under different working hypotheses are connected by $\rightarrow_1$ iff the underlying sets of working hypotheses are connected by $\rightarrow$.

Definition 4.12 *Induced* $\mathcal{L}_{Strat}$*–Model* $M_{(S,\rightarrow,t)}$

Let $(S,\rightarrow,t)$ *be a state transition system. For* $s \in S$ *let* $\mathcal{D}_s = diag_s(SD \cup OBS)$ *be the diagnoses under s. We distinguish two cases:*

1. *If* $\mathcal{D}_s \neq \emptyset$*, then let* m_s *be the number of models obtained from the system description, the observations and the diagnoses in* $\mathcal{D}_s$ *and let* $\{M_{(s,1)},\ldots,M_{(s,m_s)}\}$ *be the corresponding set of Herbrand models.*

2. *If* $\mathcal{D}_s = \emptyset$*, then let* $m_s = 1$ *and* $M_{(s,1)}$ *be the Herbrand model defined by* $M_{(s,1)} = s \cup \{ab(SD)\}$

The Induced $\mathcal{L}_{Strat}$–Model $M_{(S,\rightarrow,t)}$ *is defined as*

$M_{(S,\rightarrow,t)} = \langle W', D', \rightarrow'_1, \rightarrow'_2, I' \rangle$, *where*

$W' = \{(s,i) \mid s \in S, i \in \{1,\ldots m_s\}\}$

D' *is the set of constants in* $\mathcal{L}$

$\rightarrow'_1 = \{((s,i),(s',j)) | s \rightarrow s', 1 \leq i \leq m_s, 1 \leq j \leq m_{s'}\}$

$\rightarrow'_2 = \{((s,i),(s,j)) \mid s \in S, i,j \in \{1\ldots m_s\}\}$

$I'((s,i),P) = \{\vec{x} \in \{D'^n | P(\vec{x}) \in M_{(s,i)}\}$ *for a predicate symbole P of arity n*

$I'((s,i),a) =$ *a, for a constant a in* $\mathcal{L}$.

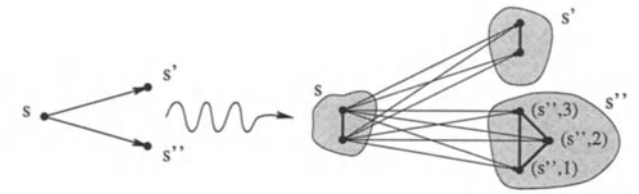


Figure 4.1. Example of the two accessibility relations $\rightarrow_1$ and $\rightarrow_2$.

Definition 4.13
Let F be an $\mathcal{L}_{Strat}$–Formula and $(S,\rightarrow,t)$ a transition system with induced model $M_{(S,\rightarrow,t)}$ and $s \in S$. We write $(S,\rightarrow,t) \models_s F$ iff $M_{(S,\rightarrow,t)} \models_{(s,i)} F$

The induced model needs some explanation. The worlds W' are a set of references to models which are either obtained from diagnoses of the system description and observations under a set of working hypothesis or given by $M_{(s,1)} = s \cup \{ab(SD)\}$ in case there are no diagnoses. There are two transition relations. The first one, $\rightarrow'_1$, is inherited from the given state transition system $(S,\rightarrow,t)$. For each transition $s \rightarrow s'$ from a set of working hypotheses s to another one s' there are transitions $(s,i) \rightarrow'_1 (s',j)$ where i and j are needed to identify the corresponding models $M_{(s,i)}$ and $M_{(s',j)}$. The second transition relation connects all models obtained under the same set of working hypotheses.

Example 4.14 *Induced Model*
The transition system on the left of Figure 4.1 induces the model on the right. For each set s of working hypotheses a set of models is obtained. The relation $\rightarrow'_2$ which is shown in bold arcs connects all models obtained under a set s of working hypotheses. The relation $\rightarrow$ on the left induces the transitions $\rightarrow'_1$ on the right.

To understand the declarative semantics, consider a given transition system $(S,\rightarrow,t)$. We check whether this transition system is a valid solution to the diagnostic problem given by $SD \cup OBS$ and a set of strategies $\mathcal{F}$. We call a transition system *consistent*, iff all its states satisfy all formulas from $\mathcal{F}$. Intuitively, the way the diagnostic process proceeds is consistent with what the strategies propose.

Definition 4.15 *Consistent*
A transition system $(S,\rightarrow,t)$ is consistent *with a set $\mathcal{F}$ of strategies, iff for all formulas $F \in \mathcal{F}$ and all $s \in S$ we have $(S,\rightarrow,t) \models_s F$.*

An important step to compute a consistent transition system is given by the next proposition. For the modalities $\Box$ and $\Diamond$ consider first a formula $L \in \mathcal{L}$ without modal

operators. A state s satisfies $\Diamond L$ if there is a diagnosis corresponding to state s under which L holds, and a state satisfies $\Box L$, if L holds under all diagnoses in state s. Additionally, we consider the case when there are no diagnoses, because the inconsistency of certain assumptions with the current situation should not lead to termination of the diagnostic process but rather to a change of assumptions. The absence of diagnoses under a given set of literals is indicated by the literal $ab(SD)$, intuitively indicating that the system description is not suited for the current set of assumptions.

For the modalities $\blacksquare$ and $\blacklozenge$ let F be a strategy formula. The semantics for the operators $\blacksquare F$ and $\blacklozenge F$ is given by the transitions. A state s satisfies $\blacksquare F$, if the formula F holds in all successor states. Similarly, a state satisfies $\blacklozenge F$, if the formula F holds in at least one successor state.

Proposition 4.16 *Let $(S,\rightarrow,t)$ be a transition system, $s \in S$, and $\mathcal{D}_s$ be the diagnoses under s. Let $L \in \mathcal{L}$ be a first order formula and F a strategy, then*

$(S,\rightarrow,t) \models_s \Diamond L$, *iff* $\mathcal{D}_s \neq \emptyset$ *and there is a diagnosis* $D \in \mathcal{D}_s$ *such that the model for* $SD \cup OBS \cup s \cup \neg\bar{s} \cup D$ *entails* L

or $\mathcal{D}_s = \emptyset$ *and the model for* $s \cup \neg\bar{s} \cup \{ab(SD)\}$ *entails* L

$(S,\rightarrow,t) \models_s \Box L$, *iff* $\mathcal{D}_s \neq \emptyset$ *and for all* $D \in \mathcal{D}_s$: $SD \cup OBS \cup s \cup \neg\bar{s} \cup D$ *entails* L

or $\mathcal{D}_s = \emptyset$ *and* $s \cup \neg\bar{s} \cup \{ab(SD)\}$ *entails* L.

$(S,\rightarrow,t) \models_s \blacklozenge F$, *iff* *ex.* $s' \in S$ *such that* $s \rightarrow s'$ *and* $(S,\rightarrow,t) \models_{s'} F$.

$(S,\rightarrow,t) \models_s \blacksquare F$, *iff* *for all* $s' \in S$: *from* $s \rightarrow s'$ *follows* $(S,\rightarrow,t) \models_{s'} F$.

Before we discuss the computation of a consistent transition system in section 4.4 we investigate the relation of the two pairs of modalities.

4.2.6 Relation between the Modalities

Since the modalities $\Box$ and $\Diamond$ give rise to an S5–structure the axioms T, 5, K, Df, Pl hold [Che80]:

T.	$\Box F \rightarrow F$
5.	$\Diamond F \rightarrow \Box\Diamond F$
K.	$\Box(F \rightarrow G) \rightarrow (\Box F \rightarrow \Box G)$
Df$\Diamond$	$\Diamond F \leftrightarrow \neg\Box\neg F$
PL.	F, where F is a tautology

For bold modalities only K and D hold:

K.	$\blacksquare(F \rightarrow G) \rightarrow (\blacksquare F \rightarrow \blacksquare G)$
D.	$\blacksquare F \rightarrow \blacklozenge F$

hold. The D axiom manifests itself also in the property that the models of a formula are serial, i.e. every state in a model has a successor. Intuitively, this means that the

diagnosis process either continues with a new set of working hypotheses or that a stable state is reached where the process loops. Apart from K and D, the bold modalities do not satisfy further axioms such as 4 ($\blacksquare F \rightarrow \blacksquare\blacksquare F$) and 5 ($\blacklozenge F \rightarrow \blacksquare\blacklozenge F$). The reason is that they are used to specify models and therefore we do not want to impose constraints on the possible transitions. However, an important relation between plain and bold modalities holds. Since the plain modal logic is embedded in the bold modal logic, the S5 modalities outside of one of the modalities $\blacksquare$ or $\blacklozenge$ can be dropped. If a world in state s is connected to a world in state s', then due to the S5 axioms, each other world in s is also connected to each other world in s'. The following table shows axioms on the relationship between the operators:

$$\begin{array}{lcllcl} \Box\Box F & = & \Box F & \Box\blacksquare F & = & \blacksquare F \\ \Diamond\Box F & = & \Box F & \Diamond\blacksquare F & = & \blacksquare F \\ \Box\Diamond F & = & \Diamond F & \Box\blacklozenge F & = & \blacklozenge F \\ \Diamond\Diamond F & = & \Diamond F & \Diamond\blacklozenge F & = & \blacklozenge F \end{array}$$

4.2.7 *Results of the Diagnostic Process*

The aim of the diagnostic process could be to identify one unique diagnosis. In general this would be too a restrictive criterion for terminating the diagnostic process because we might not have enough knowledge to discriminate among all the diagnoses. So we define that the diagnostic process terminates in a state where we assumed exactly all possible and necessary hypotheses.

This corresponds to a loop in the transition system as depicted on the right. If such a state yields diagnoses we cannot reach a more preferred state by applying another strategy.

Definition 4.17 *Stable State*
Let s be a state in the consistent state transition system $(S, \rightarrow, t)$ and $\mathcal{F}$ a set of strategy formulas. The state s is stable *wrt. $(S, \rightarrow, t)$, iff*

1. $diag_s(SD, OBS) \neq \emptyset$
2. $s = \{wh \mid (S, \rightarrow, t) \models_s \blacklozenge\Box wh \wedge \blacksquare\Box wh\}$

It is called weakly stable, *if* $s \subseteq \{wh \mid (S, \rightarrow, t) \models_s \blacklozenge\Box wh \wedge \blacksquare\Box wh\}$

The first condition states that $SD \cup OBS \cup s$ is consistent and the second condition is a fixpoint condition: s is already the set of all possible and necessary working hypotheses. The result of the diagnostic process is given by the diagnoses corresponding to the stable states and weakly stable states, respectively.

4.3 A STRATEGY KNOWLEDGE BASE FOR CIRCUIT DIAGNOSIS

In this section we apply the strategy language to the diagnosis of digital circuits. We model strategies such as the choice among multiple models, structural refinement, measurements and preferences. In section 4.4.4 use them to diagnose the voter circuit from [Isc85].

4.3.1 Multiple Views

Multiple views allow to describe the diagnosed systems emphasising different aspects. For circuit diagnosis it is often important to consider a physical view beside a functional one, because the physical view additionally takes the layout into account [Dav84]. We want to employ the functional model by default and the physical model only if we do not obtain good diagnoses.

Strategy (1) tells us how to choose between the models using the hypotheses *force_physical* and *force_functional*. The predicate *implausible*, which in our example holds if no single or double fault diagnosis exists, indicates that the other view should be activated. To avoid more than one activation it is also checked that *force_functional* does not yet hold. Once the body of the strategy is satisfied we have to make sure that the diagnostic process continues in two directions with the functional and the physical model, respectively, as active model. Thus, we adopt either *force_functional* or *force_physical*. Once this model selection has taken place both hypotheses are kept (monotonic addition of working hypotheses) (2,3).

$$\Box \mathit{functional} \wedge \Box \mathit{implausible} \wedge \Box \neg \mathit{force\_functional} \rightarrow \blacklozenge\Box \mathit{force\_functional} \wedge \blacklozenge\Box \mathit{force\_physical} \wedge \blacksquare\Box(\mathit{force\_functional} \leftrightarrow \neg \mathit{force\_physical}) \quad (1)$$

$$\Box \mathit{force\_physical} \rightarrow \blacklozenge\Box \mathit{force\_physical} \wedge \blacksquare\Box \mathit{force\_physical} \quad (2)$$

$$\Box \mathit{force\_functional} \rightarrow \blacklozenge\Box \mathit{force\_functional} \wedge \blacksquare\Box \mathit{force\_functional} \quad (3)$$

In the system description we model the connection between the working hypotheses *force_physical* and *force_functional* and the literals *physical* and *functional*, which select the appropriate system model. The functional model is used by default when no hypothesis is active:

$$\neg \mathit{force\_functional} \wedge \neg \mathit{force\_physical} \rightarrow \mathit{functional}$$
$$\mathit{force\_functional} \rightarrow \mathit{functional},\ \mathit{force\_physical} \rightarrow \mathit{physical}$$

4.3.2 Structural Refinement

Many authors address the use of hierarchies to reduce the complexity of diagnosis problems [Dav84, Ham91, Gen84, Moz91, BD94]. In particular, Böttcher and Dressler introduce the strategy of structural refinement which states that an abstract model of a component is refined only if it is uniquely identified as defective [BD94]. Only if all diagnoses contain a component C, it is possible and necessary to activate a detailed model of C:

$$\forall C.(\Box ab(C) \wedge refineable(C)) \vee \Box refine(C) \rightarrow \blacklozenge \Box refine(C) \wedge \blacksquare \Box refine(C) \quad (4)$$

In the system description the rules describing the abstract model are active when $refine(C)$ is false and the rules describing the detailed model are activated if $refine(C)$ is true. This variant of using hierarchies is very efficient since the refinement of the model is postponed until the erroneous components are identified.

4.3.3 *Flexible Evaluation of Hierarchies*

Next we extend the strategy of structural refinement for the case where we cannot uniquely identify a faulty component but want to continue the process on a more detailed level. By default we use structural refinement. If it is not possible to identify the faulty components on the abstract level of the system description, we refine the abstract components in every diagnosis where they occur. We denote the fact that the detailed model for component C is activated via structural refinement by the hypotheses $strong\_refine(C)$. The additional hypotheses $weak\_refine(C)$ is used for denoting that C is to be refined in those diagnoses, where it occurs. These two hypotheses dictate the choice of model in the system description as follows:

$$\begin{aligned} \forall C.(ab(C) \wedge weak\_refine(C) &\rightarrow refine(C)) \\ \forall C.(strong\_refine(C) &\rightarrow refine(C)) \end{aligned}$$

The hypotheses $weak\_refine(C)$ should be activated, if the component C is abnormal in some diagnosis but not in all diagnoses and all structural refinement steps are already done or $weak\_refine(C)$ has already been activated before.

$$\forall C_1.(\Diamond ab(C_1) \wedge \forall C_2.(\Box ab(C_2) \rightarrow \Box strong\_refine(C_2)) \vee \Box weak\_refine(C_1) \rightarrow \blacklozenge \Box weak\_refine(C_1)) \wedge \blacksquare \Box weak\_refine(C_1) \quad (5)$$

4.3.4 *Preference Relations among Diagnoses*

Preferences state that diagnoses with certain properties are better than others with other properties [DS92, DNP94, FNS94]. Frequently used preferences are, for example, the single fault assumption or "physical negation" [SD89], i.e. the assumption that the known fault models of the components are complete. To use preferences efficiently, the preferred property is activated by default and is relaxed only if there are no diagnoses which have the intended property. We can use $ab(SD)$ to detect if there are any diagnoses. $ab(SD)$ holds iff $diag_s(SD \cup OBS) = \emptyset$ (see sec. 4.2.5).

The preference relation on the left states that by default we are only interested in single faults (*sf*). If there are no diagnoses under the single–fault assumption we allow either diagnoses with double faults (*df*) or incompleteness of the fault models (*fm_inc*). If there are still no diagnoses under one of these relaxed hypotheses, we allow double fault diagnoses and incompleteness of fault models at the same time.

Example 4.18 *sf and df*
A working hypothesis nf (for n-faults) restricts the cardinality of diagnoses. In the system description it is used as a precondition in a formula which restricts the number of faults (ab–literals). For single and double faults we have, for example

$$\begin{aligned} \forall C_1, C_2: \quad & sf \wedge ab(C_1) \wedge ab(C_2) \rightarrow C_1 = C_2 \\ \forall C_1, C_2, C_3: \quad & df \wedge ab(C_1) \wedge ab(C_2) \wedge ab(C_3) \rightarrow \\ & C_1 = C_2 \vee C_1 = C_3 \vee C_2 = C3 \end{aligned}$$

Proposition 4.19 *n–faults*
Single- and double–fault assumption can be generalised to n–faults.

$$\forall C_1, \ldots, C_{n+1} : nf \wedge \bigwedge_{i=1}^{n+1} ab(C_i) \rightarrow \bigvee_{i,j=1, i \neq j}^{n+1} C_i = C_j$$

If SD contains the above formula and $nf \in s$ for some n, then for any diagnosis $D \in diag_s(SD \cup OBS)$ there are no more than n distinct components C_i s.th. $ab(C_i) \in D$.

The system description captures the default assumption of single faults by the rule $\neg df \wedge \neg tf \rightarrow sf$. The strategy of relaxing the single-fault property (6) checks if no diagnoses exist (using $ab(SD)$) and if neither *df* nor *fm_inc* hold. In this case it is possible to adopt either *fm_inc* or *df*, but not both at the same time. Finally, double faults together with the assumption of incomplete fault models are allowed only if there are no double fault diagnoses and no single–fault diagnoses with incomplete fault modes (7).

$$\Box ab(SD) \wedge \Box \neg df \wedge \Box \neg fm\_inc \rightarrow \blacklozenge \Box fm\_inc \wedge \blacklozenge \Box df \wedge \blacksquare(\Box df \not\leftrightarrow fm\_inc) \quad (6)$$

$$\Box ab(SD) \wedge (\Box df \vee \Box fm\_inc) \rightarrow \blacklozenge \Box(df \wedge fm\_inc) \wedge \blacksquare \Box(df \wedge fm\_inc) \quad (7)$$

These kind of strategies show the desired behaviour discussed and implemented in [FNS94]. First, the diagnosis system tries to find diagnoses under the most preferred set of properties (in our case diagnoses with only single faults). Only if this is not possible (i.e. $ab(SD)$) is true), these properties are exchanged with the next most preferred set of properties (in our case either double faults or the assumption of "fault mode incomplete") and so on. Wrt. to other strategies the preference properties are not monotonic: Whenever an unrelated diagnosis strategy is executed (e.g.

refinement, multiple views, etc.), the next state starts again by trying to find diagnoses corresponding to the most preferred set of properties.

4.3.5 *Measurements*

De Kleer, Raiman, and Shirley view diagnosis as an incremental task involving the three phases of generating explanations, choosing actions differentiating among them and performing these actions [dKRS91]. Our framework allows to incorporate these phases into the diagnostic process. Strategy (8) proposes a point X of the circuit for measurement if there are two diagnoses predicting different values for X. As measurements are expensive, we want to apply the strategy only if all refinements are already done which is checked in the first line.

$$\forall C.(\Box \mathit{refineable}(C) \wedge \Diamond \mathit{ab}(C) \rightarrow \Box \mathit{refine}(C)) \wedge$$
$$\forall X.\Box \mathit{point}(X) \wedge \Diamond \mathit{val}(X,0) \wedge \Diamond \mathit{val}(X,1) \rightarrow$$
$$\blacklozenge \Box \mathit{propose}(X) \wedge \blacksquare \Box \mathit{propose}(X) \tag{8}$$

The second phase of choosing the right action is carried out by procedural attachment in the system description. The (procedural) predicate $\mathit{best\_meas}(X)$ is true for a measurement point X, which is optimal according to some specification (for example minimum entropy). It needs only be evaluated for the measurement points X proposed by strategy (9).

$$\forall X.\Box \mathit{propose}(X) \wedge \Box \mathit{best\_meas}(X) \rightarrow$$
$$\blacklozenge \Box \mathit{measure}(X) \wedge \blacksquare \Box \mathit{measure}(X) \tag{9}$$

In the system description the hypothesis $\mathit{measure}(X)$ causes the specific measurement of X to be executed, which is also done by procedural attachment. The modification of the system description due to the insertion of the measured value has to be reflected by a change of the diagnostic state. This is achieved by using the hypothesis $\mathit{measure}(X)$ in a monotonic way (strategy (10)). Another (weak) monotonicity axiom is used for *propose*, which is active until the next consistent state is reached, in which the measurement is carried out, i.e. as long as $\mathit{ab}(SD)$ holds (strategy (11)).

$$\forall X.\Box \mathit{measure}(X) \rightarrow \blacklozenge \Box \mathit{measure}(X) \wedge \blacksquare \Box \mathit{measure}(X) \tag{10}$$
$$\forall X.\Box \mathit{ab}(SD) \wedge \Box \mathit{propose}(X) \rightarrow$$
$$\blacklozenge \Box \mathit{propose}(X) \wedge \blacksquare \Box \mathit{propose}(X) \tag{11}$$

4.4 OPERATIONAL SEMANTICS AND AN ALGORITHM

4.4.1 *Characteristic Formula*

The strategies presented are designed by describing transition systems. These strategies have the important property that they are satisfied by exactly one transition system. Thus, the meaning of these strategies is completely determined by the semantics of the strategy language. Now the question arises, whether every transition system can be defined by a strategy. The answer is positive. We will define the *Characteristic For-*

mula of a transition system in this section. All the strategies presented are equivalent to characteristic formulas. We further present a method which combines the transition systems defined by a set of strategies. Our method will have the property that it maximises the chance that a consistent diagnostic process is found. Finally, the method is exploited by a simple iterative algorithm.

Given a transition system $(S, \rightarrow, t)$, we can systematically define a formula, which completely characterises $(S, \rightarrow, t)$:

Definition 4.20 *Characteristic Formula*
Let $(S, \rightarrow, t)$ be a transition system, WHYP a set of working hypotheses and $s \in S$. Then G_s is called State Formula of s, *F_s* is called *Characteristic Formula of s and F_t is called* Characteristic Formula *of $(S, \rightarrow, t)$, where*

$$\begin{aligned} G_s &= \Box \bigwedge \{wh \mid wh \in s\} \cup \{\neg wh \mid wh \in (\bigcup S) \backslash s\} \\ F_s &= G_s \wedge \bigwedge_{s \rightarrow s'} \blacklozenge F_{s'} \wedge \blacksquare \bigvee_{s \rightarrow s'} F_{s'} \end{aligned}$$

Lemma 4.21 *Let $(S, \rightarrow, \emptyset)$ be a transition system and $s, s' \in S$ then $(S, \rightarrow, \emptyset) \models_s G_{s'}$ iff $s = s'$.*

The above lemma states that the state formula G_s holds only at state s and is false at any other state. Thus the first conjunct G_s of the formula F_s *fully* characterises the current state s, the second conjunct manifests the existence of the successor states and the third conjunct states that there are no other successors.

Example 4.22 *For the strategy structural refinement in section 4.3.2, the head of the strategy is a characteristic formula for the transition system with the only transition from the empty set to the state $\{refine(C)\}$.*

In general, all the strategy heads in this paper are equivalent to characteristic formulas. The following theorem shows that we can uniquely characterise a given transition system by a set of strategies.

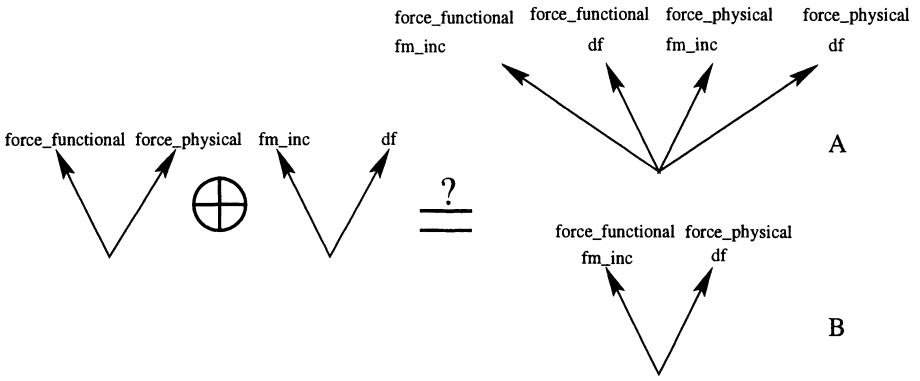
Theorem 4.23 *Given a transition system $(S, \rightarrow, \emptyset)$, then there is a set $\mathcal{F}$ of one-step strategies, such that $(S', \rightarrow', \emptyset) \models \mathcal{F}$ iff $(S', \rightarrow', \emptyset) = (S, \rightarrow, \emptyset)$.*

We conclude that one-step strategies are the "smallest" language that captures all possible transition systems, because we need at least one level of $\blacksquare$ and $\blacklozenge$–Modalities to describe transitions and we have now shown that one level is also sufficient to describe a given transition system.

4.4.2 *Combining Strategies*

When we consider more than one strategy formula we have to solve the problem of combining the proposed transitions. Suppose we have two strategy formulas $C_1 \rightarrow H_1$ and $C_2 \rightarrow H_2$ and the current state of the process satisfies both C_1 and C_2. How do we combine the transitions proposed by H_1 and H_2?

Example 4.24 *Recall the strategies* (6) *of preferring single fault diagnoses over double faults and incomplete fault modes and the strategy* (1) *of activating the physical model. Assume the bodies of* (6) *and* (1) *are satisfied in the current state and we have to perform transitions to satisfy the heads* $\blacklozenge\Box fm\_inc \wedge \blacklozenge\Box df \wedge \blacksquare(\Box df \not\leftrightarrow fm\_inc)$ *and* $\blacklozenge\Box force\_functional \wedge \blacklozenge\Box force\_physical \wedge \blacksquare\Box(force\_functional \leftrightarrow \neg force\_physical)$, *respectively. There are several transition systems satisfying the conjunct of these two heads:*



Solution B is not desired as it includes only two of four possible combinations of the working hypotheses.

Formally, the independence of two strategies proposing working hypotheses wh_1 and wh_2 means that looking at a state in which wh_1 is active, we cannot derive the truth value of wh_2 in that state.

Definition 4.25 *Independence of Strategies*
Let F be a set of strategies. Let $(S, \rightarrow, t)$ be a consistent transition system wrt. F. The state $s \in S$ satisfies Independence of Strategies, *iff there is no transition system $(S_1, \rightarrow_1, t_1)$ consistent with F such that*

1. *for all working hypotheses $wh_1, wh_2 \in S \cup \neg S$ such that $(S_1, \rightarrow_1, t_1) \models_s \blacksquare\Box wh_1 \rightarrow wh_2$ we have $(S, \rightarrow, t) \models_s \blacksquare\Box wh_1 \rightarrow wh_2$*

2. *there are working hypotheses* $wh_1, wh_2 \in S \cup \neg S$ *such that* $(S, \rightarrow, t) \models_s \blacksquare\Box wh_1 \rightarrow wh_2$ *but* $(S_1, \rightarrow_1, t_1) \not\models_s \blacksquare\Box wh_1 \rightarrow wh_2$

The transition system $(S, \rightarrow, t)$ satisfies independence of strategies iff every state $s \in S$ satisfies independence of strategies. Treating strategies as independent has several advantages: When writing down strategies we explicitly specify dependencies among certain hypotheses. Independence to other hypotheses has not to be specified. This is important in case a strategy formula is added to a large set of existing formulas. Furthermore, assuming independence maximises our chance to find a solution in the case of non–determinism. If this transition system does not lead to a stable state, there will be no other.

In the following, we will show that there is only one transition system that satisfies independence of strategies for a given set of strategies. Thus the semantics is completely specified and can be computed efficiently. In order to combine the transition systems defined by the heads of two strategies while preserving independence, we simply combine the successor states in all possible ways. We call this operation the *State Product*.

Definition 4.26 *State Product*
Given two sets of states S_1 *and* S_2 *the* State Product $S_1 \otimes S_2$ *is defined as* $\{s_1 \cup s_2 \mid s_1 \in S_1, s_2 \in S_2\}$.

When constructing the successor transitions for a given state during the diagnostic process, we instantiate the strategies (quantification over components) and collect the heads H_i of the strategies $C_i \rightarrow H_i$, whose conditions C_i are satisfied. We construct the transition systems corresponding to the heads $\{H_i\}$. Then we combine them by applying the following theorem:

Theorem 4.27 *Let* $H_1, H_2, \ldots H_n$ *be characteristic formulas of depth* 1 *which have no working hypotheses in common and let* $(S_1, \rightarrow_1, t_1), (S_2, \rightarrow_2, t_2), \ldots (S_n, \rightarrow_n, t_n)$ *be the corresponding transition systems. The following transition system* $(S, \rightarrow, t)$ *satisfies independence of strategies:*

$$S = \emptyset \cup \bigotimes_{i=1}^{n} (S_i \backslash \{\emptyset\}), \quad \rightarrow = \{(\emptyset, s) \mid s \in S\}, \quad t = \emptyset$$

4.4.3 A Strategy Algorithm

Theorem 4.27 describes how to compute the successor states of a given state under the assumption of independent strategies. Iterative application of this theorem yields a straightforward method for computing a transition system satisfying a given set of strategies. In a given state s starting with $\emptyset$ we have to execute the following steps:

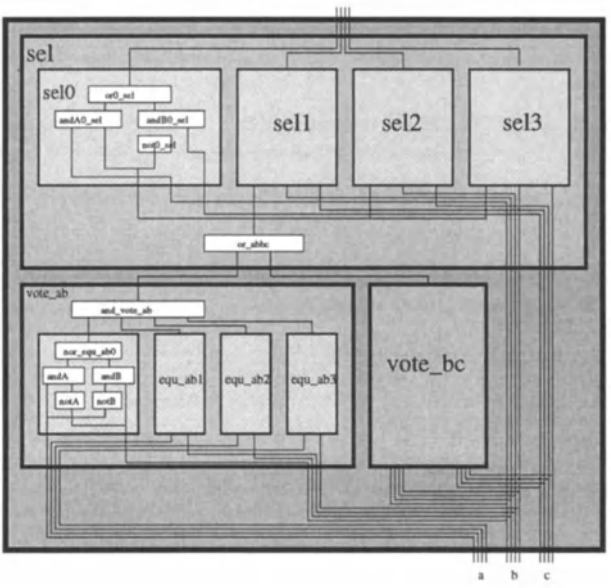


Figure 4.2. Voter.

1. Compute the diagnoses and corresponding system models under state s.
2. Instantiate the body of the strategies using the current diagnoses/models. Collect the heads of the satisfied strategies.
3. Construct a transition system for each head.
4. Combine the resulting transition systems using state product.

   The method must be recursively applied to every generated state.

4.4.4 *Voter Example*

A voter (see Fig. 4.2) has three 4-bit-inputs a, b, c. It outputs b if $(a = b) \vee (b = c)$ and otherwise c. The equality check is realised by the components *vote_ab* and *vote_bc*. Both are composed of an and-gate and 4 comparators *equ_xy*, which serve as inputs for the and-gate. A comparator *equ_xy* compares 2 bits by realizing the boolean function $x\bar{y} \bar{\vee} \bar{x}y$ and thus consists of 2 not- and 2 and-gates and a nor-gate. The select component in turn contains 4 one-bit-selectors sel_i which are controlled by the or-gate *or_abbc_sel*. If it is high, selector sel_i lets b_i pass, otherwise c_i passes. This is realised by 2 and-gates, an or- and a not-gate.

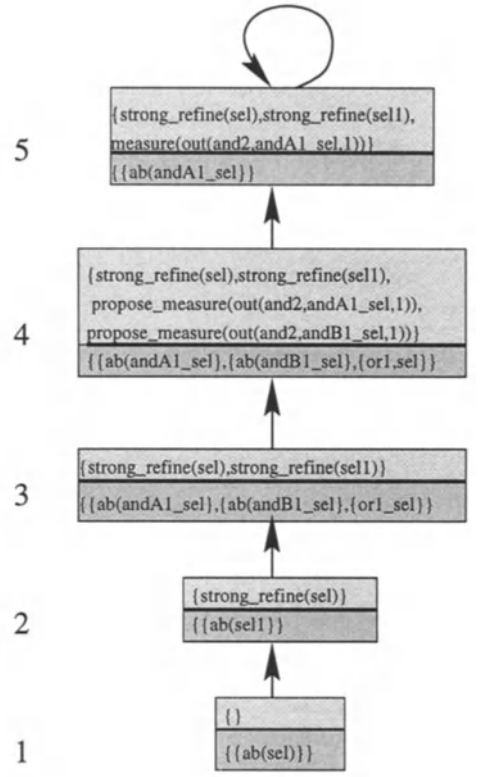


Figure 4.3. Voter example 1.

Voter Example 1. The voter inputs are $a = 0000$, $b = 0000$ and $c = 0001$, which should lead to the output 0000, because $a = b$. However, we observe the value 0010. The diagnostic process shown in Fig 4.3 starts with the empty set of assumptions where the single fault assumption is active by default (see section 4.3.4). In the first diagnosis step we infer that the abstract component *sel* is abnormal (1). Since the condition of the structural refinement strategy is satisfied, the system assumes *strong_refine*(*sel*) in the next step (2) and the abstract component *sel* is replaced by its subcomponents. Computing diagnoses reveals *sel1* as the faulty component within *sel*. Consequently, the hypothesis *strong_refine*(*sel1*) is additionally adopted (note the presence of the monotonicity axiom for structural refinement) (3). Now the process has reached the gate level of the circuit, where three components can explain the failure: *andA1_sel*, *andB1_sel* and *or1_sel*. The precondition of the measurement strategy

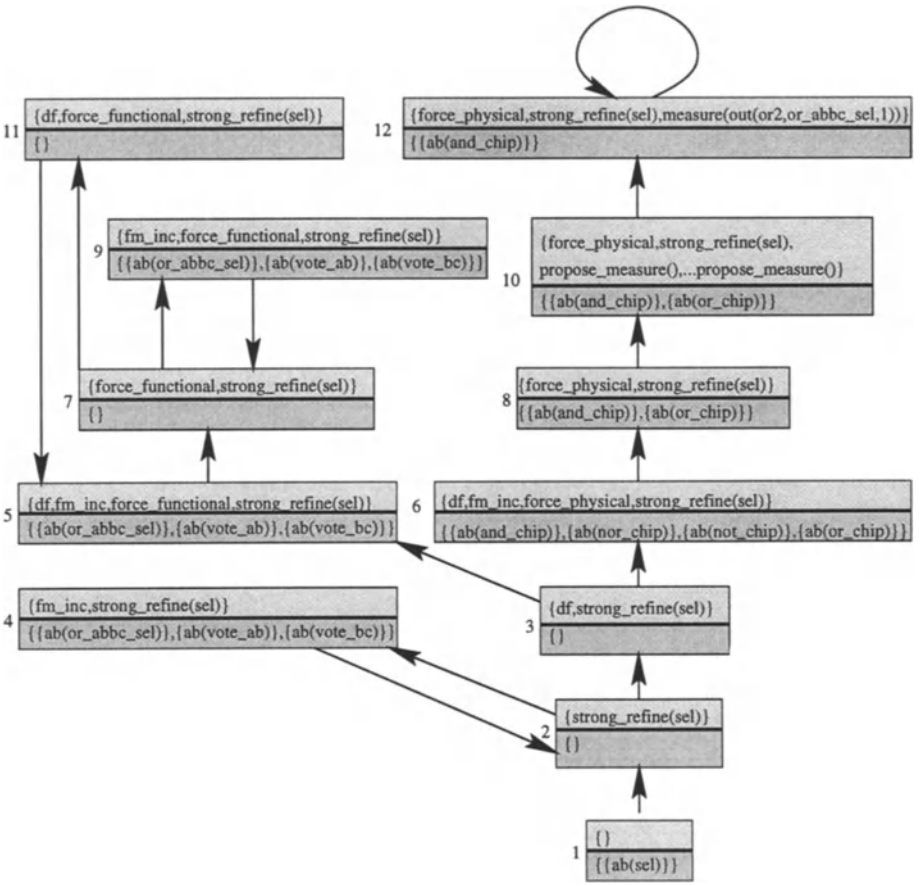


Figure 4.4. Voter example 2.

is now satisfied because the three diagnoses entail different values for the measure points $out(and2, andA1\_sel, 1)$ and $out(and2, andB1\_sel, 1)$ and no further refinement strategy is applicable. For these points a measurement is proposed (4). There is an intermediate step in which we obtain the same diagnoses. Now the measurement $out(and2, andA1\_sel, 1)$ is selected by activating the procedure *choose* in the system description. Measuring at this point yields the unique diagnosis *andA1_sel* (5) and the state $\{strong\_refine(sel), strong\_refine(sel0), measure(out(and2, andA1\_sel, 1))\}$ turns out to be stable.

Voter Example 2. In the process depicted in Fig. 4.4 the three input words are all 0000 and the output is observed to be 1111. The top level diagnosis uniquely identifies *sel* as abnormal (1), but the following refinement does not lead to any diagnoses (2). So the single-fault-assumption is relaxed and two successor states are created, allowing double faults and incomplete fault models, respectively. With incomplete fault modes some diagnoses are found (4). Since the hypothesis of incomplete fault modes is not monotonic, we have to drop it again. The consequence is a loop between two states, in which only the state with incomplete fault modes is consistent. By definition 4.17 we have reached a weakly stable state. The search for double faults (3) in the other branch is not successful. Two strategies apply in this situation. In all successor states we have to allow incompleteness of fault models in addition to double faults (section 4.3.4). Furthermore we have to branch between physical and functional model as proposed by the multiple views strategy (section 4.3.1). Two successor states are generated:

- With double faults and incomplete fault modes three diagnoses are found (5). The search for more preferred diagnoses first leads to no diagnoses (7). Allowing double faults does not help (11), while dropping the completeness of fault modes assumption yields three single faults (9), so that this state is again weakly stable.

- Beside the computations in the functional model, we obtain diagnoses of the physical model (6). With double faults and incomplete fault modes allowed, five diagnoses are consistent with the observation. Thus in the next step the preferences are relaxed and *and_chip* and *or_chip* are valid diagnoses in the physical view (8). In order to discriminate among those two diagnoses several measurements are proposed (10). Among them the point *or_abbc_sel* is chosen and finally the state $\{force\_physical, strong\_refine(sel), measure(out(or2, or\_abbc\_sel, 1))\}$ is stable.

Voter Example 3. The three input words and the output are 0001, 1010, 0100 and 1010 respectively. The first diagnoses (1) show that all abstract components may be responsible (see Fig. 4.5). Thus the components are weakly refined (2) and we identify the two and-gates *and_vote_ab*, *and_vote_bc* and the or-gate *or_abbc_sel* as suspicious. Now the measurement strategy proposes the outputs of the and-gates for measurement (3). Finally *and_vote_ab* is chosen for measurement and we obtain the final single-fault diagnosis that *and_vote_ab* is abnormal (4).

4.5 EXTENSIONS OF THE STRATEGY LANGUAGE

Motivated by a shortcoming in the initial work on the strategy language [FNS94, DNPS95, NFS95], we extend the language by an until-operator [Sch95]. It enriches the language's expressiveness and is a continuation of the previous approaches: initially Damásio, Nejdl, and Pereira introduced a preference relation over *single* diagnoses [DNP94], Fröhlich, Nejdl, and Schroeder defined the strategy language

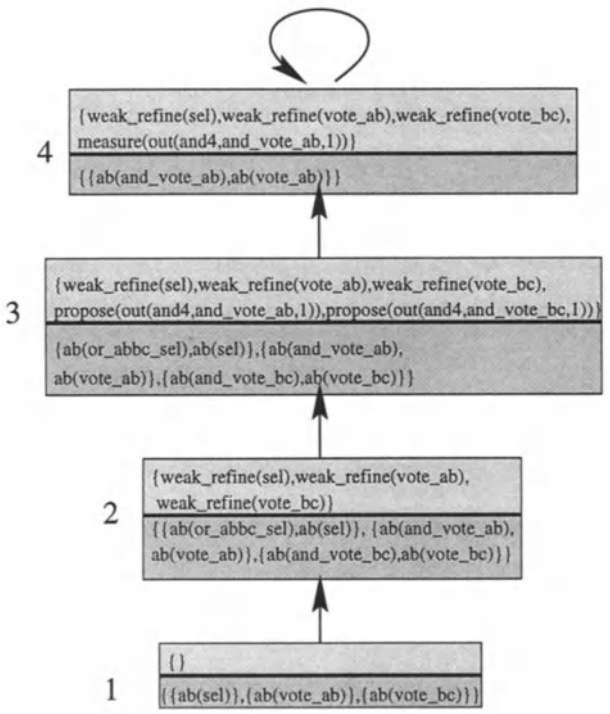


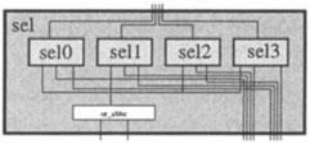
Figure 4.5. Voter example 3.

whose semantics can be viewed as a preference relation over *sets* of diagnoses [FNS94, NFS95, FNS96, FNS97] and finally, the extension of the until-operator defines a preference relation over paths of the diagnostic process, i.e. over *sets of sets* of diagnoses.

4.5.1 Motivating Example

The initial definitions of the strategy language [NFS95] focus on monotonic strategies, but preferences require non-monotonicity.

Example 4.28



Consider a 4-bit-selector sel being composed of an or-gate and four 1-bit-selectors sel_i with sel_1 and sel_2 being faulty. Thus, sel is also faulty. We specify that we prefer single over double faults and use the strategy of structural refinement.

Initially the single fault $\{ab(sel)\}$ is found and sel is refined. In the next step there are no no single-faults as both sel_1 and sel_2 are faulty. But due to the monotonic adoption of working hypotheses in [NFS95] the single fault assumption cannot be dropped anymore. This is unsatisfactory, as the process should terminate with the result that insisting on single faults fails after a refinement, so that double faults are tried and after the refinement the double fault sel_1 and sel_2 is found.

Much of the problems concerning [NFS95] and detected in [Sch95] are caused by the monotonicity when computing the diagnosis process and the treatment of the absence of diagnoses. Our approach differs in two respects from the initial work [NFS95].

1. Our algorithm is non-monotonic. A working hypothesis is only kept if this is explicitly required, otherwise it is dropped.

2. The absence of diagnosis is not a dead end in the process. In definition 4.12 of the induced model the case of no diagnoses is treated separately. The corresponding world is associated with a model including the working hypothesis that are responsible for no diagnoses together with the literal $ab(SD)$.

Therefore the diagnostic process is capable of detecting the none-existence of diagnoses and thus may continue if no diagnoses are found. So it is also possible to loosen the restrictions which are imposed by the working hypotheses and lead to no diagnoses. In fact, the preferences defined in section 4.3.4 make heavy use of the special $ab(SD)$ literal.

Example 4.29 *In our framework, the example mentioned above is solved as follows. Initially, the single-fault sel is detected. The single-fault assumption is kept and the refinement yields therefore no diagnoses. Subsequently, the model* $\{sf, refine(sel), ab(SD)\}$ *is generated. With* $ab(SD)$ *in the model the preference strategy (6) in section 4.3.4 drops sf and allows for double-faults. Therefore the interplay of non-monotonicity and* $ab(SD)$ *solves the problem.*

While preferences and none-existence of diagnoses are handled properly in our language, [Sch95] points out a strategy that lies beyond the expressiveness of our language: preferences of strategies. If we have two strategies F and G, we may want to start by using F and only if all processes based on F end up with no solution, then we try strategy G. To do so we need an additional operator AF to express that a formula holds finally on all branches of the process. Then we start using F and check by AF $\Box$false if all paths end finally in the inconsistent state. If this is the case then strategy G is activated:

Definition 4.30 *Preference of Strategies*
Given two strategy formulas F and G, we prefer F over G by the strategy formula

$$F \wedge ((AF\,\Box\text{false}) \rightarrow G)$$

The operator AF can be derived from the until-operator **U** where $F\mathbf{U}G$ means that F holds until G holds. In order to give a formal semantics to the AF operator we define the declarative semantics of the until operator **U** and show how the operator AF and some other operators can be derived from **U**.

4.5.2 The Until Operator

The basic reason why the problems cannot be properly handled by the initial proposal is that they cannot express an unlimited look ahead into the diagnostic process. Intuitively, this corresponds to an infinite strategy formula. In modal logics there are several solutions to this problem. Two familiar approaches are the introduction of an until operator [Eme90] and a minimal fixpoint operator [Koz83], respectively. As the latter is more complex and unintuitive than the former we will use the until operator.

Definition 4.31 *Semantics of* **U**
Let $M = (W, D, \rightarrow_1, \rightarrow_2, I)$ *be an* $\mathcal{L}_{Strat}$*-model,* $w_1 \in W$, $(w_1, w_2, \ldots)$ *a full path in* $\rightarrow_1$*, and* F, G $\mathcal{L}_{Strat}$*-formulas.*

$$M \models_{w_1,(w_1,w_2,\ldots),\alpha} F\mathbf{U}G \quad \textit{iff} \quad \begin{array}{l} \textit{ex. } i \textit{ s.t. } M \models_{w_i,(w_i,w_{i+1},\ldots),\alpha} G \\ \textit{f.a. } j \textit{ s.t. } j < i \; M \models_{w_j,(w_j,w_{j+1},\ldots),\alpha} F \end{array}$$

Intuitively, we extend the definition by an the argument of a path. Then $F\mathbf{U}G$ holds, if F holds in all states of the path until finally a state satisfying G is reached. Considering one path only does not reveal the full complexity of the operator **U**. Adding

the additional path argument to the old definition we see the complexity: for ■ and ♦ we do not just consider all and one successor state, respectively, but all and one *path* starting from a successor state. With the path as argument the semantics of the modalities is defined as follows:

Definition 4.32 *Semantics of* ■ *and* ♦
Let $M = (W, D, \rightarrow_1, \rightarrow_2, I)$ *be an* $\mathcal{L}_{Strat}$*–model,* $w_1 \in W$ *and* $(w_1, w_2, \ldots)$ *a path in* $\rightarrow_1$.

$$
\begin{array}{lll}
M \models_{w,\alpha} \blacksquare F & \textit{iff} & \textit{f.a. } w_1 \in W \textit{ s.t. } w \rightarrow_1 w_1 \textit{ and } ws = (w_1, w_2, \ldots) \\
 & & \textit{is a full path starting with } w_1\textit{, we have } M \models_{w_1,ws,\alpha} F \\
M \models_{w,\alpha} \blacklozenge F & \textit{iff} & \textit{ex. } w_1 \in W \textit{ s.th. } w \rightarrow_1 w_1 \textit{ and } ws = (w_1, w_2, \ldots) \\
 & & \textit{is a full path starting with } w_1 \textit{ and } M \models_{w_1,ws,\alpha} F
\end{array}
$$

$M \models_w F$ holds iff $M \models_{w,ws} F$ holds for all full paths starting with w. The semantics of logical connectors and $\Box$ and $\Diamond$-operator only needs the additional argument of a path ws.

Once we have defined the until operator we are able to give a bunch of new operators to allow formulation of propositions about possibly infinite look ahead into the diagnostic process. With these operators the user has a powerful tool at hand to define backtracking explicitly and declaratively as a part of the strategy language.

The operator AF F means that on all paths F holds finally; in the same manner AG F means that F holds generally on all paths. EF F and EG F express the same for one path, i.e. there is one path such that F holds finally and generally, respectively.

Definition 4.33 AF, AG, EF, EG, U'
The operators AF, AG, EF, EG and U' can all be specified by means of the **U***operator.*

$$
\begin{array}{rcl}
AF\,F & = & F \vee \blacksquare(\text{true}\mathbf{U}F) \\
AG\,F & = & F \wedge \blacksquare(F\mathbf{U}\text{false}) \\
EF\,F & = & F \vee \blacklozenge(\text{true}\mathbf{U}F) \\
EG\,F & = & F \wedge \blacklozenge(F\mathbf{U}\text{false}) \\
F\,U'\,G & = & F\mathbf{U}G \vee AG\,F
\end{array}
$$

The meaning of the operators can be visualised as shown in Figure 4.6. To summarise, the until-operator introduced in [Sch95] extends the strategy language's expressiveness to reason about paths of the diagnostic process. It has been introduced to deal with some shortcomings of the initial proposal [NFS95] concerning preferences and measurements. The reason for the unsatisfactory solution in [NFS95] is that preferences require non-monotonicity and that the absence of diagnosis should not be a dead-end in the diagnostic process. In our framework we take these two shortcomings into account and subsequently solve the problems mentioned in [Sch95]. In contrast to [Sch95], our solution of a non-monotonic process and a literal $ab(SD)$ to indicate no diagnoses fits elegantly into the framework and is efficiently computable.

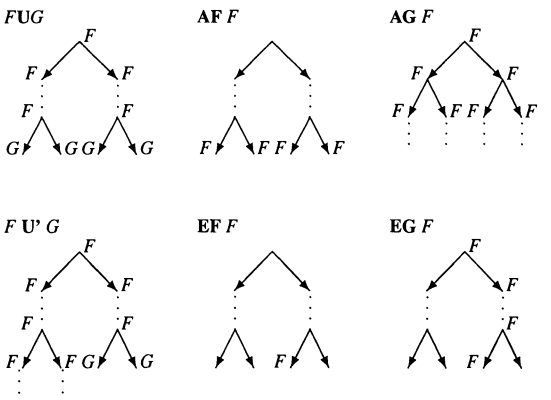


Figure 4.6. The meaning of the operators U, AF, AG, U', EF, EG.

4.6 SUMMARY

To cope with large-scale systems the theory of model-based diagnosis has been extended to include concepts such as multiple views [Dav84, Ham91], hierarchies [Dav84, Ham91, Gen84, Moz91, BD94], preferences [DNP94, FNS94] and measurements [dKRS91]. Struss introduced the idea of diagnosis as process [Str92b], further developed by Böttcher and Dressler [BD93, BD94]. We formalised it by defining a meta-language that allows to describe the process declaratively [NFS95, FNS97]. We focused on two important issues. First, we showed how to design strategies to cover the concepts mentioned above. Second, we developed an operational semantics and an algorithm that processes these strategies and efficiently computes the diagnostic process. We identified generic one-step strategies to deal with monotonic and non-monotonic working hypotheses as well as deterministic and non-deterministic strategies. In particular, the combination of non-monotonic working hypotheses and non-determinism allowed us to express preferences which usually have to be treated in a separate framework [DNP94, FNS94]. We showed how to use multiple views and how to employ hierarchies by the strategy of structural refinement. We integrated measurement strategies using procedural attachment. Beside the practical motivation of one-step strategies we proved that one-step strategies are universal, i.e. every diagnostic process can be fully characterised by a set of one-step strategies. Furthermore, we defined characteristic formulas and independence of strategies which lead to an efficient algorithm that covers the whole variety of strategies. The design of these strategies was evaluated in the domain of digital circuits using the voter circuit from [Isc85].

5 AUTONOMOUS AGENTS

We introduce vivid agents to develop distributed monitoring and diagnosis systems consisting of a variety of scalable knowledge- and perception-based agents. We develop an execution model for vivid agents which is based on an architecture for concurrent action and planning. To implement vivid agents we use PVM-Prolog which provides coarse-grain parallelism to spawn agents in a network, and fine-grain parallelism to run an agent's perception-reaction-cycle and planning facility concurrently. Finally, we evaluate the concept of vivid agents in distributed diagnosis including fault-tolerant diagnosis and the diagnosis of an unreliable communication protocol.

5.1 INTRODUCTION

A multi-agent approach to the diagnosis of distributed systems has several advantages: it achieves fault tolerance without any special hardware, diagnosis agents just need parts of the overall system description which is often too large or not available, and the system can be extended and maintained more easily. Multi-agent diagnosis involves several tasks: agents continuously monitor the operation of the subsystems they are

M. Schroeder, *Autonomous, Model-Based Diagnosis Agents*

assigned to; if they detect any suspicious behaviour they may run specific tests in order to confirm or abandon a certain suspicion; they may communicate their findings to other competent agents, and they may receive requests to run certain tests from other agents; given the findings of other agents they have to compute diagnoses and inform other agents about them; if there is no agreement on the right diagnosis of a malfunction there has to be a mechanism how to resolve conflicts and reach consensus (e.g. by some voting scheme, or by negotiation).

Thus, a diagnosis agent must be capable of performing tests concerning its own state, to compute diagnoses based on other agents' test results and to find a consensus among different diagnosis results. To meet these requirements a diagnosis agent needs an expressive knowledge base that contains a description of the system to be diagnosed and strategic knowledge to deal with hierarchies, multiple models, measurements, and preferences. All these requirements are met by vivid agents [Wag96, SMWC97], which allow various forms of knowledge representation including the CWA, deduction rules with negation-as-failure, default rules with two kinds of negation, and the rule-based specification of inter-agent cooperation.

The concept of vivid agents comprises both reactive and pro-active behaviour competences. The reactive behaviour of vivid agents is, however, not hardwired in the form of fixed stimulus-response schemes but encoded in the form of reaction rules with a twofold premise: a triggering perception event and an epistemic condition referring to the current knowledge state of the agent. Thus, the reactions of vivid agents may depend on the result of deliberation which is essential for diagnosis agents. In our present implementation the knowledge base of an agent has the form of an extended logic program [GL90], thus allowing for the representation of various forms of incomplete knowledge and of default rules. In combination with contradiction removal [PAA91, AP96], extended logic programs turned out to be sufficiently expressive and computationally tractable [DPS97b] to represent and solve many model-based diagnosis problems.

An architecture to realise a vivid agent involves four main components: a knowledge base, reaction patterns, a planner, and a plan execution facility. Since action execution and planning are two nearly independent tasks it is useful to separate them in order to achieve greater modularity and efficiency by running two processes for action and planning in parallel. We call this architecture CAP for Concurrent Action (subsuming both reactions and planned actions) and Planning. To implement the CAP architecture we use PVM-Prolog [CM96, CM97], a Prolog core extended by an interface to PVM, the Parallel Virtual Machine, a standard software which allows to view a network of heterogeneous machines as a single parallel computer. Besides PVM's coarse-grain parallelism, PVM-Prolog includes a process-internal thread concept to realise fine-grain concurrency. The coarse-grain parallelism is used to spawn agents in a network, while the fine-grain concurrency is used to run a perception-reaction-cycle and a planning facility for each agent concurrently. The implementation of

vivid agents meets all the requirements of a state-of-the-art programming language for multi-agent systems [SMWC97, FMS+97]:

- **Reactive and Pro-active Behaviour.** The language supports both reactive and pro-active behaviour. I.e. on the one hand the agents are capable of reacting timely to events and on the other hand they can pursue long-term goals based on some planning facilities.
- **Formal Semantics.** Multi-agent applications are complex and therefore likely to be error-prone. Since many applications require high safety standards the language needs a formal semantics in order to simulate and verify the behaviour of the programs. The transition semantics given for the CAP architecture perfectly matches these needs.
- **Executable Specifications.** The specification of agent behaviour is declarative and executable.
- **Platform Independence.** Applications are usually developed on a different platform than they are later run on. Furthermore, hardware may be updated and is subject to change. Our vivid agent implementation is portable and hardware-independent supporting any UNIX-based machine.
- **Heterogeneity.** The language supports the use of agents based on different concepts and architectures and implemented in different programming languages.
- **Modularity and Scalability.** The agent architecture underlying the language is modular, so that different types of agents can be composed. For example, a diagnosis agent [SdAMP97, SW97] needs a powerful knowledge base and little planning, whereas a scheduling agent may need an optimal planner and only little knowledge representation functionality such as in a relational database.

5.2 VIVID AGENTS

A *vivid agent* is a software-controlled system whose state comprises the mental components of knowledge, perceptions, tasks, and intentions, and whose behaviour is represented by means of *action* and *reaction rules*. The basic functionality of a vivid agent comprises a knowledge system (including an update and an inference operation), a perception (event handling) system, and the capability to represent and perform reactions and actions in order to be able to react to events, and to generate and execute plans.

Reactions may be immediate and independent from the current knowledge state of the agent but they may also depend on the result of deliberation. In any case, they are triggered by events which are not controlled by the agent. A vivid agent without the capability to accept explicit tasks and to solve them by means of planning and plan

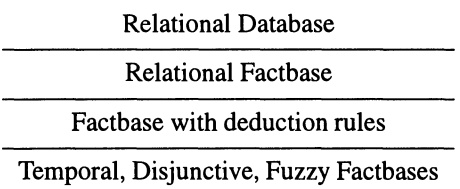


Figure 5.1. Knowledge systems with different complexity.

execution is called *reagent*. The tasks of reagents cannot be assigned in the form of explicit ('see to it that') goals at run time, but have to be encoded in the specification of their reactive behaviour at design time.

The concept of vivid agents is not based on a specific logical system for the knowledge-base of an agent. Rather, it allows to choose a suitable knowledge system for each agent individually according to its domain and its tasks. In the case of diagnosis agents, extended logic programs proved to be an appropriate form of the knowledge base of an agent because it is essential for model-based diagnosis to be able to represent negative facts, default rules, and constraints.

5.2.1 *Vivid Knowledge Systems*

The knowledge system of a vivid agent is based on three specific languages: L_{KB} is the set of all admissible knowledge bases, L_{Query} is the query language, and L_{Input} is the set of all admissible inputs, i.e. those formulas representing new information a KB may be updated with. In a diagnosis setting, L_{Input} may be $\{test(\_,\_), diagnoses(\_,\_)\}$, where *test* is used to update other agents' test results and *diagnoses* to update the agents' diagnosis results. The input language defines what the agent can be told (i.e. what it is able to assimilate into its KB); the query language defines what the agent can be asked. Let L be a set of formulas, then L^0 denotes its restriction to closed formulas (sentences). Elements of L^0_{Query}, i.e. closed query formulas, are also called *if-queries*.

Knowledge systems of various complexity can be distinguished (see Figure 5.1). The knowledge system of relational databases allows only atomic sentences and requires complete information. Conservative extensions of it are called vivid [Wag94]. Formally we can define a vivid knowledge system as follows:

Definition 5.1 *Vivid Knowledge System*
An abstract knowledge system [Wag95] consists of three languages and two operations: a knowledge representation language L_{KB}, a query language L_{Query}, an input language L_{Input}, an inference relation $\vdash$, such that $X \vdash F$ holds if $F \in L^0_{\text{Query}}$ can be inferred from $X \in L_{\text{KB}}$, and an update operation Upd, *such that the result of updating $X \in L_{\text{KB}}$ with $F \in L^0_{\text{Input}}$ is the knowledge base* $\mathsf{Upd}(X,F)$.

Positive vivid knowledge systems use a general Closed-World Assumption, whereas general vivid knowledge systems employ specific Closed-World Assumptions and possibly two kinds of negation. Relational databases can be extended to a general vivid knowledge system, called relational factbases, by allowing for literals instead of atoms as information units. Further important examples of positive vivid knowledge systems are temporal, fuzzy, and disjunctive factbases. All these kinds of knowledge bases can be extended to deductive knowledge bases by adding deduction rules of the form $F \leftarrow G$ [Wag95].

A knowledge base consisting of a consistent set of ground literals, i.e. positive and negative facts, is called a relational factbase. In a factbase, the CWA does not in general apply to all predicates, and therefore in the case of a non-CWA predicate, negative information is stored along with positive. By storing both positive and negative information, the KB can represent incomplete predicates. The schema of a factbase stipulates for which predicates the CWA applies by means of a special set *CWRel* of relation symbols. Explicit negative information is represented by means of a *strong* negation $\neg$.

Example 5.2 *In the telecommunication scenario in section 3.2.3 a base transceiver station BTS sends by default alive signals to the base stations BSC. In the case of a failure alarm the alive signal is explicitly not present at the BSC (left rule below). The CWA does not apply to signal, but for the ab-predicate we use the CWA, since we want to assume by default that components are not abnormal (right rule). In contrast to this, we can distinguish the two cases that we have explicitly derived that there is no signal and that we do not have information about the signal. I.e.* $X \vdash \neg signal(bsc, down, bts20, alive)$ *means that we know explicitly that no alive signal of* $bts20$ *arrived, whereas* $X \vdash not\ signal(bsc, down, bts20, alive)$ *only expresses that we cannot infer whether there is a signal, which means that there is none or that there is no information about it.*

$$\neg signal(bsc, down, Sender, alive) \leftarrow bts\_failure\_alarm(Sender), type(Sender, bts). \quad \Big| \quad signal(NE, up, Sender, Signal) \leftarrow not\ ab(NE), signal(NE, down, Sender, Signal).$$

Definition 5.3 *Deduction in Factbases*
As a kind of natural deduction from positive and negative facts an inference relation $\vdash$ *between a factbase X and an if-query is defined in the following way:*

$$\begin{array}{lll} (\vdash not\ ) & X \vdash not\ p(c) & \text{if} \quad p(c) \notin X \\ (\vdash \neg) & X \vdash \neg p(c) & \text{if} \quad \neg p(c) \in X \\ (\vdash \neg_{\mathrm{CWA}}) & X \vdash \neg p(c) & \text{if} \quad p \in CWRel\ \&\ X \vdash not\ p(c) \end{array}$$

where $p(c)$ *stands for an atomic sentence with predicate p and constant (tuple) c.*

The negations $\neg$ and *not* are called *strong* and *weak* since the coherence principle holds and thus $X \vdash \neg F$ implies $X \vdash not\ F$. A factbase X answers an if-query F by yes if $X \vdash F$, by no if $X \vdash \neg F$, and by unknown otherwise.

Definition 5.4 *Updates in Factbases*
Updates are recency-prefering revisions:

$$\mathsf{Upd}(X, p(c)) := \begin{cases} X \cup \{p(c)\} & \text{if } p \in CWRel \\ X - \{\neg p(c)\} \cup \{p(c)\} & \text{else} \end{cases}$$

$$\mathsf{Upd}(X, not\ p(c)) := \begin{cases} X - \{p(c)\} & \text{if } p \in CWRel \\ X - \{p(c)\} \cup \{\neg p(c)\} & \text{else} \end{cases}$$

The extension of factbases by adding deduction rules leads to extended logic programs with two kinds of negation as introduced in chapter 3. Inference in extended logic programs can be defined either top-down by the *demo*-predicate in Figure 3.26 or bottom-up by a fixpoint semantics [GL90, PA92, AP96].

Besides a knowledge system that captures the agent's information processing we have to specify agent behaviour. We distinguish reactive and pro-active behaviour. Reactive behaviour is specified by reaction rules. Triggered by an event a condition is evaluated wrt. to the agent's knowledge base and in case it holds the corresponding actions are performed. Action rules specify how actions cause effects if they are applicable. Given action rules and a goal, a planner can compute an action sequence to achieve the goal.

5.2.2 Reaction Rules

Reaction rules encode the behaviour of vivid agents in response to perception events created by the agent's perception subsystems, and to communication events created by communication acts of other agents. They are similar to 'event-condition-action' rules as used in active databases [MD89, KGB+95]. We distinguish between epistemic, physical, and communicative reaction rules.

Definition 5.5 *Reaction Rule*
Let A_1, A_2 be agent names. Let $L_{\text{Query}}, L_{\text{Input}}, L_{\text{Evt}}$, and L_{Act} be a query, update, perception and communication event, and physical action language, respectively. Then rules of the form

$$\begin{aligned} Eff &\leftarrow \mathsf{recvMsg}(\varepsilon(U), A_1), Cond && \textit{(Epistemic)} \\ \mathsf{do}(\alpha(V)), Eff &\leftarrow \mathsf{recvMsg}(\varepsilon(U), A_1), Cond && \textit{(Physical)} \\ \mathsf{sendMsg}(\eta(V), A_2), Eff &\leftarrow \mathsf{recvMsg}(\varepsilon(U), A_1), Cond && \textit{(Communicative)} \end{aligned}$$

where $Cond \in L_{\text{Query}}$, $Eff \in L_{\text{Input}}$, $\varepsilon(U) \in L_{\text{Evt}}$, $\alpha(V) \in L_{\text{Act}}$, $\eta(V) \in L_{\text{Evt}}$ are called epistemic, physical and communicative reaction rules.

The event condition recvMsg[$\varepsilon(U), A_1$] is a test whether the event queue of the agent contains a message of the form $\varepsilon(U)$ sent by some perception subsystem of the agent or by another agent identified by A_1, where $\varepsilon(U) \in L_{\text{Evt}}$ represents a perception or a communication event, and U is a suitable list of parameters. The epistemic condition $Cond \in L_{\text{Query}}$ refers to the current knowledge state, and the epistemic effect $Eff \in L_{\text{Input}}$ specifies an update of the current knowledge state. For physical reactions, L_{Act} is the language of all elementary physical actions available to an agent; for an action $\alpha(V) \in L_{\text{Act}}$, do($\alpha(V)$) calls a procedure realising the action α with parameters V. For the communicative reaction, sendMsg[$\eta(V), A_2$] sends the message $\eta \in L_{\text{Evt}}$ with parameters V to the receiver A_2. Both perception and communication events are represented by incoming messages. In a robot, for instance, appropriate perception subsystems, operating concurrently, will continuously monitor the environment and interpret the sensory input. If they detect a relevant event pattern in the data, they report it to the knowledge system of the robot in the form of a perception event message.

In general, reactions are based both on perception and on knowledge. Immediate reactions do not allow for deliberation. They are represented by rules with an empty epistemic premise, i.e. $Cond = true$. Timely reactions can be achieved by guaranteeing fast response times for checking the precondition of a reaction rule. This will be the case, for instance, if the precondition can be checked by simple table look-up such as in relational databases or factbases. Reaction rules are triggered by events. The agent interpreter continuously checks the event queue of the agent. If there is a new event message, it is matched with the event condition of all reaction rules, and the epistemic conditions of those rules matching the event are evaluated. If they are satisfiable in the current knowledge base, all free variables in the rules are instantiated accordingly resulting in a set of triggered actions with associated epistemic effects. All these actions are then executed, leading to physical actions and to sending messages to other agents, and their epistemic effects are assimilated into the current knowledge base.

Example 5.6 *The system FORKS simulates an automated loading dock where autonomous forklift agents load and unload trucks [Mül96, FMP96, MPT95, Mül94]. Consider the grid in Figure 5.2 with two forklift agents and several containers to be moved around. The agents may exchange their objectives in order to avoid collisions. In this case, a_2 sends to a_1 an acknowledgment and remembers that a_1 deals with container b if a_2 receives the corresponding message by a_1 and a_2 itself does not work with container b (left rule). Messages may also come from the agents' sensor subsystems. In case a_1's sensors indicate that it is standing in front of the box, a_1 may perform the physical reaction of taking the box (right rule).*

$$\text{sendMsg}(ok(lock(a_1, box(b))), a_1) \leftarrow \text{recvMsg}(lock(a_1, box(b)), a_1), i\_am(a_2), not\ lock(a_2, box(b))$$

$$\text{do}(take(box(b))) \leftarrow \text{recvMsg}(box(b), camera(a_1)), lock(a_1, box(b))$$

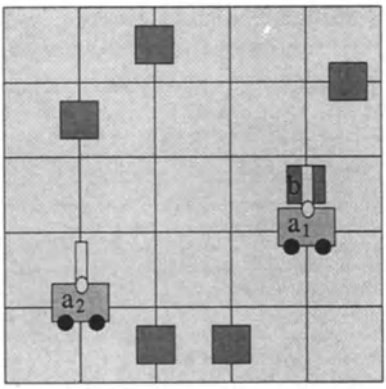


Figure 5.2. Two forklift agents a_1 and a_2 in a loading dock.

5.2.3 Action Rules

In addition to its reactive behaviour, an agent can exhibit pro-active behaviour. Action rules have the general form of *Action* $\leftarrow$ *Condition.* Depending on the domain and the capabilities and tasks of an agent, only certain actions can be performed by it. Besides communication acts, an agent can perform physical actions by means of its effectors. An action type corresponds to a set of action rules which represent the different execution conditions and effects of different situation contexts.

Definition 5.7 *Action Rule*
Let A be an agent name. Let $L_{\text{Query}}, L_{\text{Input}}, L_{\text{Evt}}$, *and* L_{Act} *be a query, update, perception and communication event, and physical action language, respectively. Then rules of the form*

$$\begin{aligned} r : \mathsf{do}(\alpha(V)), &\quad \mathit{Eff} \leftarrow \mathit{Cond} \\ r : \mathsf{sendMsg}[\eta(V), A], &\quad \mathit{Eff} \leftarrow \mathit{Cond} \\ r : &\quad \mathit{Eff} \leftarrow \mathit{Cond} \end{aligned}$$

where $\mathit{Cond} \in L_{\text{Query}}, \mathit{Eff} \in L_{\text{Input}}, \alpha(V) \in L_{\text{Act}}, \eta(V) \in L_{\text{Evt}}$ *are action rules.*

The formulas $\mathit{Eff} \in L_{\text{Input}}$ and $\mathit{Cond} \in L_{\text{Query}}$, are effects and preconditions of the actions. The procedure call $\mathsf{do}(\alpha(V))$ executes the physical action $\alpha(V) \in L_{\text{Act}}$; $\mathsf{sendMsg}[\eta(V), A]$ is a procedure call to execute the communicative event $\eta(V) \in L_{\text{Evt}}$ where A identifies the receiver of the outgoing message. r is the name of the action rule. We require that the precondition *Cond* is an evaluable formula and that all free variables of the epistemic effect formula *Eff* occur also in *Cond.*

Example 5.8 *In the loading dock example, agents have to plan their ways through the docks. Consider the case that a_1 has taken the container and has to load it on a truck at a given position. Based on its current position it has to plan the way to the truck. An action to move one step northbound in the loading dock can be specified by the action rule:*

$$move\_north : do(move\_north), not\ at(X,Y), at(X,Y') \leftarrow at(X,Y), Y' = Y + 1$$

Similarly, we can define actions move_east, move_west, move_south in order to move east, west, and south, respectively.

Action rules combine declarative inference with update. Since in general, the effects of an action may be context-dependent, an action is represented by a set of action rules expressing the dependence of different effects on different preconditions. The execution of an action in a situation described by the knowledge base X is realised by firing the corresponding action rule whose precondition *Cond* applies to X, i.e. $X \vdash Cond$.

Definition 5.9 *Rule Application*
Let $E \in L_{\text{Input}}$, $C \in L_{\text{Query}}$. An epistemic action rule $r : E \leftarrow C$ represents an update function, i.e. a mapping $r : L_{\text{KB}} \rightarrow L_{\text{KB}}$, whose application is defined as

$$r(X) := \mathsf{Upd}(X, \{E\sigma : \sigma \text{ is a substitution s.th. } X \vdash C\sigma\})$$

Example 5.10 *Assume a forklift agent is at position (1,1), i.e. $X = \{at(1,1)\}$. Applying the action rules move_east, move_north, move_north results in the knowledge base*

$$X' = move\_east(move\_north(move\_north(X))) = \{at(2,3)\}$$

Definition 5.11 *Plan*
In order to achieve a goal $G \in L^0_{\text{Query}}$ in a situation described by $X_0 \in L_{\text{KB}}$, an agent generates a suitable plan P being a sequence of action rules $r_1, \ldots, r_n$, such that when the corresponding sequence of actions is performed in X_0, it leads to a situation $P(X_0) \in L_{\text{KB}}$ where G holds:

$$P = r_n \circ \ldots \circ r_1\,, \quad \textit{and} \quad P(X_0) \vdash G$$

where P is applied to X_0 as a composed function.

Notice that our concept of planning on the basis of action rules in knowledge systems can be viewed as a generalisation of the STRIPS [FN71] paradigm which corresponds to planning on the basis the knowledge system of relational databases. Therefore, the frame problem is solved in the same way as in STRIPS: by means of a minimal change policy incorporated in the update operation of a knowledge system.

5.2.4 *Specification of Vivid Agents*

Since our agent concept is based on relational databases and its conservative extensions such as factbases and deductive factbases, we dub them vivid agents [Wag94]. Formally, vivid agents are defined as follows.

Definition 5.12 *Vivid Agent*
Let L_{KB}, L^0_{Query}, *and* L_{Evt} *be a knowledge representation, query, perception and communication event language, respectively. Then a vivid agent* $\mathcal{A} = \langle X, EQ, GQ, PQ, RR, AR \rangle$ *consists of a vivid knowledge base* $X \in L_{\text{KB}}$, *an event queue* EQ *being a list of instantiated event expressions* $\varepsilon(V) \in L_{\text{Evt}}$, *a goal queue* GQ *being a list of sentences* $G \in L^0_{\text{Query}}$, *an intention queue* PQ *being a list of goal/plan pairs, a set* RR *of reaction rules, and a set* AR *of action rules. A multi-agent system is a tuple of agents:* $\mathcal{A} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$.

Simple vivid agents whose mental state comprises only beliefs and perceptions, and whose behaviour is purely reactive, i.e. not based on any form of planning and plan execution, are called *reagents*.

Definition 5.13 *Vivid Reagent*
A reagent $\mathcal{A} = \langle X, EQ, RR \rangle$ *based on a vivid knowledge system consists of a knowledge base* $X \in L_{\text{KB}}$, *an event queue* EQ, *being a list of instantiated event expressions, and a set* RR *of reaction rules.*

We assume that RR is a consistent encoding of reactive behaviour in the sense that whenever a set of reaction rules is triggered by an event, the resulting actions are compatible with each other, i.e. the epistemic effects associated with them do not cancel out each other. To guarantee consistent actions requires techniques such as contradiction removal. The normative constraints described in [SdAMP97] solve this problem.

The execution of an agent, a reagent, and a multi-agent system is described by transitions for perception, reaction, plan execution, replanning, and planning [Wag96]. We will refine the operational semantics of vivid agents as initially proposed in [Wag96] to accommodate for concurrent action and planning.

5.3 CONCURRENT ACTION AND PLANNING

An architecture to realise a vivid agent involves four main components: a knowledge base, reaction patterns, a planner, and a plan execution facility. Since action and planning are two nearly independent tasks it is useful to separate them in order to achieve greater modularity and efficiency by running two processes for action and planning in parallel.

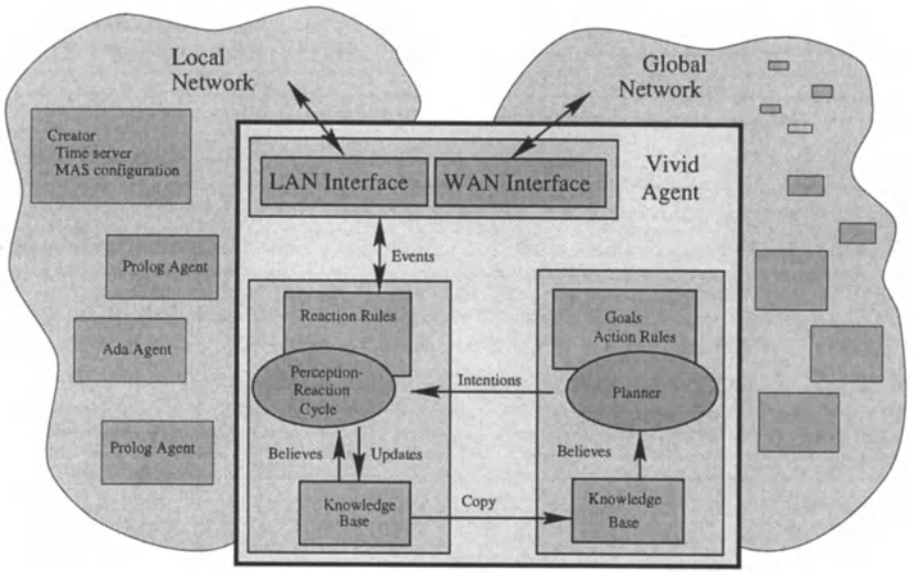


Figure 5.3. The CAP architecture of a vivid agent.

5.3.1 An Architecture for Concurrent Action and Planning

We call the architecture which is depicted in Figure 5.3 CAP for Concurrent Action (subsuming both reactions and planned actions) and Planning. The reactive component receives incoming messages representing perception and communication events and generates a reaction based on the set of reaction rules. The reactions depend on the agent's knowledge base and may update it. The planner runs concurrently with the perception-reaction-cycle on a copy of the knowledge base. Therefore, plans depend on the current state of the knowledge base but do not change it. Once the planner has generated a plan it communicates it to the action component, where plan execution is interleaved with reaction.

Formally, we can capture the CAP architecture by a tuple (A,P) of an action component A and a planning component P:

Definition 5.14 *Agent State*
Let X and X' be knowledge bases, EQ an event queue, PQ a plan queue, GQ a goal queue and $Flag \in \{r,p\}$ a flag. Then $A = (X,EQ,PQ,Flag)$ is called action state, *$P = (X',GQ)$* planning state *and $\mathcal{A} = (A,P)$* agent state.

The different components of the agent state need some explanation. First of all, there are two knowledge bases: The action state contains an up-to-date knowledge base X and the planning state a knowledge base X' which is copied from X once in a while. The knowledge base X is subject to rapid change so that it would be inappropriate for planning. Thus, plans are generated with reference to a fixed copy of X. Of course, we expect that the changes of X do normally not affect the quality of plans based on the older X'. Thus the plan is executed with respect to X. In case any planned action is no more applicable, replanning is initiated.

There are three queues, namely event, plan, and goal queue. The event queue represents the agent's connection to the environment. It stores incoming messages from the agent's perception subsystems and from other agents. The stored events are consumed one by one triggering appropriate reactions. The plan queue connects the planner and action component. The planner generates plans and communicates them to the action component which is in lieu of acting. The plans in the queue are executed step by step. If a plan's next action is not applicable anymore the plan is removed from the queue and the failure is communicated to the planner so that the goal is replanned. Finally, the goal queue is part of the action component and informs the planner of what goals are still to be achieved. Last but not least, the flag indicates for the action state whether it has to react (r) in response to events or whether a planned action is to be executed next (p).

5.3.2 *Transition System Semantics*

We can describe the temporal behaviour of an agent by means of transitions between agent states. Five kinds of transitions are performed: perception, reaction, plan execution, replanning, and planning.

Definition 5.15 *Transition Semantics*
Let $\mathcal{A} = (A,P) = ((X,EQ,PQ,Flag),(X',GQ))$ *be an agent state,* ε *an event,* π *an action,* RR_ε *a function that updates a knowledge base* X *with all effects of reaction rules in* RR *which are triggered by* ε *and whose conditions hold in* X*. An empty queue is represented by* $[]$*, a queue with head* q *and tail* Q *by* $q : Q$*, and adding an element* q *to a queue* Q *by* $Q + q$*. Then*

1. *Perception*
   $((X,EQ,PQ,Flag),P) \xrightarrow{\varepsilon} ((X,EQ+\varepsilon,PQ,Flag),P)$
2. *Reaction*
   $((X,\varepsilon : EQ,PQ,r),P) \xrightarrow{RR_\varepsilon} ((RR_\varepsilon(X),EQ,PQ,p),P)$
   $((X,[],PQ,r),P) \longrightarrow ((X,[],PQ,p),P)$
3. *Plan Execution*
   $((X,EQ,(G,\pi : \Pi) : PQ,p),P) \xrightarrow{\pi+} ((\pi(X),EQ,(G,\Pi) : PQ,r),P)$

if $\pi : Eff \leftarrow Cond$ *and* $X \vdash Cond$
$((X, EQ, [], p), P) \longrightarrow (X, EQ, [], r), P)$

4. *Replanning*
$((X, EQ, (G, \pi : \Pi) : PQ, p), (X', GQ)) \xrightarrow{\pi -} ((X, EQ, PQ, r), (X, GQ + G))$ *if* $\pi :$ $Eff \leftarrow Cond$ *and* $X \nvdash Cond$

5. *Planning*
$((X, EQ, PQ, F), (X', G : GQ)) \xrightarrow{planG} ((X, EQ, PQ + (G, \Pi), F), (X', GQ))$ *if* $\Pi(X') \vdash G$

Perception is realised by adding an event at the end of the event queue (1). For the reaction transition (2) the flag must be set for reaction and in case there is an event ε, it is removed from the event queue and the knowledge base is updated by RR_ε according to the reaction rules. With no event present, the flag is switched to plan execution. For plan execution (3) it is checked whether the next planned action π is executable. In case it is, it is removed from the plan queue and the flag is set for reaction. If there is no action to be executed, the flag is directly switched for reaction. Replanning (4) applies if a planned action is not applicable anymore, then the plan is removed from the plan queue, the goal is added to the goal queue and the planner's outdated knowledge base X' is replaced by the up-to-date X. An alternative to removing the full plan from the plan queue is to remove only the action π and initiate planning for the condition *Cond*. But since the whole plan Π is based on the planners old copy of the knowledge base it may be the case that Π is not very useful anymore. Finally, the planning transition (5) is independent from the action state F. The planner generates a plan for a goal and communicates it to the reaction state.

The above definition has several advantages over previous approaches: Most important, (re)acting and planning are completely separated tasks. In many other approaches control is distributed among the reactive and the planning component in a fixed scheme which gives rise to the problem of guaranteeing effective response times by means of bounded (or real-time) rationality. In our CAP architecture (re)actions and planning are performed concurrently and can communicate asynchronously through the plan and goal queues. This may solve the problem of bounded rationality since we can tune the responsiveness of agents by assigning different priorities to these tasks.

To implement the CAP architecture we use PVM-Prolog.

5.3.3 PVM-Prolog

The parallel virtual machine [Gea94], short PVM, is a standard software in distributed and parallel computing that permits a heterogeneous collection of UNIX computers (ranging from a personal computer or workstation to massively parallel processors) to be viewed as a single parallel computer. PVM coordinates the different hardware ar-

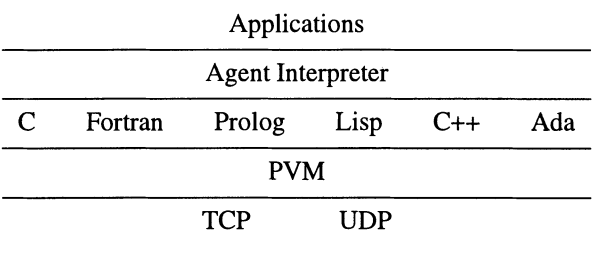


Figure 5.4. Programming paradigm layers.

chitectures and data formats within a network. The PVM computing model is simple, yet very general: tasks are spawned in the network accessing PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronisation of the tasks.

Besides the standard interfaces to C, C++ and Fortran, PVM is available in Lisp, Ada and Prolog. Its widespread use and generality makes PVM an excellent candidate for implementing interpreters for multi-agent programming languages. A general scheme for such implementations contains PVM as a layer on top of low-level network protocols (see Figure 5.4). The next layer contains programming languages with full PVM access. On top of them, agent interpreters can be implemented, allowing to run high-level agent programs as the topmost layer.

PVM realises communication of spawned tasks by message-passing. The message-passing primitives are tailored for heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronisation, and global sum. There are no limits for messages: A task can send an unlimited number of messages of unlimited size, though the size is, of course, limited by the available memory of the host.

An essential feature for multi-agent applications is asynchronous communication. Tasks can perform an asynchronous blocking send and receive. A *blocking* send or receive waits until the message is sent or an incoming message arrives. Blocking is often useful to let the agent idle and safe computational power. Besides the primitive send, a multi-cast to a set of tasks and broadcast to user-defined groups is supported. In general, the message order is preserved in point-to-point communication. Additionally, PVM provides dynamic hardware configuration, i.e. addition and deletion of hosts at run-time and signaling, such that on events such as exit, addition or deletion of tasks other tasks are informed. All these PVM-primitives are included in PVM-Prolog, a Prolog-core extended by a PVM-interface [CM97] (see Figure 5.5). In addition to PVM's coarse grain parallelism, PVM-Prolog provides process-internal

Process Control	*pvm_mytid*(−*TID*) *pvm_spawn*(+*LogicProgram*, +*Goal*, +*OptList*, ...) *pvm_exit*
Dynamic Configuration	*pvm_addhosts*(+*HostList*, −*StatusList*) *pvm_delhosts*(+*HostList*, −*StatusList*)
Communication	*pvm_send*(+*TID*, +*Tag*, +*Term*) *pvm_mcast*(+*TIDList*, +*Tag*, +*Term*) *pvm_*[*n*]*recv*(+*TID*, +*Tag*, −*Term*)

Figure 5.5. PVM predicates.

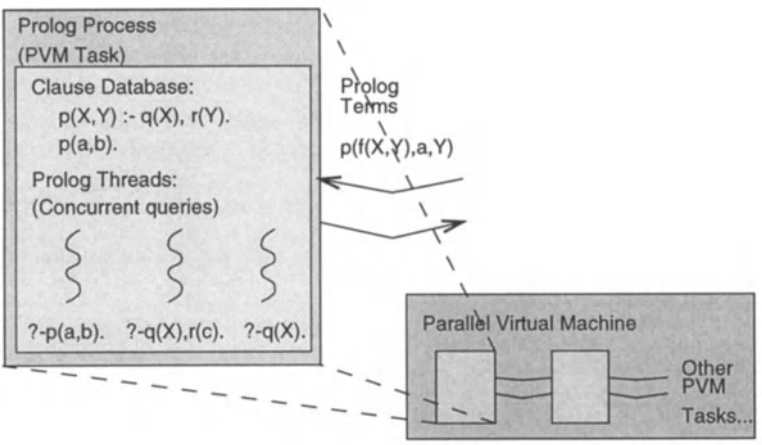


Figure 5.6. PVM-Prolog programming model.

threads. While PVM tasks run physically distributed the threads are process internal and therefore have less overhead (see Figure 5.6).

Threads allow concurrent execution of Prolog goals. They are independent of each other and must not share variables. To control threads, PVM-Prolog offers primitives to create, kill, and identify them. A second class of commands deals with communication of threads through queues. Queues may be created, killed, identified, and can be accessed by threads through put and get primitives. The scheduling of threads is done in a pre-emptive, round-robin scheme and can be manipulated with regard to their priority and activity. High-priority threads are executed before low-priority ones.

Thread Control	*t_create*(+*Goal*, +*Stacks*, +*Priority*, +*Activity*, −*Thread_id*) *t_mytid*(−*Thread_id*) *t_kill*(+*Thread_id*)
Term Queues	*q_create*(+*QName*, +*MaxTerms*, +*MaxMem*, −*QId*) *q_name*(?*QName*, ?*QId*) *q_*[*n*]*put*[*p*](+*QId*, +*Term*) *q_*[*n*]*get*[*p*](+*QId*, −*Term*) *q_*[*n*]*select*[*p*](+*QId*, −*Term*, −*Queue*) *q_destroy*(+*QId*)
Scheduling	*t_activity*(+*Thread_id*, ?*Thread_Activity*) *t_priority*(+*Thread_id*, ?*Thread_Priority*)

Figure 5.7. Multi-threading predicates.

A thread's activity defines the number of commands which is executed per time slice. The threads commands are summarised in Figure 5.7

5.3.4 Implementation

The CAP architecture is easily implementable in PVM-Prolog. In fact, it was motivated by practical experience. Given a multi-agent system represented as a tuple $(\mathcal{A}_1, \ldots \mathcal{A}_n)$ of agents all $\mathcal{A}_i$ run as parallel processes on distinct machines. Besides this coarse-grain parallelism, a particular agent $\mathcal{A} = (A, P)$ installed on a single machine invokes two concurrent threads to run A and P. In our implementation, the MAS is initially set up by configuring the network and spawning the agents in the network using the primitive *pvm_spawn* (see the initial process in Figure 5.8). To set up a particular agent, the primitive *t_create* forks a thread for the planner (currently we use STRIPS [FN71]) to run concurrently to the perception-reaction-cycle. The perception-reaction-cycle pops events from the event queue by calling *pvm_recv*. Subsequently, it generates a reaction in response to the event. The reaction comprises besides assimilating epistemic effects, performing actions which are available as meta predicates and sending messages which are translated to *pvm_send*. The knowledge base is an extended logic program [AP96] being easily implementable in Prolog.

The perception-reaction-cycle is implemented as follows: First, we check if there is a message (*newEvt*, see Figure 5.9). If there is one, the predicate *pvm_nrecv*, which realises a non-blocking receive, succeeds and the message is popped from the event queue. Next, the reaction rules are evaluated. For all reaction rules matching the event expression the precondition *Cond* is tested and in case it holds, the triggered actions

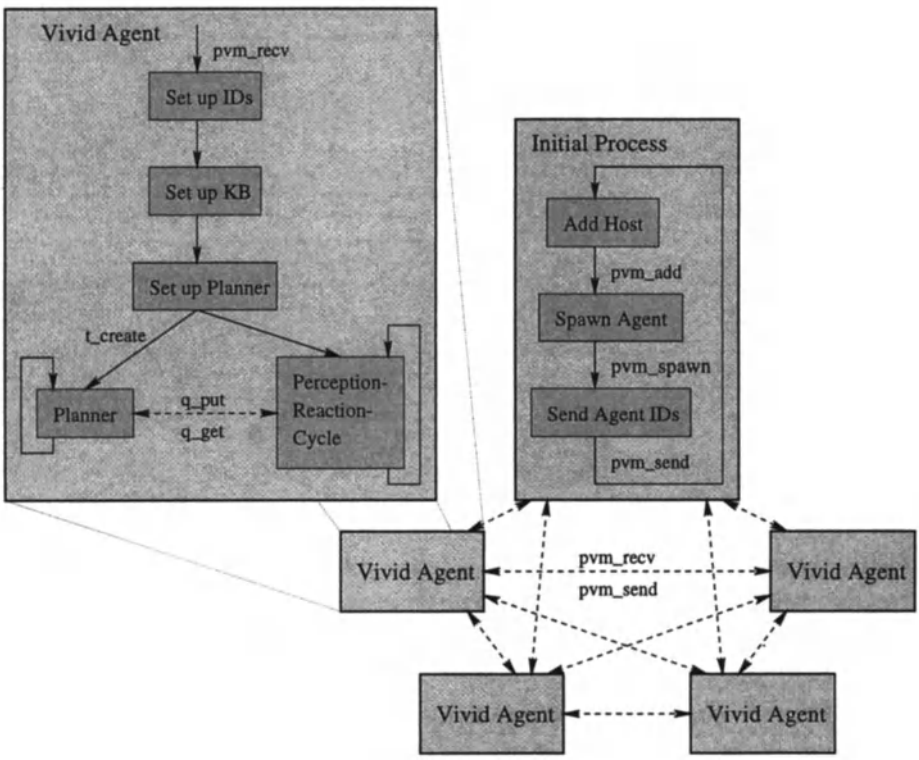


Figure 5.8. Design of the implementation.

```
cycle :-
   newEvt(Evt),
   findall(ActEff,(reaction(ActEff,Evt,Cond),demo(Cond)),ActEffs),
   perform(ActEffs),
   cycle.

newEvt(Msg/From) :-pvm_nrecv(-1,1,Msg/From).
newEvt(noEvt).
```

Figure 5.9. Perception-Reaction-Cycle.

```
perform([noAct/Effs|ActEffs]) : –
    assimilate(Effs),perform(ActEffs).
perform([sendMsg(Msg,To)/Effs|ActEffs]) : –
    name2id(To,Id),i_am(Self),
    pvm_send(Id,1,Msg/Self),
    assimilate(Effs),perform(ActEffs).
perform([do(Act)/Effs|ActEffs]) : –
    call(Act),
    assimilate(Effs),perform(ActEffs).
perform([halt/_|ActEffs]) : –
    perform(ActEffs),
    pvm_exit,fail.
assimilate([not L|Effs]) : –
    retractall(fact(L)),
    assimilate(Effs).
assimilate([L|Effs]) : –
    assert(fact(L)),
    assimilate(Effs).
```

Figure 5.10. Executing actions and reactions.

are collected in the set *ActEffs*. Subsequently, they are performed and the cycle is closed. Triggered actions are performed as follows (see Figure 5.10): The sendMsg-action is translated into PVM-Prolog's *pvm_send*, whereas a physical action do(*Act*) causes a call to a procedure with the same name. The epistemic effects of actions are assimilated into the knowledge base by means of the Prolog operations *assert* and *retract*.

5.3.5 Experiments

Applications of purely reactive vivid agents include distributed diagnosis [SdAMP97, SW97, FdAMNS97]. However, to reflect the deliberative and pro-active capacities we illustrate the above implementation and concepts by two examples. The first uses a simple blocks world example to demonstrate basic behaviour specification, and the second uses the loading dock to show replanning.

Blocks World. An agent a sends a task to agent b and monitors b's progress until b finished its task. In this case b is supposed to plan and execute moving actions in the blocks world as depicted in Figure 5.11. The agents a and b are specified as follows:

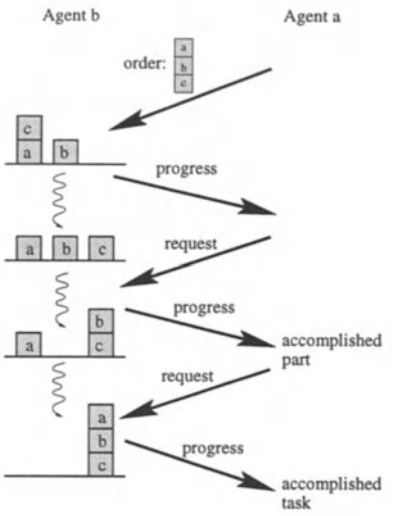


Figure 5.11. Agent a gives agent b a planning task and observes $b's$ progress.

- a sends b on receipt of an initial message by the creator process, the command to plan to achieve that block a is on top of b which is on top of c.
- On receipt of the initial message by the creator, a also sends a request for information about b's progress.
- a records the progress of b and continues monitoring b until b has accomplished the whole task.
- If agent b receives a command from a, it performs it. In particular, it serves a's requests for information on the state of affairs.

Formally, we can translate the above description into the action and reaction rules and knowledge bases shown in Figure 5.12 and Figure 5.13.

If executed on the PVM platform the above specification leads to a trace as depicted in Figure 5.14. An entry $x \longleftarrow y\ z$ means that x received from y the message z. For the sake of brevity we left out send messages (note that communication is asynchronous) and the creator's start and halt messages.

Grid World. Consider the fork lift agents of the loading dock again (see Example 5.6). Assume that a_1 receives initially the order to move to position $(3,3)$ and a_2 is sent to position $(2,3)$. Both plan their ways, but cannot foresee that their ways cross.

Reaction Rules

$\mathsf{sendMsg}(\mathsf{do}(plan\_goal(on(a,b)\&on(b,c))),b) \leftarrow$
  $\mathsf{recvMsg}(tell(start),creator).$

$\mathsf{sendMsg}(request(on),b) \leftarrow$
  $\mathsf{recvMsg}(tell(start),creator).$

$\mathsf{sendMsg}(request(on),b) \leftarrow$
  $\mathsf{recvMsg}(reply(on),b),$
  $not\ accomplished.$

$accomplished\_part_1 \leftarrow$
  $\mathsf{recvMsg}(reply(on(b,c),T),b),$
  $not\ accomplished\_part_1.$

$accomplished\_part_2 \leftarrow$
  $\mathsf{recvMsg}(reply(on(a,b),T),b),$
  $not\ accomplished\_part_2.$

Action Rules

$\emptyset$

Initial Knowledge Base

$accomplished \leftarrow$
  $accomplished\_part_1,$
  $accomplished\_part_2.$

Figure 5.12. Specification of agent a.

Reaction Rules

$\mathsf{do}(A) \leftarrow$
  $\mathsf{recvMsg}(\mathsf{do}(A),a).$

$\mathsf{sendMsg}(reply(on(A,B),T),a) \leftarrow$
  $\mathsf{recvMsg}(request(on),a),$
  $on(A,B),time(T).$

$\mathsf{sendMsg}(reply(on),a) \leftarrow$
  $\mathsf{recvMsg}(request(on),a).$

Initial Knowledge Base

$on(a,floor).$
$on(b,floor).$
$on(c,a).$
$clear(b).$
$clear(c).$

Action Rules

$move(U,V,floor) : not\ on(U,\_), on(U,floor), clear(V) \leftarrow$
  $on(U,V), V \neq floor, clear(U).$

$move(U,V,W) : not\ on(U,\_), not\ clear(W), on(U,W), clear(V) \leftarrow$
  $W \neq floor, clear(W), on(U,V), U \neq W, clear(U).$

Figure 5.13. Specification of agent b.

b	⟵ a	do(*plan_goal*(*on*(*a*,*b*)&*on*(*b*,*c*)))
b		did *plan_goal*(*on*(*a*,*b*)&*on*(*b*,*c*))
b	⟵ a	*request*(*on*)
a	⟵ b	*reply*(*on*(*a*,*floor*),*9109812*)
a	⟵ b	*reply*(*on*(*b*,*floor*),*9109812*)
a	⟵ b	*reply*(*on*(*c*,*a*),*9109812*)
a	⟵ b	*reply*(*on*)
...		
b	⟵ a	*request*(*on*)
a	⟵ b	*reply*(*on*(*a*,*floor*),*9109815*)
a	⟵ b	*reply*(*on*(*b*,*floor*),*9109815*)
a	⟵ b	*reply*(*on*(*c*,*a*),*9109815*)
a	⟵ b	*reply*(*on*)
b		executes *start*
b	⟵ a	*request*(*on*)
a	⟵ b	*reply*(*on*(*a*,*floor*),*9109815*)
a	⟵ b	*reply*(*on*(*b*,*floor*),*9109815*)
a	⟵ b	*reply*(*on*(*c*,*a*),*9109815*)
b		executes *move*(*c*,*a*,*floor*)
b		assimilates *not on*(*c*,*a*)
b		assimilates *on*(*c*,*floor*)
b		assimilates *clear*(*a*)
a	⟵ b	*reply*(*on*)
b	⟵ a	*request*(*on*)
a	⟵ b	*reply*(*on*(*a*,*floor*),*9109815*)
a	⟵ b	*reply*(*on*(*b*,*floor*),*9109815*)
a	⟵ b	*reply*(*on*(*c*,*floor*),*9109815*)
b		executes *move*(*b*,*floor*,*c*)
a	⟵ b	*reply*(*on*)
b		assimilates *not on*(*b*,*floor*)
b		assimilates *not clear*(*c*)
b		assimilates *on*(*b*,*c*)
b		assimilates *clear*(*floor*)
b	⟵ a	*request*(*on*)
a	⟵ b	*reply*(*on*(*a*,*floor*),*9109816*)
a	⟵ b	*reply*(*on*(*c*,*floor*),*9109816*)
a	⟵ b	*reply*(*on*(*b*,*c*),*9109816*)
a		assimilates *accompl'd_part*$_1$
b		executes *move*(*a*,*floor*,*b*)
b		assimilates *not on*(*a*,*floor*)
b		assimilates *not clear*(*b*)
b		assimilates *on*(*a*,*b*)
b		assimilates *clear*(*floor*)
a	⟵ b	*reply*(*on*)
b	⟵ a	*request*(*on*)
a	⟵ b	*reply*(*on*(*c*,*floor*),*9109816*)
a	⟵ b	*reply*(*on*(*b*,*c*),*9109816*)
a	⟵ b	*reply*(*on*(*a*,*b*),*9109816*)
a		assimilates *accompl'd_part*$_2$
a	⟵ b	*reply*(*on*)

Figure 5.14. A trace for the blocks worlds.

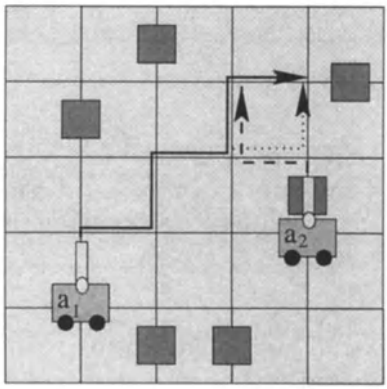


Figure 5.15. a_1's initial plan (bold arc), a_2's plan (dashed arc), a_1's revised plan (dotted arc).

Reaction Rules

$do(plan\_goal([at(3,3)])) \leftarrow$
  $recvMsg(tell(start), creator)$.
$obstacle(X,Y) \leftarrow$
  $recvMsg(inform(at(X,Y)), a_2)$,
  $i\_am(a_1)$.
$not\ obstacle(X,Y) \leftarrow$
  $recvMsg(inform(not\ at(X,Y)), a_2)$,
  $i\_am(a_1)$.

Initial Knowledge Base

$at(0,0)$.

Action Rules

$walk\_north : not\ at(X,,Y), at(X,Y')$
$\leftarrow at(X,Y), Y'\ is\ Y+1,$
  $not\ obstacle(X,Y')$.
$walk\_east : not\ at(X,,Y), at(X',Y)$
$\leftarrow at(X,Y), X'\ is\ X+1,$
  $not\ obstacle(X',Y)$.
$walk\_south : not\ at(X,,Y), at(X,Y')$
$\leftarrow at(X,Y), Y'\ is\ Y-1,$
  $not\ obstacle(X,Y')$.
$walk\_west : not\ at(X,,Y), at(X',Y)$
$\leftarrow at(X,Y), X'\ is\ X-1,$
  $not\ obstacle(X',Y)$.

Figure 5.16. Specification of a forklift agent.

Since a_2 is nearer to its destiny it can move there. Agent a_1 has to replan the goal of going to $(3,3)$ when it realises at position $(2,2)$ that agent a_2 blocks its planned way. The behaviour of agent a_1 is described in Figure 5.16. Initially, a_1 knows that it is at position $(0,0)$. The first reaction rule states that on receipt of a start message a_1 pushes the goal to go to $(3,3)$ on top of the goal queue. If a_1 receives information about a_2's position it updates its knowledge base accordingly. If executed the specification leads to a trace as shown in Figure 5.17.

Next we develop two applications of distributed diagnosis. The first implements fault-tolerant diagnosis, the second diagnoses an unreliable communication protocol.

5.4 DISTRIBUTED DIAGNOSIS OF A COMPUTER NETWORK

To achieve fault-tolerant diagnosis three agents a,b,c diagnose a node d. Since a,b,c have the same description of d their findings are redundant in case they are up and they received the same test results of d. The tasks of the agents can be divided into four steps as shown in Figure 5.18.

Initially, the agents receive a *start* message from the creator process. In a second step, the diagnosing agents send requests for test results to the group of agents whose underlying hardware is examined, the tests are executed and the requests are answered. Thirdly, the diagnosing agents compute diagnoses based on the new test results they received and communicate their diagnoses among the diagnosis group. Finally, each diagnosis agent computes a consensus based on all the diagnoses it received, and sends it to the (possibly human) agent it has to report to.

$a_2 \longleftarrow$ creator : *tell*(*start*)
a_2 did *plan_goal*([*at*(2,3)])
$a_1 \longleftarrow$ creator : *tell*(*start*)
a_1 did *plan_goal*([*at*(3,3)])
a_2 executes *walk_north_to*(3,2)
a_2 assimilates *notat*(3,1)
a_2 assimilates *at*(3,2)
a_2 executes *walk_west_to*(2,2)
a_2 assimilates *notat*(3,2)
a_2 assimilates *at*(2,2)
$a_1 \longleftarrow a_2$:*inform*(*not*(*at*(3,1)))
a_1 assimilates *notobstacle*(3,1)
a_2 executes *walk_north_to*(2,3)
a_2 assimilates *notat*(2,2)
a_2 assimilates *at*(2,3)
$a_1 \longleftarrow a_2$:*inform*(*at*(3,2))
a_1 assimilates *obstacle*(3,2)
$a_1 \longleftarrow a_2$:*inform*(*not*(*at*(3,2)))
a_1 assimilates *notobstacle*(3,2)
$a_1 \longleftarrow a_2$:*inform*(*at*(2,2))
a_1 assimilates *obstacle*(2,2)
$a_1 \longleftarrow a_2$:*inform*(*not*(*at*(2,2)))
a_1 assimilates *notobstacle*(2,2)
$a_1 \longleftarrow a_2$:*inform*(*at*(2,3))
a_1 assimilates *obstacle*(2,3)
a_1 executes *walk_north_to*(0,1)
a_1 assimilates *notat*(0,0)
a_1 assimilates *at*(0,1)
a_1 executes *walk_east_to*(1,1)
a_1 assimilates *notat*(0,1)
a_1 assimilates *at*(1,1)
a_1 executes *walk_north_to*(1,2)
a_1 assimilates *notat*(1,1)
a_1 assimilates *at*(1,2)
a_1 executes *walk_east_to*(2,2)
a_1 assimilates *notat*(1,2)
a_1 assimilates *at*(2,2)
a_1 has to replan since *walk_north_to*(2,3) failed
a_1 pushed [*at*(3,3)] on goal queue
a_1 executes *walk_east_to*(3,2)
a_1 assimilates *notat*(2,2)
a_1 assimilates *at*(3,2)
a_1 assimilates *walk_north_to*(3,3)
a_1 assimilates *notat*(3,2)
a_1 assimilates *at*(3,3)

Figure 5.17. Trace for the two forklift agents.

5.4.1 Specification of the Diagnosis Agents

This agent behaviour is specified declaratively via the following reaction rules based on the communication events

$$CEvt = \{ \; ask(Query), reply(Answer), \\ tell(Input), request(Action) \}$$

When the creator process tells a diagnoser to start, the diagnoser asks the test agents for a list of all current test results:

$$\mathsf{sendMsg}(ask(testResults), T) \longleftarrow \\ \mathsf{recvMsg}(tell(start), creator), \\ i\_am(A), diagnoser(A), tester(T).$$

When a tester receives a query for test results it performs the tests and replies to the query. To obtain test results low-level system calls can be integrated into the predicate *perform_tests* by procedural attachment.

$$\mathsf{sendMsg}(reply(test(Results)), D) \longleftarrow \\ \mathsf{recvMsg}(ask(testResults), D), \\ perform\_test(Results).$$

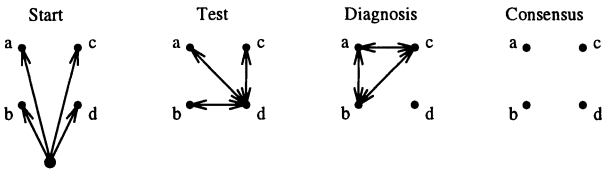


Figure 5.18. Steps of the diagnosis process.

If an agent receives a reply with new test results, the agent enters the third phase of computing and exchanging diagnoses among the group of diagnosers. The next reaction rule expresses that an agent which received test results, computes diagnoses based on the received test data and sends the diagnoses to all diagnosers B. Similar to *perform_tests*, the predicate *compute_diags* is implemented by procedural attachment. A call of *compute_diags* adds the new data *Results* to the KB and starts a revision of KB, which may be partially inconsistent due to the newly received test data and the agent's default predictions concerning the correctness of involved hardware. The repair of the partial inconsistency yields diagnoses which are returned in the parameter *Ds*.

$$\begin{aligned}\mathsf{sendMsg}(&\mathit{tell}(\mathit{diagnoses}(Ds)),A) \longleftarrow\\ &\mathsf{recvMsg}(\mathit{reply}(\mathit{test}(Results)),T),\\ &\mathit{compute\_diags}(Ds,Results),\\ &\mathit{diagnoser}(A).\end{aligned}$$

Incoming diagnosis results are added to the knowledge base:

$$\begin{aligned}&\mathit{diagnoses}(A,Ds) \longleftarrow\\ &\quad\mathsf{recvMsg}(\mathit{tell}(\mathit{diagnoses}(Ds)),A).\end{aligned}$$

The incoming diagnosis results are counted and if the last message is received the agent triggers the final phase of consensus computation. The pending diagnosis results are counted by checking if there is a diagnoser for which no diagnosis results are stored. The last result is received if the agent is still waiting for at least one result, but not for two anymore. This is expressed by the following deduction rules:

$$\begin{aligned}\mathit{wait\_for}(A) &\leftarrow \mathit{diagnoser}(A), \mathit{not}\ \mathit{diagnoses}(A,\_).\\ \mathit{wait\_for\_one} &\leftarrow \mathit{wait\_for}(A).\\ \mathit{wait\_for\_two} &\leftarrow \mathit{wait\_for}(A),\\ &\qquad \mathit{wait\_for}(B), A \neq B.\\ \mathit{last} &\leftarrow \mathit{wait\_for\_one}, \mathit{not}\ \mathit{wait\_for\_two}.\end{aligned}$$

In case a diagnosis result is received and it is the last one, the agent triggers its voting procedure (by sending a corresponding request to itself):

$$\begin{aligned}&\mathsf{sendMsg}(\mathit{request}(\mathit{vote}),B) \longleftarrow\\ &\quad \mathsf{recvMsg}(\mathit{tell}(\mathit{diagnoses}(Ds)),A),\\ &\quad \mathit{i\_am}(B),\mathit{last}.\end{aligned}$$

The final step of computing a consensus does not involve any communication with other agents. On receipt of a request to vote, the agent computes a consensus by calling *vote*, outputs the result and as an effect cleans up its knowledge base from all diagnosis results stored intermediately. The computation of the consensus is procedurally attached. For the purpose of fault tolerance a consensus can be implemented by a simple voting mechanism, i.e. the call *vote* checks all the collected *diagnoses* facts and determines the most frequent result.

$$\begin{aligned}&\mathit{not}\ \mathit{diagnoses}(A,Ds) \longleftarrow\\ &\quad \mathsf{recvMsg}(\mathit{request}(\mathit{vote}),A),\\ &\quad \mathit{vote}(C),\mathit{write}(C),\mathit{diagnoses}(A,Ds).\end{aligned}$$

5.4.2 Execution of the Agent Specification

When executed, the above agent specification invoked for four agents a,b,c,d leads to a trace as shown in Figure 5.19. The agents a,b,c form the diagnosing group which takes care of a tester group with a single member d. As we want to focus on the communication among the agents we have d always sending the same test results. When the agents perform the four steps of starting, testing, diagnosing and computing a consensus, as sketched in Figure 5.18, they are not synchronised and the steps may overlap. Therefore we represent *send* and *receive* by a bold and dotted arc, respectively. For example, the first icon (1) indicates that b received from the creator a start message and the second (2) means that b sends to d a request message. The receipt of this request message (4) is preceded by d receiving a start message from the creator (3). In general, the trace reveals that agent b is much faster than a and c. For example, after c receives the start message (10) agent b already received test results, computed its diagnoses and starts to broadcast them (11,13,16).

5.5 DIAGNOSIS OF A COMMUNICATION PROTOCOL

Consider the diagnosis of a communication protocol described in section 3.2.5. The whole network being too complex we break the problem down and assign agents to areas of feasible size. The agents know all necessary details about the area and view the rest of network as an abstraction. In section 3.2.5 we have modelled the knowledge base of the agents that allows them to perform their local diagnosis tasks, in this section we further develop the interaction and cooperation with other agents. The resulting algorithm has an attractive complexity compared to a centralised solution, both concerning communication overhead and computational complexity.

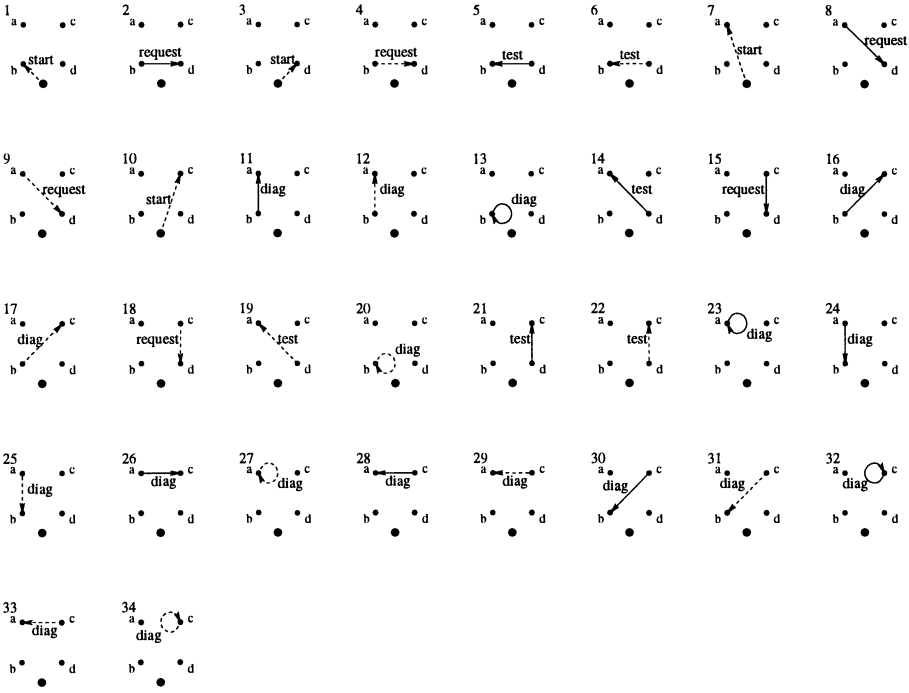


Figure 5.19. A trace of communication of agents a, b, c diagnosing d. The big node is the creator process.

5.5.1 Specification of the Diagnosis Agents

An agent's local diagnosis can produce three different results. The agent may find that there is no diagnosis to explain the misbehaviour, it may find that itself is faulty, or it may find another agent responsible. In the second case no cooperation is initiated since we assume that an agent knows its own domain well enough to believe in its own findings. In the two other cases a cooperation is started. To encode the reaction rules for the interaction we need two meta-predicates.

1. There is no diagnosis to explain the observation (*no_diags*).
2. There are diagnoses which do not involve the agent itself (*next*). In this case the agent abduces a new, refined observation.

With the two meta predicates *no_diags*/1 and *next*/2 we encode the agents' reaction rules:

If an agent receives an observation and has no explanation for it, the fault must be intermittent, since neither the agent itself is faulty nor are there any neighbours to accuse. This is reported to the requesting agent:

$$\begin{aligned}&\mathsf{sendMsg}(\mathit{intermittent\_failure}(B),A) \leftarrow\\ &\quad \mathsf{recvMsg}(\mathit{message\_lost}(N,C),A),\\ &\quad \mathit{no\_diags}(\mathit{message\_lost}(N,C)),\mathit{i\_am}(B).\end{aligned}$$

If an agent receives an observation and is itself the cause of the problems it reports this fact back to the requesting agent:

$$\begin{aligned}&\mathsf{sendMsg}(\mathit{responsible}(B),A) \leftarrow\\ &\quad \mathsf{recvMsg}(\mathit{message\_lost}(N,C),A),\\ &\quad \mathit{i\_am}(B),\mathit{obs}(\mathit{down},B).\end{aligned}$$

If the agents area is not abnormal and there are diagnoses suspecting the agents neighbours, the newly abduced observation is sent to the suspected neighbour:

$$\begin{aligned}&\mathsf{sendMsg}(\mathit{message\_lost}(M,C),M) \leftarrow\\ &\quad \mathsf{recvMsg}(\mathit{message\_lost}(N,C),A),\\ &\quad \mathit{i\_am}(B),\mathit{not\ obs}(\mathit{down},B),\\ &\quad \mathit{next}(M,\mathit{message\_lost}(N,C)).\end{aligned}$$

In this case the agent has to remember to forward the final diagnosis result to the requesting agent:

$$\begin{aligned}&\mathit{remember\_to\_reply\_to}(A) \leftarrow\\ &\quad \mathsf{recvMsg}(\mathit{message\_lost}(N,C),A),\\ &\quad N \neq A,\mathit{i\_am}(B),\mathit{not\ obs}(\mathit{down},B),\\ &\quad \mathit{not\ no\_diags}(\mathit{message\_lost}(N,C)).\end{aligned}$$

If an agent receives a diagnosis result from one of its neighbours and has to report the result to another neighbour, it forwards it:

$$\begin{aligned}&\mathsf{sendMsg}(\mathit{intermittent\_failure}(A),C) \leftarrow\\ &\quad \mathsf{recvMsg}(\mathit{intermittent\_failure}(A),B),\\ &\quad \mathit{remember\_to\_reply\_to}(C).\\ &\mathsf{sendMsg}(\mathit{responsible}(A),C) \leftarrow\\ &\quad \mathsf{recvMsg}(\mathit{responsible}(A),B),\\ &\quad \mathit{remember\_to\_reply\_to}(C).\end{aligned}$$

After forwarding a diagnosis result, the bookmark to reply is removed from the agent's knowledge base:

$$
\begin{aligned}
&\textit{not remember\_to\_reply\_to}(C) \leftarrow \\
&\quad \mathsf{recvMsg}(\textit{intermittent\_failure}(A), B), \\
&\quad \textit{remember\_to\_reply\_to}(C). \\
&\textit{not remember\_to\_reply\_to}(C) \leftarrow \\
&\quad \mathsf{recvMsg}(\textit{responsible}(A), B), \\
&\quad \textit{remember\_to\_reply\_to}(C).
\end{aligned}
$$

5.5.2 *Execution of the Agent Specification*

Consider Figure 1.5 and assume that node n_1 sends a message to c_7, but the messages gets lost. Since n_1 does not receive an acknowledgment, a timeout mechanism informs n_1 that the message is lost and the diagnosis process starts. Consider two scenarios: In the first n_3 looses the message, in the second an intermittent failure occures.

Initially, the creator process sends a start message to all nodes (see Figure 5.20, lines 1,2,3,4,9,14). The timeout mechanism informs n_1 of the lost message (5,6). Node n_1 knows that it is working fine and suspects the neighbour in charge of sending messages to c_7, namely n_2. Subsequently n_1 sends the refined observation that the message is lost from n_2 to c_7 to n_2 (8,10). Similarly n_2 informs n_3 (11,12,15). Additionally it remembers that it has to report the final result to n_1 (13). Finally, n_3 turns out to be the cause of the fault and the result is sent from n_3 to n_2 (16,17) and from n_2 to n_1 (18,19). n_2 removes the fact that it has to respond to n_1 (20).

In the second trace (see Figure 5.21) all nodes are ok at diagnosis time so the fault is intermittent. The initial phase is similar to the first trace. Only when n_3 comes up with no diagnoses (16), message of an intermittent failure is sent back.

5.6 SUMMARY

We have introduced and implemented the concept of vivid agents and evaluated it in the domain of distributed diagnosis. The work presented improves several other approaches to agent-oriented programming.

First, the agents' reactive and pro-active behaviour can be implemented by using PVM-Prolog's thread concept to run a perception-reaction-cycle concurrently to a planning facility. Such light-weight concurrency has the advantage of fast communication between strongly connected parts of a single agent. Second, the formal semantics of the CAP architecture helps understanding and allows program verification. The formal semantics is independent of PVM-Prolog as implementation language, but it was influenced by the concepts provided by PVM-Prolog and is therefore easily implementable in PVM-Prolog. Third, the idea of executable specifications realised in Prolog is lifted by the vivid agent concept and its implementation in PVM-Prolog to a higher and more declarative level of multi-agent specifications. Forth, PVM is hardware independent and available for a range of architectures including PCs, Work-

1	n_2	$\longleftarrow$	*creator*	*tell*(*start*)
2	n_4	$\longleftarrow$	*creator*	*tell*(*start*)
3	n_1	$\longleftarrow$	*creator*	*tell*(*start*)
4	n_5	$\longleftarrow$	*creator*	*tell*(*start*)
5	n_1	$\longrightarrow$	n_1	*message_lost*(n_1,c_7)
6	n_1	$\longleftarrow$	n_1	*message_lost*(n_1,c_7)
7	n_1			*diag*[*ab*(n_2),*new_message_lost*(n_2,c_7)]
8	n_1	$\longrightarrow$	n_2	*message_lost*(n_2,c_7)
9	n_6	$\longleftarrow$	*creator*	*tell*(*start*)
10	n_2	$\longleftarrow$	n_1	*message_lost*(n_2,c_7)
11	n_2			*diag*[*ab*(n_3),*new_message_lost*(n_3,c_7)]
12	n_2	$\longrightarrow$	n_3	*message_lost*(n_3,c_7)
13	n_2			assimilates *remember_to_reply_to*(n_1)
14	n_3	$\longleftarrow$	*creator*	*tell*(*start*)
15	n_3	$\longleftarrow$	n_2	*message_lost*(n_3,c_7)
16	n_3	$\longrightarrow$	n_2	*responsible*(n_3)
17	n_2	$\longleftarrow$	n_3	*responsible*(n_3)
18	n_2	$\longrightarrow$	n_1	*responsible*(n_3)
19	n_1	$\longleftarrow$	n_2	*responsible*(n_3)
20	n_2			assimilates *not remember_to_reply_to*(n_1)

Figure 5.20. Trace for a lost message.

stations and Multi-computer platforms. PVM's and PVM-Prolog's hardware independence also makes vivid agents flexible. Fifth, PVM has been widely used up to now, so many tools and different language interfaces are available. The PVM distribution supports C and Fortran; interfaces for C++, Ada, Lisp, Perl, Tcl, and Prolog have subsequently been developed. First results show that heterogeneous agents can run together in a net but that the MAS configuration has to be public to ensure that agents can use the proper formats to communicate. A possible solution is a public yellow pages service, which might be implemented itself in PVM-Prolog.

Using vivid agents we have shown how to solve two applications in distributed diagnosis, namely fault-tolerant diagnosis and the diagnosis of communication protocol.

1	n_2	$\longleftarrow$	*creator*	*tell*(*start*)
2	n_1	$\longleftarrow$	*creator*	*tell*(*start*)
3	n_5	$\longleftarrow$	*creator*	*tell*(*start*)
4	n_1	$\longrightarrow$	n_1	*message_lost*(n_1,c_7)
5	n_1	$\longleftarrow$	n_1	*message_lost*(n_1,c_7)
6	n_1			*diag*[*ab*(n_2),*new_message_lost*(n_2,c_7)]
7	n_1	$\longrightarrow$	n_2	*message_lost*(n_2,c_7)
8	n_2	$\longleftarrow$	n_1	*message_lost*(n_2,c_7)
9	n_6	$\longleftarrow$	*creator*	*tell*(*start*)
10	n_2			*diag*[*ab*(n_3),*new_message_lost*(n_3,c_7)]
11	n_2	$\longrightarrow$	n_3	*message_lost*(n_3,c_7)
12	n_2			assimilates *remember_to_reply_to*(n_1)
13	n_4	$\longleftarrow$	*creator*	*tell*(*start*)
14	n_3	$\longleftarrow$	*creator*	*tell*(*start*)
15	n_3	$\longleftarrow$	n_2	*message_lost*(n_3,c_7)
16	n_3			*nodiagnoses*
17	n_3	$\longrightarrow$	n_2	*intermittent_failure*(n_3)
18	n_2	$\longleftarrow$	n_3	*intermittent_failure*(n_3)
19	n_2	$\longrightarrow$	n_1	*intermittent_failure*(n_3)
20	n_1	$\longleftarrow$	n_2	*intermittent_failure*(n_3)
21	n_2			assimilates *not remember_to_reply_to*(n_1)

Figure 5.21. Trace for an intermittent failure.

6 CONCLUSIONS

This chapter rounds out the picture by comparisons to related work, by a general evaluation of the previous chapters and by pointing to future work.

6.1 COMPARISONS

6.1.1 Logic Programming and Model-based Diagnosis

Chapter 3 showed how extended logic programming is employed to solve diagnosis problems. The diagnosis engine REVISE is related to both classical diagnosis engines and non-monotonic-reasoning systems.

Model-based Diagnosis. The algorithm used in the diagnosis system REVISE corresponds roughly to the early diagnosis engines such as GDE [dKW87] or Sherlock [dKW89]. REVISE differs in that it is based on logic programming and therefore offers a rich language with two forms of negation and integrity constraints. Similar to Sherlock, REVISE supports diagnoses minimal by probability. In contrast to GDE and

M. Schroeder, *Autonomous, Model-Based Diagnosis Agents*

Sherlock, REVISE allows for abduction. REVISE' expressiveness leads to compact programs which turned out to be useful for the system description for fault management in cellular phone networks (see section 3.2.3). It has become shorter due to explicit and implicit negation and abduction than the one developed for the diagnosis engine DRUM II [FNJW97].

Unfortunately, REVISE' expressiveness leads to bad performance when dealing with real-world problems. Other diagnosis engines such as IMPLODE [RdKS93] and DRUM II [FN96a, FN96b, NF97, FN97] already deal successfully with large real-world problems such as all the ISCAS85 benchmark circuits. The difference between DRUM and REVISE is that DRUM computes updates based on a given consistent model while REVISE computes revisions. Updates allow to repair violated constraints locally, whereas program revision requires a global check of all constraints involving heavy recomputation of the programs' models. To leap forward and solve real-world problems, REVISE has to be enhanced by an efficient management of the hitting-set tree and program model to avoid recomputations in the proofs. The work done in TMSs [Doy79], ATMSs [dK86], and tabling [SSW94, SSW96] is relevant for this purpose.

Non-monotonic Reasoning. Apart from the practically motivated diagnosis engines mentioned above, there are several non-monotonic reasoning systems comparable to REVISE. PROTEIN [BF94, BFS95] is a theorem prover employing minimal model semantics and featuring negation as failure and disjunction. The system STATIC [BDP96, Prz95] additionally provides for classical negation. A non-monotonic reasoning system more similar to REVISE is smodels [NS96]. It computes well-founded semantics and stable models. It has been evaluated on a variety of examples including some examples in the domain of model-based diagnosis. All these systems do not provide integrity constraints and contradiction removal which is essential for diagnosis.

6.1.2 Strategies in Model-based Diagnosis

Diagnosis as a Process. The strategy language presented in chapter 4 is motivated by Struss' idea to view diagnosis as a process [Str92b, Str92a]. He criticises that previous approaches see diagnosis as a static problem. While in practice diagnostic reasoning is often guided by working hypotheses and simplifications which may be revised later and which give rise to non-monotonicity, most implemented systems stick to an overall inference procedure working on a global model. Furthermore, previous work does not tackle the issue of modelling and simply assumes the existence of powerful and unique models of the device and its components. But in practice several experts design various models for various purposes. The models may contradict each other and are only approximations to reality [Str92b].

Based on this criticism, Struss, Böttcher, and Dressler coined the notion of working hypotheses to guide the diagnostic process [Str92b, Str92a, BD93, BD94]. To select appropriate working hypotheses, Böttcher and Dressler develop a bunch of strategies for the diagnostic process including structural and behavioural refinement and fault-mode incompleteness [BD93, BD94]. Various authors have used other strategies and coded them into their diagnosis engines. The use of hierarchies to reduce the complexity of computing diagnoses is adopted by Davis [Dav84], Hamscher [Ham91], Genesereth [Gen84], Mozetič [Moz91]. Davis also distinguishes between functional and physical models for digital circuits [Dav84]. Preferred diagnoses are computed in [DS92, DNP94, FNS94] where preferences are the single fault assumption and physical negation [SD89], i.e. the assumption that the known fault models of the components are complete. The idea of a diagnosis process is also present in the work of De Kleer, Raiman and Shirley on measurements [dKRS91]. They view diagnosis as an incremental task involving the three phases of generating explanations, choosing actions differentiating among them, and performing these actions.

The idea of diagnosis as a process and the different instances of diagnostic strategies are the motivation for our strategy language. The very idea underlying model-based diagnosis that a device is described declaratively is lifted to the level of computing the diagnostic process. The selection of appropriate working hypotheses is expressed by rule-like strategies and can be flexibly adjusted to the problem domain. The initial proposal for a strategy language by Fröhlich, Nejdl, and Schroeder [FNS94] still distinguished between preferences and strategies. Subsequent work proved that the strategy language is capable of incorporating the preference concept as well [FNS97, FNS96]. The work presented in chapter 4 allows a unified view of the diagnostic process and commonly used strategies.

Preferences. Closely related to our notion of preferences are preferred diagnoses and revisions as defined in [DNP94, FNS94, DNPS95]. A main difference is that in our framework preferences are an integral part of strategies whereas preferences are embedded into strategies in [DNP94, FNS94, DNPS95]. The authors define a preference graph whose nodes are sets of revisables and integrity constraints [DNPS95]. The preference graph induces a partial order and the preferred diagnosis is obtained by finding the minimal node such that diagnoses exist. This concept allows for a variety of preferences such as choice of model, number of faults, preference of homogeneous diagnoses, preference of fault modes, preference of faults complete. To summarise, the preference concept is capable of expressing any property of a single diagnosis. It does, however, not allow strategies that require properties over sets of diagnoses such as structural refinement, measurements, etc. These strategies can only be expressed with additional operators such as the modalities to quantify over the successor states. Therefore the preference graph does not replace a more expressive strategy language. Taking into account the difficulty to properly design the interplay of revisables and in-

tegrity constraints as necessary in the preference graph, the strategy language turns out as an advantage since there is only the need to program in one language instead of two. Furthermore, the non-monotonicity and the special treatment of missing diagnoses of the framework fully replace the preference graph used in [DNP94, FNS94, DNPS95]. Rather than defining an additional preference concept, the notion of diagnosis should be kept as a parameter. Depending on the problem domain a suitable diagnosis engine and notion can be plugged in. The implementation of the strategy language, for example, is built on top of REVISE and can be configured to compute diagnoses minimal by set-inclusion, cardinality, and probability. The applications presented in chapter 3 show the usefulness of these notions.

Meta-Logic and Meta-Logic programming. For the design of strategies meta-programming may be an alternative approach to modal logic. Let us reconsider the way we use modal logic in our operational semantics. Starting from a given state with no working hypotheses active we construct the S5-worlds for this state. These worlds are given by the diagnosis semantics. Based on the truth values of the formulas in this world a specialised algorithm computes transitions to the following process states. By iterating this algorithm we obtain a transition system. Thus, besides providing a formal framework we use modal logic mainly for model generation. In this sense our work is more motivated by dynamic logics [Eme90] where model checking and generation are main issues than by classical modal logic.

Furthermore, modal deduction [FdCH95] is computationally very complex. Diagnosis strategies are means of speeding up diagnosis by proposing suitable assumptions. So, performing modal deduction during the diagnosis process would slow down computation. Our model generation algorithm is restricted to the given problem because it exploits the problem structure and therefore is quite efficient. A modal deduction system might still be useful here, it could be used to check if a given set of strategies is consistent, independent of a diagnostic problem. However, such a system is currently outside the scope of this work.

Another possible formalism for our framework is meta-programming. Meta-programming is suitable for implementation rather than for providing a formal background. In fact, a former version [DNPS95] of our formalisation and the current one have been implemented on top of the non-monotonic reasoning system REVISE using meta–logic programming with a *demo*-predicate for both, provability under well-founded semantics and abduction of conflicts [DNPS95].

Another approach closely related to ours is the meta architecture developed by ten Teije and van Harmelen [tT97, tTvH96, vHtT94]. It consists of three layers for application description, methods description, and knowledge for solving a problem flexibly. Our strategy language can be seen as implementation of this architecture. The application description corresponds to our system description and observations. The methods description contains knowledge about problem solving methods. In our language this

knowledge is kept as a parameter $diag_s$ for the underlying diagnosis engine and notion of diagnosis. The knowledge for solving a problem flexibly corresponds to our strategies. While ten Teije develops a general meta architecture for diagnostic problem solving [tTvH96, tT97], our work focuses on the declarative description of the diagnostic process and efficient algorithms to compute the process.

6.1.3 Vivid Agents

The concept of vivid agents introduced in chapter 5 is related to other high-level agent languages such as Agent0 [Sho93], PLACA [Tho95], AgentSpeak[Rao96], BDI Agents [RG91, RG93], or ARCHON [CJ96, JCL+96, WJM94, JW92]. Our agent interpreter is implemented in PVM-Prolog [CM96, CM97] which is closely related to other (logic) programming languages with explicit parallelism such as IC-Prolog][[Chu93], April [MC95], Stream Logic Programming [Rin89, HJR95], and ICE [Amt95, AB96]. We start by comparing our work to the high-level agent languages and then review the related (logic) programming languages with explicit parallelism.

Agent-oriented Programming. There are a number of proposals for high-level agent-oriented programming languages such as Agent0 [Sho93], PLACA [Tho95], or AgentSpeak [Rao96]. In these languages, the knowledge base of an agent admits only of literals upon which classical inference is performed, i.e. the CWA and negation-as-failure are not used. In the vivid agent approach, however, we take into account that, based on the experience with SQL and Prolog, the CWA and negation-as-failure are essential for information and knowledge processing. Also, neither of these languages allows intensional predicates expressing causal relations and generic laws in the form of possibly non-monotonic deduction rules which are essential for model-based diagnosis.

BDI Agents. BDI agents are modelled in terms of beliefs, desires, and intensions. Theoretical work has been investigating multi modal logics to specify agents [RG91, RG93] and independently, systems such as the Procedural Reasoning System PRS [IG90, LHDK94] have been implemented. Unfortunately, implemented systems and specification in BDI logics are not formally coupled, so that the specification logics are not executable and the implementation lacks theoretical foundation.

However, the BDI architecture underlying the implemented systems is quite general. It comprises a belief base, a set of current desires or goals, a plan library, and an intension structure. The plan library provides plans describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations. The intension structure contains those plans that have been chosen for execution. An interpreter manipulates these components, selecting appropriate plans

based on the system's beliefs and goals, placing those selected on the intension structure and executing them.

Since the BDI architecture is held quite general, the components of vivid agents can be loosely mapped to the BDI components. However, the similarity stems basically from the idea of using mental states [Sho93]. At a closer look there are several important differences. First, vivid agents are based on the concept of knowledge systems which are plugged in and therefore allow for a wide range of applications whereas BDI agents use a relational database which is for diagnosis applications insufficient. Second, vivid agents plan online whereas BDI agents fall back on plans found in a plan library. For planning as in the loading dock this is insufficient. Third, for BDI agents there is gap between theory and practice. An attempt to bridge the gap is described in [Rao96] where so-called plans which are similar to our reaction rules are introduced. Such agent specifications are in principle executable, but the actual translation executable in PRS is left open. In contrast to this, vivid agents are executable specifications with a formal semantics, thus bridging the gap between a formal underpinning and a working system.

ARCHON. ARCHON [CJ96, JCL+96, WJM94, JW92] is a framework to design industrial applications in distributed artificial intelligence. To devise a well-structured distributed application, ARCHON provides assistance for interaction of components and a methodology to structure them. ARCHON supports two approaches to the design of applications: bottom-up existing software systems are integrated to improve their performance by taking into account additional information; top-down a system is decomposed into subcomponents which are associated with an ARCHON agent. The agents' architecture consists of four main components: a monitor, a planning and coordination module, an agent information management module, and a high-level communication module. The ARCHON agents are implemented in IC-Prolog][and similar to our approach, the components of an agent run as light-weight threads. Different from our approach, ARCHON addresses the re-engineering of large, operational software systems using agent technology. Besides example applications such as an urban traffic scenario and an appointment manager [CJ96], ARCHON is used for real-world applications such as electricity transportation management and particle accelerator control [JCL+96].

IC-Prolog][. IC-Prolog][[Chu93]is a full Prolog system which supports parallel and distributed logic Programming. It offers basically the same concepts as PVM-Prolog. It allows asynchronous execution of multiple Prolog threads, each resolving independent goals. Those threads communicate through pipes the same way as PVM-Prolog threads communicate through term queues. In a networking environment, IC-Prolog][supports communication through sockets, offering a predicate library interface, whereas PVM-Prolog offers a full-fledged PVM interface library. Sockets are

good for network communication and they are widely accepted. But their use involves dealing with low-level details such as finding the communication partner, binding the socket and handling different data formats. PVM supports a higher level communication model, where name resolution and port allocation are hidden from the programmer. It also supports multi-casting, group communication, and synchronisation mechanisms. PVM also covers many multi-processor architectures where sockets are not available.

In contrast to IC-Prolog][, PVM supports heterogeneous hardware configurations with different data formats, which has two advantages: First, the internal data format allows a more efficient representation than the textual one and it makes communication among processes created by different languages possible. Second, PVM's host configuration can be changed dynamically, an issue which is not handled in IC-Prolog][. Another difference concerns IC-Prolog's indirect naming of mailboxes and PVM-Prolog's direct naming of the communication partner. IC-Prolog's mailboxes also support multiple readers. In PVM the group communication mechanisms can be used for a similar purpose.

April. `April` [MC95] is a language which supports a model of execution similar to CSP, the Communicating Sequential Processes [Hoa85], using pattern matching to check for incoming messages. It also includes real-time support, and higher-order features such as lambda abstractions, and a macro pre-processor. The communication model is point to point, with direct naming of the communicating partners. It also supports process creation and global naming through a structured name service for long range communication, which mirrors the traditional DNS. Though `April` includes ideas from logic programming such as pattern matching of messages, it does not focus on declarative and executable high-level specifications. It is rather a mix of several paradigms. Concerning `April`'s level of abstraction it ranges between PVM-Prolog which is a pure paradigm extended by message passing and a high-level multi-agent language such as vivid agents.

Stream Logic Programming. Stream logic programming [Rin89, HJR95] advocates object-based concurrent programming and combines ideas from object-orientation and logic programming. Objects are associated with heads of rules. Rules consisting of a head, a condition, and a behaviour can be nearly mapped one-to-one to event-condition-action rules [MD89]. Per event only one rule is fired which is similar to forward chainers which differs however from our approach where various reaction rules may fire if triggered by an event. An important difference between Stream Logic Programming (SLP) and other approaches concerns the message queue which is left implicit in SLP's objects message slots. Rather than a linear stream of events SLP objects are capable to handle message arrays. SLP inherits from Prolog pattern matching and rules, but important concepts such as unification, backtracking, and

negation as failure are not supported. Especially, the latter is a prerequisite for many applications. The concurrency provided by SLP is fine-grained and closer to neurons in neural networks than to our threads and processes. Synchronisation of objects is realised by pattern matching of the objects' conditions. Message receipt blocks by default, but a non-blocking receive can be programmed explicitly. The use of SLP for multi-agent systems is demonstrated by various protocols such as the contract net protocol [HJR95].

ICE. ICE [Amt95, AB96] defines a model to support distributed AI applications over hybrid languages and heterogeneous, distributed platforms. It is available from Prolog, Lisp, C, C++ and Tcl/Tk and it is, similar to PVM-Prolog, implemented on top of PVM. Its communication model is point to point through named channels but it also supports broadcasting of messages to all components. Components may be created dynamically but their communication topology must be defined in a configuration file. Configuration of components and their communication channels is done through a configuration file by a special process, the license server. PVM-Prolog, as ICE, also supports communication with tasks written in different languages. ICE's communication model is defined at a higher level than PVM-Prolog's. In ICE, it is necessary to specify the topology of the communication before running the system, in contrast to PVM's dynamic configuration, communication and process control features.

6.2 GENERAL EVALUATION

The principle goal of this work is the definition and implementation of autonomous, model-based diagnosis agents. We set out to tackle logic programming and model-based diagnosis, strategies in model-based diagnosis, and to develop and implement an architecture of autonomous, model-based diagnosis agents.

Concerning logic programming and model-based diagnosis, we showed how the rich language of extended logic programs including explicit and implicit negation as well as integrity constraints can be employed to model diagnosis problems and how contradiction removal and abduction allow to compute diagnoses. Based on a variety of applications such as digital circuits, traffic control, integrity checking of a chemical database, alarm-correlation in cellular phone networks, diagnosis of an automatic mirror furnace, and diagnosis of communication protocols, we developed modelling techniques and showed how to model diagnosis problems as extended logic programs. We improved a previous bottom-up algorithm to compute diagnoses. Our newly developed top-down algorithm solves some examples in quadratic time which previously lead to exponentially increasing timings. Our algorithm was evaluated on some of the ISCAS benchmark circuits [Isc85]. The top-down algorithm is implemented in the non-monotonic reasoning system REVISE. Our implementation features various min-

imality notions for diagnosis such as minimality by set-inclusion, by cardinality, and by probability.

To deal with complex diagnosis problems and compute diagnoses dynamically, we developed a strategy language. Taking into account both practical needs and rigorous formal treatment, we defined syntax and semantics of the strategy language. Specifying the diagnosis process declaratively with the strategy language, the very idea of model-based diagnosis is lifted to the level of the diagnosis process itself. In order to define strategies for practical applications, we identified the principles of deterministic and non-deterministic, as well as monotonic and non-monotonic strategies. We designed a strategy knowledge base for circuit diagnosis that includes a bunch of strategies for choice of models, structural refinement, measurements, and preferences. We identified rule-like strategies as appropriate for practical needs and sufficiently expressive to be still universal. We defined an operational semantics and designed an efficient algorithm for these strategies. We evaluated the strategy knowledge base and the algorithm in the domain of digital circuits and, as a proof-of-principle, we provided traces of diagnosis processes for a voter which is part of the benchmark circuits mentioned above.

For distributed applications, we presented the concept of vivid agents including a knowledge system and reaction- and action-rules to exhibit reactive as well as pro-active behaviour. To realise the concept of vivid agents we developed an architecture for concurrent action and planning. We defined a transition semantics that formally underpins the architecture. To implement the architecture we introduced PVM-Prolog which provides coarse-grain and fine-grain parallelism. The former is used to spawn agents in the network, while the latter is used to run the action and planning component concurrently. We evaluated the concept and implementation of vivid agents in distributed diagnosis by implementing fault-tolerant diagnosis and the diagnosis of a communication protocol. To evaluate these applications we gave traces of the executed specifications. The implemented agent interpreter satisfies the main criteria for a state-of-the-art multi-agent programming language. It allows to specify reactive and pro-active behaviour, it is under-pinned by a formal semantics, the high-level agent specifications are executable, the agents are platform independent, and heterogeneous agents are supported.

6.3 FUTURE WORK

This book opens up three interesting areas for future work: First, the comparison of the diagnosis agents with agent standards currently under development; second, the further improvement of the algorithms; third, the development of real-world applications.

The agent interpreter is closely related to the ongoing specification of agent standards by a consortium of many (telecommunication) enterprises and universities. The

second version of a proposal for the *Foundation of Intelligent Physical Agents* (FIPA) is still under discussion [C+97]. Once it is approved as a standard, it will be interesting to investigate if the vivid agents meet the FIPA-specification or if they can be modified accordingly. At the current stage, it appears that the interpreter is quite close to the first part of the proposal concerning agent management. Similarities include for example the distinction of local and global communication and a central directory facilitator. The second part of the FIPA proposal deals with an agent communication language based on speech acts [Aus62, Sea69, FWW+93] to classify different types of messages and according protocols for message exchange as well as a framework to reason about agent beliefs. The vivid agent interpreter may serve as an implementation platform to this extent. It would be interesting to evaluate the higher-level second part of the FIPA proposal using our agent interpreter in order to find out whether our agent interpreter is suitable to implement the specified protocols and, more general, whether FIPA's agent communication language is suitable for implementation.

The second area for future work concerns the improvement of the algorithms, in particular the revision and planning algorithms. To gain further speed-ups, the diagnosis engine REVISE may benefit from tabling systems such as XSB-Prolog [SSW94] and assumption-based truth maintenance [Doy79, dK86] which both help to avoid recomputations. A different line of thinking is to adopt the techniques used in the very efficient diagnoser DRUM II [FN96a, FN96b, NF97, FN97]. Recently, it turned out that the ideas underlying DRUM II are general and have been successfully applied to a theorem prover [PBN97b, PBN97a]. The main obstacle for using the DRUM II-algorithm for REVISE will concern the two negations provided by REVISE and REVISE' abductive capabilities which are more complex than consistency-based diagnosis.

Recently, there has been progress in efficient planning. Kautz and Selman presented a planning algorithm based on a translation of planning problems to a SAT problem which is efficiently solved by a random–walk algorithm [KS96, SKM97]. To integrate such a state-of-the-art planner into our implementation, it has to be enhanced by reactive planning. Currently, replanning generates a new plan independent of the previous plan which failed. To save computational resources, it would be interesting to investigate how the failed plan could be as slightly as possible modified to accomplish the given goal.

Third, future work may be dedicated to the design of applications. In the meantime, we have gathered first experiences in the development of an agent-based system for knowledge-integration on the internet [SN97] and argumentation in multi-agent systems [SMA97, MAS97]. As a real-world application the domain of telecommunication network management is interesting. While diagnosis and monitoring have always been important topics in this field, current technological developments suggest that agent technology will be of great importance for future innovations.

7 PROOFS AND PROOF SKETCHES

Proof Sketch of Theorem 3.24 To prove that the top-down algorithm in Figure 3.27 terminates, we argue that there is only a finite number of revisables and subsequently only a finite number of possible nodes. With a node being marked in each step of loop (2) the algorithm terminates. The second part of the theorem that $H(n)$ is a hitting-set iff n is a leaf marked $\surd$ is true since we can replace conflict generation beforehand by online conflict generation using *demo* by proposition 3.21 and lemma 3.23. Thus the top-down algorithm 3.27 is equivalent to the hitting-set algorithm 3.21. □

Proof of Proposition 4.16

1. We show that $(S,\rightarrow,t) \models_s \Diamond L$ iff $\mathcal{D} \neq \emptyset$ and there is a diagnosis $D \in \mathcal{D}_s$ such that the model for $SD \cup OBS \cup s \cup \neg\bar{s} \cup D$ entails L or $\mathcal{D} = \emptyset$ and the model for $s \cup \neg\bar{s} \cup \{ab(SD)\}$ entails L.

   Before we start the proof we need a lemma

   Lemma 7.1 *Let $L \in \mathcal{L}$. Then $M_{(S,\rightarrow,t)} \models_{(s,j)} L$ iff $M_{(s,j)} \models_{\mathcal{L}} L$.*

By definition 4.13 $(S,\rightarrow,t) \models_s \Diamond L$ iff $M_{(S,\rightarrow,t)} \models_{(s,i)} \Diamond L$. By definition 4.11 this is equivalent to $\exists w'$ s.th. $(s,i) \rightarrow_2' w'$ and $M_{(S,\rightarrow,t)} \models_{w'} L$. By definition 4.12 there is $1 \leq j \leq m_s$ s.th. $(s,i) \rightarrow_2' (s,j)$ and $M_{(S,\rightarrow,t)} \models_{(s,j)} L$ and by the lemma above we have $M_{(s,j)} \models_{\mathcal{L}} L$ which means by the definition of $M_{(s,j)}$ in definition 4.12 that $\mathcal{D} \neq \emptyset$ and there is a diagnosis $D \in \mathcal{D}_s$ such that the model for $SD \cup OBS \cup s \cup \neg\bar{s} \cup D$ entails L or $\mathcal{D} = \emptyset$ and the model for $s \cup \neg\bar{s} \cup \{ab(SD)\}$ entails L.

2. The proof for $\Box L$ is similar.

3. To prove $(S,\rightarrow,t) \models_s \blacklozenge F$ iff ex. $s' \in S$ such that $s \rightarrow s'$ and $(S,\rightarrow,t) \models_{s'} F$ we proceed as above and apply definition 4.13, 4.11, and 4.12.

4. The proof for $\blacksquare F$ is similar. □

Proof for Proposition 4.19 We show that any diagnosis under the n-fault assumption nf does not contain more than n components. Let $\mathcal{D}_s = diag_s(SD \cup OBS)$. Assume $nf \in s$ and $D \in \mathcal{D}_s$ s.th. $|D| > n$. Then there are at least $n+1$ distinct components C_i s.th. $ab(C_i) \in D$. Thus the lefthand side of the formula

$$\forall C_1,\ldots,C_{n+1} : nf \wedge \bigwedge_{i=1}^{n+1} ab(C_i) \rightarrow \bigvee_{i,j=1,i\neq j}^{n+1} C_i = C_j$$

is satisfied. To be consistent the right-hand side has to be satisfied as well, i.e. at least two components are equal. This is contradictory to choosing $n+1$ distinct components. □

Proof Sketch of Lemma 4.21 $\Leftarrow$ Application of the definition of $\models_s$ and G_s. $\Rightarrow$ Assume $s \neq s'$. Then there is a working hypothesis wh such that without loss of generality $wh \in s'$ but $wh \notin s$. From the premise $(S,\rightarrow,\emptyset) \models_{s'} G_s$ we can conclude $(S,\rightarrow,\emptyset) \models_{s'} \Box\neg wh$ and $(S,\rightarrow,\emptyset) \not\models_{s'} \Box wh$ since $wh \notin s$. But by definition of $\models_{s'}$ we have also $(S,\rightarrow,\emptyset) \models_{s'} \Box wh$ since $wh \in s'$. A contradiction. □

Note that the last conclusion requires that a working hypothesis is only derivable iff it is contained in the state which is guaranteed by adding working hypotheses and the negated complements.

Proof of Theorem 4.23 The set $\mathcal{F}$ of one-step strategies is defined by transition formulas that express that if we are in state s, where G_s holds then the successors' state formulas hold possibly and their disjunction holds necessarily. The case that the state s has no successors is captured by the formula $\blacksquare$*false*:

Definition 7.2 *Let $(S,\rightarrow,\emptyset)$ be a transition system and G_s the state formula for a state $s \in S$. Then*

$$G_s \rightarrow \bigwedge_{s\rightarrow s'} \blacklozenge G_{s'} \wedge \blacksquare \bigvee_{s\rightarrow s'} G_{s'} \qquad \text{if there is } s'' \text{ such that } s \rightarrow s''$$
$$G_s \rightarrow \blacksquare \mathit{false} \qquad \text{else}$$

is called Transition Formula of s.

Remark: Transition formulas of states with successors are one-step strategies. The heads are characteristic formulas.

Now we can turn to the proof of the theorem. Let $(S,\rightarrow,\emptyset)$ be a transition system and let $\mathcal{F}$ be the set of transition formulas of the states S.
$\Leftarrow$ We have to show that $(S,\rightarrow,\emptyset) \models \mathcal{F}$. Let $s \in S$ and $F_s \in \mathcal{F}$. We show that for all states $s' \in S$ we have $(S,\rightarrow,\emptyset) \models_{s'} F_s$ by examining the cases $s \neq s'$ and $s = s'$. In case $s' \neq s$ we know by lemma 4.21 that $(S,\rightarrow,\emptyset) \not\models_{s'} G_s$ and thus $(S,\rightarrow,\emptyset) \models_{s'} F_s$. In case $s' = s$ we know by lemma 4.21 that $(S,\rightarrow,\emptyset) \models_{s'} G_s$ so that it is left to prove that $(S,\rightarrow,\emptyset) \models_{s'} \bigwedge_{s\rightarrow s''} \blacklozenge G_{s''} \wedge \blacksquare \bigvee_{s\rightarrow s''} G_{s''}$ in case $\exists s'' : s \rightarrow s''$ and $(S,\rightarrow,\emptyset) \models_{s'} \blacksquare \mathit{false}$ otherwise. The latter formulas hold by definition of $\blacksquare$ and $\blacklozenge$.
$\Rightarrow$ The proof is by contradiction. Assume that $(S',\rightarrow',\emptyset) \models \mathcal{F}$ and $(S',\rightarrow',\emptyset) \neq (S,\rightarrow,\emptyset)$. We have $\emptyset \in S$ and starting with $\emptyset$ all states of S are "traversed" by the transition formulas $\mathcal{F}$. Since also $\emptyset \in S'$ and $(S',\rightarrow',\emptyset) \models \mathcal{F}$ we can conclude that $S' \supset S$. Now assume $s \in S' \cap S$ and $s'' \in S' \backslash S$ such that $s \rightarrow s''$. Since $s'' \notin S$ we have $(S',\rightarrow',\emptyset) \not\models_{s''} \bigvee_{s\rightarrow s'} G_{s'}$. As $s \rightarrow s''$ we conclude $(S',\rightarrow',\emptyset) \not\models_s \blacksquare \bigvee_{s\rightarrow s'} G_{s'}$. From this and the fact that $(S',\rightarrow',\emptyset) \models_s G_s$ we conclude $(S',\rightarrow',\emptyset) \not\models_s F_s$ in contradiction to the assumption. □

Proof Sketch of Theorem 4.27 The proof proceeds in three steps: 1. $(S,\rightarrow,t)$ satisfies the conjunction of all H_i. 2. all transition systems that possibly satisfy the conjunction of all H_i consist of subsets of S. 3. None of these candidates actually satisfies independence of strategies, so $(S,\rightarrow,t)$ in turn does.

Ad 1. In order to show that $(S,\rightarrow,t) \models H_i$ for $1 \leq i \leq n$ it is sufficient to prove that $(S,\rightarrow,t) \models_\emptyset \bigwedge_{\emptyset\rightarrow_i s} \blacklozenge F_s \wedge \blacksquare \bigvee_{\emptyset\rightarrow_i s} F_s$, where $F_s = \Box \bigwedge \{wh \mid wh \in s\} \cup \{\neg wh \mid wh \in (\bigcup S_i) \backslash s\}$.

To prove that the two conjuncts hold the definition of the state product is essential. Intuitively, the first conjunct holds as every state in S_i is a subset of at least one state in S. For the second conjunct we argue the other way around: it holds as every state in S is a superset of at least one state in S_i.

Ad 2. As H_i is characteristic and has depth 1 it consists of subformulas $\blacksquare \bigvee_{\emptyset\rightarrow_i s} F_s$. Thus $\bigwedge_{i=1}^{n} H_i$ contains $\bigwedge_{i=1}^{n} \blacksquare \bigvee_{\emptyset\rightarrow_i s} F_s$ which is equivalent to $\blacksquare \bigwedge_{i=1}^{n} \bigvee_{\emptyset\rightarrow_i s} F_s$. As the formulas

F_s do not have any working hypotheses in common the latter expression is in minimal CNF. The equivalent DNF corresponds one-to-one to the state product. So the conjunction of H_i can only be satisfied by a subset of the state product.

Ad 3. We show that the first condition for independence of strategies is violated. Let $(S_1, \rightarrow_1, t_1)$ such that $S_1 \subset S$, $\rightarrow_1 \subset \rightarrow$ and $t_1 = t$. Then $(S, \rightarrow, t) \models_s \blacksquare\Box wh_1 \rightarrow wh_2$ for all $wh_1, wh_2 \in S \cup \neg S$ implies $(S_1, \rightarrow_1, t_1) \models_s \blacksquare\Box wh_1 \rightarrow wh_2$ since $S_1 \subset S$. □

References

[AB96] J. W. Amtrup and J. Benra. Communication in large distributed AI systems for natural language processing. In *Proceedings of the 16th International Conference on Computational Linguistics, COLING-96.*, pages 35–40, Copenhagen, 1996.

[ADP94a] J. J. Alferes, C. V. Damásio, and L. M. Pereira. Top–down query evaluation for well–founded semantics with explicit negation. In A. Cohn, editor, *Proc. of the European Conference on Artificial Intelligence'94*, pages 140–144. John Wiley & Sons, August 1994.

[ADP94b] J. J. Alferes, C. V. Damásio, and L. M. Pereira. A top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *Proc. of the International Logic Programming Symposium'94*, pages 424–438. MIT Press, November 1994.

[ADP95] J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14(1):93–147, 1995.

[Amt95] J. W. Amtrup. *ICE - Intarc Communication Environment: User's Guide and Reference Manual. Version 1.4.* University of Hamburg, 1995.

[AP96] J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming.* (LNAI 1111), Springer-Verlag, 1996.

[Aus62] J.L. Austin. *How to Do Things with Words.* Harvard University Press, Cambridge (MA), 1962.

[BD93] C. Böttcher and O. Dressler. Diagnosis process dynamics: Holding the diagnostic trackhound in leash. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1460–1471. Morgan Kaufmann Publishers, Inc., 1993.

[BD94] C. Böttcher and O. Dressler. A framework for controlling model-based diagnosis systems with multiple actions. *Annals of Mathematics and Artificial Intelligence, special Issue on Model-based Diagnosis*, 11(1–4), 1994.

[BDP96] S. Brass, J. Dix, and T. C. Przymusinski. Super logic programs. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning KR96*, pages 529–541, San Francisco, 1996. Morgan Kaufmann Publishers, Inc.

[BF94] P. Baumgartner and U. Furbach. Model elimination without contrapositives and its application to PTTP. *Journal of Automated Reasoning*, 13(1):339–359, 1994.

[BFS95] P. Baumgartner, U. Furbach, and F. Stolzenburg. Model elimination, logic programming and computing answers. In *IJCAI-95*, Montréal, Canada, August 1995. Morgan Kaufmann Publishers, Inc.

[BM92] C. B. Boyer and U. C. Merzbach. *History of Mathematics*. John Wiley & Sons, second edition, 1992. First published in 1968.

[BPH85] F. Brglez, P. Pownall, and R. Hum. Accelerated ATPG and fault grading via testability analysis. In *Proceedings of IEEE Int. Symposium on Circuits and Systems*, pages 695–698, 1985. The ISCAS85 benchmark netlists are available via ftp `mcnc.mcnc.org`.

[BS84] B. G. Buchanan and E. H. Shortliffe. *Rule-based expert systems: the MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley Publishing Company, Reading, MA, 1984.

[C+97] L. Chiariglione et al. Specification version 2.0. Technical report, Foundations of Intelligent Physical Agents, 1997. http://drogo.cselt.stet.it/-fipa/.

[Che80] B. F. Chellas. *Modal Logic – An Introduction*. Cambridge University Press, 1980.

[Chu93] D. Chu. I.C. PROLOG][: A language for implementing multi-agent systems. In S. M. Dean, editor, *Proceedings of the 1992 Workshop*

on Cooperating Knowledge Based systems (CKBS92), pages 61–74. DAKE Centre, University of Keele, UK, 1993.

[CJ96] D. Cockburn and N. R. Jennings. ARCHON: A distributed artificial intelligence system for industrial applications. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*. John Wiley & Sons, 1996.

[CM96] J. C. Cunha and R. F. P. Marques. PVM-Prolog: A prolog interface to PVM. In *Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS'96*, Miskolc, Hungary, 1996.

[CM97] J. C. Cunha and R. F. P. Marques. Distributed algorithm development with PVM-Prolog. In *5th Euromicro Workshop on Parallel and Distributed Processing*, London, UK, 1997. IEEE Computer Society Press.

[CT91] L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[Dav83] M. Davis. The prehistory and early history of automated deduction. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning Classical Papers on Computational Logic, 1957–1966*. Springer–Verlag, 1983.

[Dav84] R. Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24:347-410, 1984.

[Dee94] S.M. Deen (Ed.). *Proc. 2nd Int. Working Conf. on Cooperating Knowledge-Based Systems*. DAKE Centre, University of Keele, 1994.

[DH88] R. Davis and W. Hamscher. Model-based reasoning: Troubleshooting. In *Exploring Artificial Intelligence*, chapter 8, pages 297–346. Morgan Kaufmann Publishers, Inc., 1988.

[dK86] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[dK91] J. de Kleer. Focusing on probable diagnoses. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 842–848, Anaheim, July 1991. Morgan Kaufmann Publishers, Inc.

[dKB92] J. de Kleer and J. S. Brown. Model-based diagnosis in SOPHIE III. In *[HCd92]*, 1992.

[dKRS91] J. de Kleer, O. Raiman, and M. Shirley. One step lookahead is pretty good. In *Second International Workshop on the Principles of Diagnosis*, Milano, Italy, October 1991.

[dKW87] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[dKW89] J. de Kleer and B. C. Williams. Diagnosis with behavioral modes. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1324–1330, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[DNP94] C. V. Damásio, W. Nejdl, and L. M. Pereira. REVISE: An extended logic programming system for revising knowledge bases. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Knowledge Representation and Reasoning*, pages 607–618, Bonn, Germany, May 1994. Morgan Kaufmann.

[DNPS95] C. V. Damásio, W. Nejdl, L. M. Pereira, and M. Schroeder. Model-based diagnosis preferences and strategies representation with meta logic programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, chapter 11, pages 269–311. The MIT Press, 1995.

[Doy79] J. Doyle. A truth maintenace system. *Artificial Intelligence*, 12:231–272, 1979.

[DPS96] C. V. Damásio, L. M. Pereira, and M. Schroeder. REVISE progress report. In *Workshop on Automated Reasoning: Bridging the Gap between Theory and Practice*. University of Sussex, Brighton, 1996.

[DPS97a] C. V. Damásio, L. M. Pereira, and M. Schroeder. Revise online. http://www.kbs.uni-hannover.de/~schroede/ revise/ revise.html, 1997.

[DPS97b] C. V. Damásio, L. M. Pereira, and M. Schroeder. REVISE: Logic programming and diagnosis. In *Proceedings of the Conference on Logic Programming and Non-monotonic Reasoning LPNMR97*. LNAI 1265, Springer–Verlag, 1997.

[DS92] O. Dressler and P. Struss. Back to defaults: Characterizing and computing diagnoses as coherent assumption sets. In *ECAI92*, pages 719–723, 1992.

[Eco94] U. Eco. *Die Suche nach der vollkommenen Sprache*. C.H. Beck, 1994. Originally published in italian "La ricera della lingua perfetta nella cultura europea".

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16. Elsevier, Amsterdam, New York, 1990.

[FdAMNS97] P. Fröhlich, I. de Almeida Móra, W. Nejdl, and M. Schroeder. Diagnostic agents for distributed systems. In *Proceedings of ModelAge97*, Siena, Italy, 1997.

[FdCH95] L. Farinas del Cerro and A. Herzig. Modal deduction with applications in epistemic and temporal logics. In C.J. Hogger Dov M. Gabbay and J.A. Robinson, editors, *The Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, pages 499–594. Oxford Science Publications, 1995.

[FGN90] G. Friedrich, G. Gottlob, and W. Nejdl. Physical impossibility instead of fault models. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 331–336, Boston, August 1990. Also appears in W. Hamscher, L. Console, J. de Kleer, eds, Readings in Model-Based Diagnosis, Morgan Kaufmann, 1992.

[FMP96] K. Fischer, J. P. Müller, and M. Pischel. A pragmatic BDI architecture. In M. Wooldridge, J. Müller, and M. Tambe, editors, *Intelligent Agents II*. LNAI 1037, Springer–Verlag, 1996.

[FMS+97] M. Fisher, J. P. Müller, M. Schroeder, G. Wagner, and G. Staniford. Methodological foundations of agent-based systems. *IEEE Knowledge Engineering Review*, 12(3):323-329, 1997.

[FN71] R.E. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, pages 189–208, 1971.

[FN90] G. Friedrich and W. Nejdl. MOMO — Model-based diagnosis for everybody. In *Proceedings of the IEEE Conference on Artificial Intelligence Applications (CAIA)*, Santa Barbara, March 1990. A slightly revised and extended version appears in W. Hamscher, L. Console, J. de Kleer, eds, Readings in Model-Based Diagnosis, Morgan Kaufmann, 1992.

[FN96a] P. Fröhlich and W. Nejdl. A model–based reasoning approach to circumscription. In *Proceedings of the 12th European Conference on Artificial Intelligence*, 1996.

[FN96b] P. Fröhlich and W. Nejdl. A model–based reasoning approach to circumscription – extended version. In *Sixth International Workshop on Nonmonotonic Reasoning*, 1996.

[FN97] P. Fröhlich and W. Nejdl. A static model-based engine for model-based reasoning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, Nagoya, Japan, 1997.

[FNJW97] P. Fröhlich, W. Nejdl. K. Jobmann, and H. Wietgrefe. Model-based alarm correlation in cellular phone networks. In *Fifth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, January 1997.

[FNS94] P. Fröhlich, W. Nejdl, and M. Schroeder. A formal semantics for preferences and strategies in model-based diagnosis. In *5th International Workshop on Principles of Diagnosis (DX-94)*, pages 106–113, New Paltz, NY, October 1994.

[FNS96] P. Fröhlich, W. Nejdl, and M. Schroeder. Design and implementation of diagnostic strategy using modal logic. In *JELIA96 - European workshop on Logic in AI*. LNAI 1126, Springer–Verlag, 1996.

[FNS97] P. Fröhlich, W. Nejdl, and M. Schroeder. Strategies in model-based diagnosis. *Journal of Automated Reasoning*, 1998.

[FWW+93] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro, and C. Beck. Specification of the KQML agent communication language. Technical report, The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, Baltimore, USA, 1993.

[Gea94] A. Geist and et al. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[Gel89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the 8th ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.

[Gen84] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1984.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. of ICLP90*, pages 579–597. MIT Press, 1990.

[GRS88] A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceeding of the*

7th ACM Symposium on Principles of Databse Systems, pages 221–230. Austin, Texas, 1988.

[GRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of the ACM*, 38(3), 1991.

[GSW89] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction of the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[Ham91] W. C. Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51(1-3):223–271, October 1991.

[Ham94] S. Hampe. Methoden und Verfahren der Konsistenzerhaltung am Beispiel eines Expertensytems zur Bestimmung der Medienbeständigkeit von Kunststoffen. Master's thesis, RWTH Aachen, 1994. Prof. W. Nejdl, zusammen mit IKV, Prof. M.i/Hr. Menzenbach.

[HCd92] W. Hamscher, L. Console, and J. de Kleer. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[HJR95] M. M. Huntbach, N. R. Jennings, and G. A. Ringwood. How agents do it in stream logic programming. In *Proceedings of the First International Conference on Multi-Agent Systems ICMAS95*, pages 177–184, San Francisco, USA, 1995.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[IG90] F. F. Ingrand and M. P. Georgeff. Managing deliberation and reasoning in real-time AI systems. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning*, pages 284–291, San Diego, CA, 1990.

[Isc85] The ISCAS-85 Benchmarks. http://www.cbl.ncsu.edu/www/CBL_-Docs/iscas85.html, 1985.

[JCL+96] N. R. Jennings, J. M. Corera, I. Laresgoiti, E. H. Mamdani, F. Perriollat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle accelerator control. *IEEE Expert*, 1996.

[JW92] N. R. Jennings and T. Wittig. ARCHON: Theory and practice. In N. M. Avouris and L. Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 179–195. Kluwer Academic Press, 1992.

[KGB+95] D. Kinny, M. Georgeff, J. Bailey, D. B. Kemp, and K. Ramamohanarao. Actice databases and agent systems – a comparison. In *Proceedings of RIDS95, International Workshop of Rules in Database Systems*, Athens, Greece, 1995.

[Kon92] K. Konolige. Abduction versus closure in causal theories. *Artificial Intelligence*, 53:255–272, 1992.

[Kow95] R. A. Kowalski. Using meta-logic to reconcile reactive with rational agents. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, chapter 9, pages 227–242. The MIT Press, 1995.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI96*, pages 1194–1201, 1996.

[LHDK94] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *CIRFSS94, Conference on Intelligent Robotics in Field, Factory, Service and Space*, pages 842–849. MIT Press, 1994.

[MA95] I. A. Móra and J. J. Alferes. Diagnosis of distributed systems using logic programming. In C. Pinto-Ferreira and N.J. Mamede, editors, *Progress in Artificial Intelligence, 7th Portuguese Conference on Artificial Intelligence EPIA95*, volume LNAI990, pages 409–428. Springer–Verlag, Funchal, Portugal, 1995.

[MAS97] I. Móra, J. J. Alferes, and M. Schroeder. Argumentation for distributed extended logic programs. In *Proceedings of the International Workshop on Logic Programming and Multi-Agents*, 1997.

[MC95] F. G. McCabe and K. L. Clark. `APRIL` - Agent PRocess Interaction Language. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents I.* LNAI 890, Springer–Verlag, 1995.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD-89*, pages 215–224, 1989.

[MH93] I. Mozetič and C. Holzbauer. Controlling the complexity in model–based diagnosis. *Annals of Mathematics and Artificial Intelligence*, 1993.

[Moz91] I. Mozetič. Hierarchical model-based diagnosis. *International Journal of Man-Machine Studies*, 35:329–362, 1991.

[MPT95] J. P. Müller, M. Pischel, and M. Thiel. Modelling reactive behaviour in vertically layered agent architectures. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents I.* LNAI 890, Springer–Verlag, 1995.

[Mül94] J.P. Müller. A conceptual model of agent interaction. In *[Dee94]*, pages 213–233, 1994.

[Mül96] J. P. Müller. *The Design of Intelligent Agents: A layered Approach.* LNAI 1177, Springer–Verlag, 1996.

[NF97] W. Nejdl and P. Fröhlich. Minimal model semantics for diagnosis – techniques and first benchmarks. In *7th International Workshop on Principles of Diagnosis*, Val Morin, Canada, 1997.

[NFS95] W. Nejdl, P. Fröhlich, and M. Schroeder. A formal framework for representing diagnosis strategies in model–based diagnosis systems. In *IJCAI-95*, pages 1721–1727, Montréal, Canada, August 1995. Morgan Kaufmann Publishers, Inc.

[NG94] W. Nejdl and B. Giefer. DRUM: Reasoning without conflicts and justifications. In *5th International Workshop on Principles of Diagnosis (DX-94)*, pages 226–233, October 1994.

[NS96] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, 1996. MIT Press.

[PA92] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In *B. Neumann (Ed.), European Conference on Artificial Intelligence*, pages 102–106. John Wiley & Sons, 1992.

[PAA91] L. M. Pereira, J. J. Alferes, and J. Aparicio. Contradiction Removal within Well Founded Semantics. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 105–119, Washington, USA, June 1991. MIT Press.

[PAA93] L. M. Pereira, J. N. Aparicio, and J. J. Alferes. Non–monotonic reasoning with logic programming. *Journal of Logic Programming. Special issue on Nonmonotonic reasoning*, 17(2, 3 & 4), 1993.

[PAA94] L. M. Pereira, J. J. Alferes, and J. N. Aparicio. Contradiction removal semantics with explicit negation. In M. Masuch and L. Pólos, editors, *Knowledge Representation and Reasoning Under Uncertainty*, volume 808 of *LNAI*, pages 91–106. Springer-Verlag, 1994.

[PBC+97] B. Pell, D. E. Bernhard, S. A. Chien, E. Gat, N. Muscettola, P. Pandurang Nayak, M. D. Wagner, and B. C. Williams. An autonomous spacecraft agent prototype. In *Proceedings of First Internationl Conference on Autonomous Agents, AA97*, pages 253–261. ACM Press, 1997.

[PBN97a] U. Furbach P. Baumgartner, P. Fröhlich and W. Nejdl. Semantically guided theorem proving for diagnostic applications. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 460–465, Nagoya, 1997.

[PBN97b] U. Furbach P. Baumgartner, P. Fröhlich and W. Nejdl. Tableaux for diagnosis applications. In *Proceedings of the International Conference on Analytical Tableaux and Related Methods (TABLEAUX'97*, 1997.

[PBT+93] E. Pelikan, K. Bohndorf, T. Tolxdorff, D. Zarrinnam, and B. Wein. Computer-assisted diagnosis of bone tumors. In H. U. Lemke, K. Inamura, C. C. Jaffe, and R. Felix, editors, *Computer Assisted Radiology*, pages 630–633, Berlin, 1993. Springer-Verlag.

[PBT+94] E. Pelikan, K. Bohndorf, T. Tolxdorff, D. Zarrinnam, and B. Wein. Computerunterstützte Diagnose von pathologischen Skelettbefunden. *Radiologica Diagnostica*, 35/1, juli 1994.

[PDA93] L. M. Pereira, C. V. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *2nd Int. Workshop on Logic Programming and Non-Monotonic Reasoning*, pages 334–348, Lisboa, Portugal, June 1993. MIT Press.

[Poo89] D. Poole. Normality and faults in logic-based diagnosis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1304–1310, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[Prz95] T. C. Przymusinski. Static semantics for normal and disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 14(1):323–357, 1995.

[Rao96] A. S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In *Proceedings of MAAMAW96, LNAI 1038*. Springer-Verlag, 1996.

[RdKS93] O. Raiman, J. de Kleer, and V. Saraswat. Critical reasoning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 18–23, Chambery, August 1993. Morgan Kaufmann Publishers, Inc.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.

[RG91] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR91, International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, Inc., 1991.

[RG93] A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In Ruzena Bajcsy, editor, *IJCAI93, International joint Conference on Artificial Intelligence*, volume 1, pages 318–324. Morgan Kaufmann Publishers, Inc., 1993.

[Rin89] G. A. Ringwood. A comparative exploration of concurrent logic languages. *Knowledge Engineering Review*, 4(1):305–332, 1989.

[Sch94] D. Schüth. Interaktives fehlermanagement in verkehrsinformationsnetzen. Master's thesis, RWTH Aachen, 1994.

[Sch95] M. Schroeder. A final frontier: Preferences for strategies. In *VI Conference of the Spanish Association for Artificial Intelligence CAEPIA95*, Alicante, Spain, November 1995.

[Sch97] M. Schroeder. A brief history of the notation of Boole's algebra. *Nordic Journal of Philosophical Logic*, 2(1):41–62, 1997.

[SD89] P. Struss and O. Dressler. Physical negation — Integrating fault models into the general diagnostic engine. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1318–1323, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[SdAMP97] M. Schroeder, I. de Almeida Móra, and L. M. Pereira. A deliberative and reactive diagnosis agent based on logic programming. In *Intelligent Agents III, LNAI 1193*. Springer–Verlag, 1997. As poster in *Proc. of International Conference on Tools in Artificial Intelligence ICTAI96*, Toulouse, 1996.

[SDP96] M. Schroeder, C. V. Damásio, and L. M. Pereira. REVISE report: An architecture for a diagnosis agent. In *Proceedings of the ECAI'96 Workshop on Integrating Nonmonotonicity into Automated Reasoning Systems*, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1996.

[Sea69] J.R. Searle. *Speech Acts*. Cambridge University Press, Cambridge (UK), 1969.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[SKM97] B. Selman, H. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, Nagoya, Japan, 1997.

[SMA97] M. Schroeder, I. Móra, and J. J. Alferes. Vivid agents arguing about distributed extended logic programs. In *Proceedings of the Portuguese Conference on Artificial Intelligence EPIA97*. LNAI 1323, Springer–Verlag, 1997.

[SMWC97] M. Schroeder, R. Marques, G. Wagner, and J. Cunha. CAP - Concurrent Action and Planning: Using PVM-Prolog to implement vivid agents. In *Proceedings of the 5th Conference on Practical Applications of Prolog*, London, UK, April 1997.

[SN97] M. Schroeder and W. Nejdl. Rapid prototyping for web-mediators - integrating distributed knowledge using vivid agents. In *Working Notes of the Sixth Workshop on Enabling Technologies: Infrastructure for Collabortive Enterprises*, Boston, USA, 1997.

[SNH94] D. Schüth, W. Nejdl, and R. Hager. Fault management of infrastructure networks. In *Proceedings of the Vehicular Technology Conference*, Stockholm, June 1994. IEEE/VTS.

[SSW94] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994.

[SSW96] K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for computing the well-founded semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 274–288, Bonn, Germany, 1996. MIT Press.

[Str92a] P. Struss. Diagnosis as a process. In *[HCd92]*, 1992. First appeared in Working Notes of the first International Workshop on Model-based Diagnosis, Paris, 1989.

[Str92b] P. Struss. What's in SD? Towards a theory of modeling in diagnosis. In *Readings in Model-Based Diagnosis*, pages 419–449. Morgan Kaufmann Publishers, Inc., 1992.

[SW97] M. Schroeder and G. Wagner. Distributed diagnosis by vivid agents. In *Proceedings of First Internationl Conference on Autonomous Agents, AA97*, pages 268–275, Marina del Rey, USA, February 1997. ACM Press.

[TD95] G. Theiss and R. Dobiasch. Study on application of system modelling in spacecraft control. Technical report, European Space Agency, 1995.

[The95] G. Theiss. *Systemtechnischer Ansatz für die Entwicklung von Expertensystemen zur Überwachung und Diagnose automatisierter Anlagen : am Beispiel einer Spiegelofen-Kristallzuchtanlage*. PhD thesis, University of Karlsruhe, Germany, 1995.

[Tho95] S. R. Thomas. The PLACA agent programming language. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents I*. LNAI 890, Springer–Verlag, 1995.

[tT97] A. ten Teije. *Automated Configuration of Problem Solving Methods in Diagnosis*. PhD thesis, Universiteit Amsterdam, 1997.

[tTvH96] A. ten Teije and F. van Harmelen. Using refelxion techniques for flexible problem solving. *Future Generation Computer Systems*, 12(1):217–234, 1996. Special Issue on Reflection and Meta-level AI Architectures.

[vHtT94] F. van Harmelen and A. ten Teije. Using domain knowledge to select solutions in abductive diagnosis. In A. G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence ECAI94*, pages 652–656. John Wiley & Sons, 1994.

[Wag91] G. Wagner. Logic programming with strong negation and innexact predicates. *Journal of Logic and Computation*, 1(6), 1991.

[Wag94] G. Wagner. *Vivid Logic – Knowledge-Based Reasoning with Two Kinds of Negation*, LNAI 764. Springer–Verlag, 1994.

[Wag95] G. Wagner. From information systems to knowledge systems. In E.D. Falkenberg et al., editor, *Proc. of Information System Concepts*. Chapman & Hall, 1995.

[Wag96] G. Wagner. A logical and operational model of scalable knowledge- and perception-based agents. In *Proceedings of MAAMAW96, LNAI 1038*. Springer-Verlag, 1996.

[WJ95] M.J. Wooldridge and N.R. Jennings. Agent theories, architectures and languages: A survey. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents I*. LNAI 890, Springer–Verlag, 1995.

[WJM94] T. Wittig, N. R. Jennings, and E. H. Mamdani. ARCHON - a framework for intelligent cooperation. *IEE-BCS Journal of Intelligent Systems Engineering*, 3(3):168–179, 1994.

[WN96] B. C. Williams and P. P. Nayak. Immobile robots: AI in the New Millenium. *AI Magazine*, 17(3):16–36, Fall, 1996.

Index