BREL — A Prolog Knowledge-Based System Shell for VLSI CAD

Marwan A. Jabri

Systems Engineering and Design Automation Laboratory (SEDAL)
Sydney University Electrical Engineering
NSW 2006 Australia

I. Introduction

VLSI design automation is an activity that has a combinatorial nature, a large solution space (it is a *design* problem), it is complex (the number of interacting devices is an example of complexity), it is of a multi-constraint optimisation nature; several design constraints such as speed, power and area are competing at the same time and each representing a *dimension* of an NP complete problem. As a result, Artificial Intelligence (AI) programming techniques are becoming widely used in the automation of VLSI design tasks. Most notable amongst these techniques is what is commonly known as Knowledge-Based Systems (KBS). The implementation of a KBS that deals with a VLSI CAD domain requires consideration to key issues including complexity, the nature of information processing, and automation requirements. These issues influence considerably the structure of the KBS.

Solving a problem corresponds to the transformation of an original statement of the problem to a final statement representing a solution. Each transformation leads to a new statement that describes a partial (incomplete) or complete solution. We use the briefer terms **state** and **context** interchangeably in place of the term **statement-of-the-problem**. The current context (or current state) is held in the Current Context Memory (CCM) or simply the context. Transformation is carried out by the application of rules to the current context. Rules are held in the knowledge base. The inference engine is the procedure which selects and applies the rules.

In a KBS where rules are used to represent knowledge, it is important to devise a solution search strategy. Two strategies are commonly used: solution improvement (we produce a solution and then improve on it) and backtracking (we produce a solution and if it is not satisfactory we backtrack in order to find a better one). The former search strategy involves the design of complex transformation rules that are going to improve the quality of the solution. The latter search strategy is more computationally demanding and may produce search states that will not always necessarily improve the solution. The acquisition of knowledge in this case, however, is much simpler than the solution improvement strategy. On the other hand, backtracking control is more complex to implement as it involves the recovery of a previous search state and readjusting the context accordingly.

The prototyping of a KBS requires special attention to the choice of a programming language. Key elements in the choice include flexibility, support of various knowledge representation schemes and interface to other programming environments. These reasons make LISP and PROLOG the most popular languages for KBS development. We have chosen PROLOG for the following reasons:

1. Built-in support for predicate calculus and first order logic,
2. built-in search mechanism, and
3. built-in Backtracking.

In addition, the results of investigations of the performance of declarative and procedural languages in optimisation [4] supported PROLOG. PROLOG's built-in predicates may be used as primitives in the representation of knowledge. The search mechanism offered by PROLOG is also an important asset in the fast prototyping of various solving procedures that use depth-first search. PROLOG's built-in backtracking facilitates the generation of an alternative solution on request.

These advantages, however, come at the cost of the following well known disadvantages of PROLOG:

1. Complex program control, and
2. poor data representation for algorithms.

An additional limitation of PROLOG (at least in its defacto standard) is the lack of recovery of prior state of the knowledge base during backtracking. As the description of the problem and of the current context is held in the database of PROLOG and not as arguments to its predicate, a mechanism that is able to keep track of changes performed during the search process is needed. This mechanism will enable the system, when backtracking, to "forget" information learned during the depth-first search. Furthermore, we may wish that the system does not "forget" all the information it learnt as some of it may still be valid even after backtracking and may be computationally expensive to reproduce. Therefore, the "memorisation" mechanism has to tag this information so it is not "forgotten" during backtracking.

A KBS *shell* is a computer program that includes an inference engine, support for knowledge representation and manipulation, a user interface and an explanation system. To build a VLSI CAD application, the "knowledge engineer" needs to extract knowledge (rules) from experts and from the literature, express this knowledge using the KBS shell syntax and then "tune" the rules in order to produce good solutions. Most available shells are severly limited in terms of search mechanisms, knowledge representation and interface to other programming environments. Furthermore, available shells are not equipped with mechanisms to implement backtracking with "memorisation" and "forgetting", and do not offer the variety of knowledge representation schemes (*e.g.* frames, procedures, rules, predicates, *etc*) essential to KBS based VLSI CAD applications.

This paper presents Brel, a KBS shell especially equipped for VLSI CAD systems. Brel has a context recovery system that implements "memorisation" and "forgetting" and supports a wide range of knowledge representation (frames, rules, procedures, first-order logic, *etc*). Brel has been developed using PROLOG, and has successfully been used to implement PIAF, a top-down floorplanning system [7,8,6], TEMPO a formal verification system for asynchronous circuits based upon Temporal Logic and in the development of an Automatic Layout Generation tool.

II. Key Features of Brel

Considering the VLSI CAD characteristics presented above, we find that two important design issues particularly govern an efficient KBS for VLSI CAD: multiple knowledge representation and backtracking.

A. *Knowledge Representation Schemes*

The nature of the VLSI CAD domain requires multiple knowledge representation schemes. As objects in IC design domain might have a large number of details, it is important to have a structure that regroups this data. Such a structure is also useful to home a predefined set of data that characterise an object. An example is a sub-circuit with the different attributes that it might possess, such as: its children (its own sub-circuits), the other sub-circuits with whom it has interconnections, its operation type, a procedure to evaluate its transparency to foreign signals. This knowledge is well suited to a frame representation. On the other hand, there are areas where knowledge is better expressed and formulated with *if then* rules. For example, it is much easier to extract from a designer a piece of knowledge by asking him/her: What would you do if the situation is *such and such*? Human experts find it easier to answer such a question instead of enumerating the states of the reasoning chain behind any of their decisions.

Other forms of knowledge representation such as procedural and declarative are also important in VLSI CAD knowledge representation.

B. *Backtracking and Context Adjustment*

Context adjustment represents an important issue in the design of KBSs where backtracking can take place and a mechanism is needed to put the system in a previously defined state. The nature of the domain makes impractical the consideration of "undoing rules" and a more efficient memory context structure is crucial. In addition, as VLSI CAD involves intense computation, it is appropriate to devise a new context structure that would properly "memorise" and "forget" calculation results, and enhance the system performance. The structure of the current context memory adopted in Brel is based on a dynamic frame system discussed in the next section, which permits an efficient, simple and portable context recovery system.

III. The Structure and Implementation of Brel

The design issues discussed in the previous section motivated the investigation of a KBS shell structure that would match the needs of our application domain. The system structure adopted in Brel (see Figure 1) satisfies these needs.

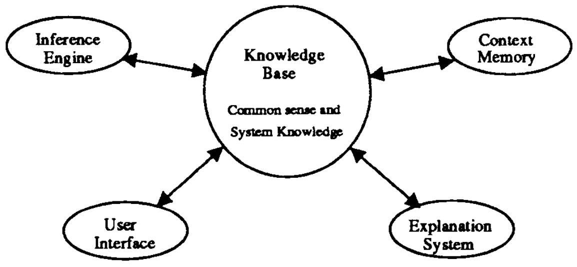

Figure 1: The structure of the Brel system.

In the remainder of this paper, we discuss the structure of Brel and the basic issues that affected its implementation. We will introduce the idea of a quality factor that characterises statements, descriptions and attributes of objects in integrated circuit design. Then we will describe the knowledge representation schemes. We will also discuss the implementation of the inference engine and the current context memory. Due to lack of space the description of the user interface and the explanation system have been omitted and may be found in [5].

A. *Quality Factors*

It is often necessary to classify object attribute values in VLSI CAD. To do so, *Quality Factors* (QFs) which are used to model the degree to which an attribute's value applies to an object. The modelling of the QFs is based mainly on the MYCIN [2] model of Certainty Factors.

B. *Knowledge Representation*

As stated earlier, Brel uses several knowledge representation schemes including predicates, procedures, rules, frames, production rules. A predicate in Brel is a standard PROLOG predicate with a *functor* and *arguments* and represents the primitive representation. Procedures in Brel are represented as PROLOG rules. Brel's frames and production rules are more complicated, and we present them in the following paragraphs.

i. Static and Dynamic Frames

Two types of frames are used by Brel: Static and dynamic. Static frames are used to represent the objects of the domain knowledge that have invariable attributes. Dynamic frames are used to represent objects with attribute values changing during the problem solving process. As mentioned earlier, the introduction of dynamic frames was necessary to implement an efficient memory context structure. The basic difference between the two frames is that the static one is not modified during system operation, while the second is updated every time the value of an object attribute changes. A frame, static or dynamic, has the following information associated with it and stored in the knowledge-base:

Object Class: This states the class to which an object belongs.

Object Identifier: This gives the identifier of the object.

Slots: There is one slot for each attribute, with the form:

- **Attribute:** an attribute name,
- **Value Type:** the class of the attribute value type, and
- **A Value:** the corresponding attribute value.

A frame is implemented as a collection of predicates. These hold five arguments corresponding to the descriptions shown above. This implementation proved the most efficient on a range of PROLOG systems [9,1,10], especially in the case of dynamic frames where alteration of attribute values and consequent update of the database are performed.

ii. Frame Access

The access to an attribute and its value in a frame is accomplished through different types of functions depending on the object type and the access context. These functions are developed around a "core" which is designed for the corresponding PROLOG implementation. This permits us to exploit any database management procedures that may be offered, in addition to the defacto PROLOG standard as described by Clocksin and Mellish [3]. As two types of frames are available, we will concentrate on the functions that access static frames, those corresponding to dynamic frames are similar and will be discussed later in the paper.

General Access to Static Frames: In this type of access, a PROLOG procedure *present* effectuates a blind search for a match in the knowledge base. *present* succeeds and returns the value if the attribute exists and fails otherwise. Figure 2 shows the PROLOG code of the procedure *present* together with an English explanation.

The placement of attribute values in the object frames is performed by the procedure *place* that we show the PROLOG code in Figure 3. Note, again, that this procedure only handles the case of static frames.

Similar procedure are available for deleting attribute values from the frames.

Directed Access to Frames: Another frame access type is a level higher than the one presented above and is based around the procedure *fetch*. This procedure will use *present* first, and if it fails then it generally uses a procedure to guide the system in evaluating the attribute value either with internal calculations or by interrogating the user. This

```
present(S,Id,Field_att,Vt,Value):-
   nonvar(S),
   =..(Q,[fact,S,Id,Field_att,Vt,Value_list]),
   (Q),
   member(Value,Value_list).
```

In the goal above the arguments are:
S: the object class
Id: the object identification
Field_att: the attribute name
Value: either a variable to receive a value or a an actual value
Vt: The type of attribute value

The goal may be used to either:
1- Check if an object has an attribute value as instantiated in Value
2- To retrieve the attribute value of an object
3- To retrieve the identification of an object with instantiated attribute name, value type and attribute value (the Class, S, needs always to be instantiated in this goal which fails otherwise)
4- To retrieve the attribute name or value type, given the other arguments.
The goal works as follows:
First the sub-goal "nonvar(S)" is called and will only succeed if S is instantiated to a value (not a variable). The following goal (starting with "=..") builds a goal predicate (Q) from the object descriptions. Then the built goal is called, and if it succeeds, the arguments passed to this procedure and which are not instantiated will become instantiated (That is, the Value_list variable will receive a value which is a list). Finally the sub-goal "member" will either check that "Value" is member of the list (if Value is instantiated) or instantiate "Value" to a member of the list (the first member if it is the first call to "present" and the following member on backtracking).

Figure 2: The present *frame access procedure.*

is accomplished by asserting at the end of the knowledge-base a PROLOG rule that evaluates the attribute. When interrogating the user, the goal *fetch* will succeed if the user supplies a valid answer and fail otherwise.

Inheritance of attributes (based on *is_a*) is implemented as a directed access with a specialised *fetch* procedure which performs the inheritance mechanism.

Accessing Attributes of Relationships: The access to relationships is done via procedures similar to the two we described above. There are two of them: *present_2* and *fetch_2*.

iii. Brel Production Rules

PROLOG rules, frame functions and facts make the body of a Brel Production Rule (PR) which has the form:

If Old context : W_i **and** A_i
then New context : W_f **and** A_f

where W_i, A_i and W_f, A_f are the resultant Quality Factor (QF, see above) for and against the rule in the old and new context respectively. The resultant QF of a rule is evaluated by taking the minimum of all the individual QFs as only conjunctions are used in the rule expressions. Weighting is also used to permit a priority scheme for conflict resolution.

```
place(S,Id,Field_att,Vt,Value):-
   nonvar(Vt), nonvar(Value), nonvar(S),
   =..(Q,[fact,S,Id,Field_att,Vt,Value_list]),
   ifthenelse((Q),
      {retract_(Q),
      =..(Q1,[fact,S,Id,Field_att,Vt,
         [Value |Value_list]]),
      asserta(Q1)
      },
      {=..(Q1,[fact,S,Id,Field_att,Vt,
         [Value]]),
      asserta(Q1)
      }
   ).
```

The goal above places an attribute value Value *of the object identified* Id *of class* S *to the database. The procedure checks first if the predicate (sub-goal) Q, formed using S, Id, Field_att, Vt and a new variable* Value_list*, succeeds. If it does, then a list of attribute value associated with the object exists already in the database, and the procedure removes the predicate Q from the database and adds a new predicate formed by inserting Value to the begining of the list* Value_list*. Else, the procedure creates a new predicate using the arguments and adds it to the database.*

Figure 3: The place *procedure for attribute value placement.*

C. The Inference Engine

The inference engine is a simple procedure (see Figure 4) which carries out the transformation of the context describing a VLSI CAD problem description state towards a solution.
The procedure *find_all_rules* may succeed twice. In the first time, all rules that match the context and satisfy a unity QF are selected. In the second time, all remaining rules that match the context are collected and sorted according to their decreasing degree of quality using their QFs.

i. Inference Backtracking

As Figure 5 shows, three levels of backtracking may occur. In the first, the system backtracks to the previous inference call to pick the next rule in the list of matching rules, adjusting the context accordingly. In the second backtracking level, the system will backtrack in *find_all_rules* taking the second path in the procedure and finding matching rules with non-unity QF paths. In the third backtracking level, the system will unwind the last recursive call to the procedure itself with a failure which forces a backtracking at the previous call.
Backtracking affects only dynamic attribute values. At each inference, the system carries in the current context memory all information necessary to track its path. The sorting of the rules is based on the arithmetic difference between the "for" QF and "against" QF. The History Tree holds the list of applied rules, and the Search Tree holds the list of matching rules and applied rules.

```
fire_rules(Ow,Oa,Nw,Na):-
   stack(inference,[Task,Q]),
   find_all_rules(Task, Lrules),
   select_rule(Lrules,N1),
   toggle_context,
   update_history(N1),
   update_list(search_tree,[N1,Lrules]),
   apply_rule(N1,Ow,Oa,Nww,Naa),
   ifthenelse(Q,
      {Nw is Nww, Na is Naa,
         message('Tke task ' and Task
         and ' was solved.')},
      fire_rules(Nww,Naa,Nw,Na)).
```

This PROLOG rule has two input and two output arguments representing respectively the initial and final "pro" and "con" QFs of the rule inference. The rule executes as follow:

```
get task informations from stack
find all rules that match current context
select a rule
switch to new context index
update history tree
update search tree
apply the rule
if stop-goal succeeds
   then stop
   else recurse
```

Figure 4: The inference procedure as written in PROLOG. Note that the sub-goal find_all_rules *always succeeds twice, once finding the* unity path *matching rules (with a "pro"= 1) and the other finding all other matching rules.*

ii. Inference Stop-Goal

Another important feature of the inference mechanism is the *stop-goal* marked Q in Figure 4. This goal contains the desired specifications and constraints that the current description of the problem in the context has to meet in order to represent a solution. The stop-goal may also be used to access and modify the current specifications themselves.

D. Current Context Memory Structure

The rule firing procedure shown in Figure 4 shows a call to a goal *toggle_context*. The definition of this goal is shown in Figure 6. The unification mechanism of PROLOG permits this rule to operate in the following way:

1. Increment context index,
2. Any further call to access an object attribute value will push on a stack, of the corresponding current context index, a predicate that states the previous version of the slot describing the attribute,
3. On backtracking, the procedure will treat the stack as a LIFO and will "pop" an element at a time and use it to replace the current value of the object attribute in the context.

Figure 7 shows the path (in thick line) taken by Brel during a context recovery process. We assumed in this figure that

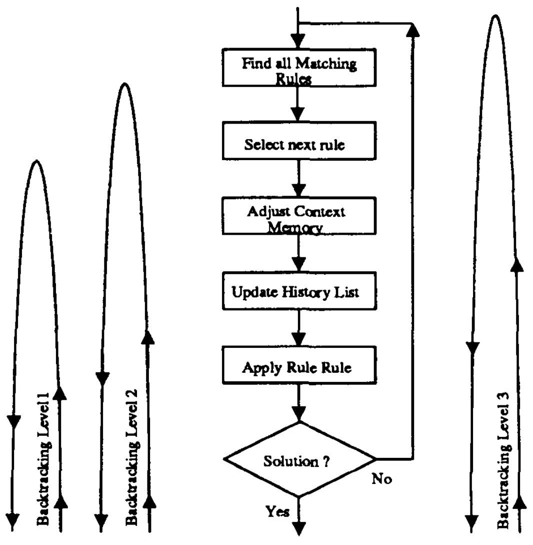

Figure 5: The inference engine procedure and its three backtracking levels.

```
toggle_context:-
   new_id(modif,N),
   message('Context id :  ' and N),
   push(modif,N).
toggle_context:-
   pop(modif,N),
   message('Context backtracking:  '
      and N),
   toggle_context_0(N),
   !, fail.

toggle_context_0(Idc):-
   removeh(modif,Idc,(Idc,S,Id,Att,
      Vt,Val)),
   doall(removeh(S,Id,(Id,Att,Vt,_))),
   ifthen((Val = []),recordh(S,Id,
         (Id,Att,Vt,Val))),
      fail.

   toggle_context_0(_).
```

The above PROLOG rule executes as follow:
The first predicate toggle_context sets up a new context modification index ("modif" stack). On backtracking, the second definition 'pops' the last content index and calls a sub-goal that replaces the old values of object attributes contained in the "modif" stack identified by the 'popped' index.

Figure 6: The context "toggling" procedure.

the *fire_rules* procedure has failed and backtracking is taking place.

i. Context Access Functions

The functions used to access the frames in the CCM are similar to those used for the knowledge base except that they update the context "modif" stack each time they are called. The context access functions are built around *c_pl* shown in Figure 8.

E. "Memorisation" and "Forgetting"

To describe how "memorisation" and "forgetting" are implemented in Brel we will use Figure 7. Suppose that a problem solving task has been run by calling the procedure *fire_rules*. This procedure will get the task name from a stack. It then calls the procedure *find_all_rules* which was described above. Then a rule is selected using the procedure *select_rule*. The *update_history* procedure saves some information for the explanation system (list of rules which have been applied). Then the procedure *toggle_context* creates a new context list to save a copy of any objects attribute values that may change during the application of the rule performed by the procedure *apply_rule*. Once a rule has been applied, transformation of the context would have taken place by the means of changes to attribute values. This transformation corresponds to a "memorisation" phase. Now suppose that at some stage, the system has failed to satisfy a constraint imposed on the design and has started a backtracking process. The first stop on the backtracking path is a "recall" to the *toggle_context* procedure which will perform the "forgetting" mechanism. The exact actions of this procedure were described in Section D. above. In few words, what this procedure carries out is a discriminate recovery of previous attribute values. It is discriminate because some of the attribute values may be computationally expensive to recalculate (like a shortest path in a graph for example) and the rule which has performed the transformation would have used a tag to inform the context toggling procedure about it. After the discriminate recovery of the attribute values, backtracking will continue up to the *select_rule* procedure which will return another rule that matches the readjusted context. Having a new rule, the process restart by updating the rules history list, toggling the context again, applying the new rule and so on...

IV. Results

As mentioned previously, Brel has been used to implement three systems: a floorplanner PIAF [8], TEMPO a formal verification system based upon Temporal Logic and an automatic layout generation system. Among these systems, PIAF is the most mature and is currently being used by circuit designers and students within our department. Its rule based is evolving continuously. The design of PIAF involved the development of rules (over 300) which make intensive use of graph processing algorithms written in Pascal and C. Data access from and to the algorithms is implemented through ASCII files as no standard argument passing mechanism between PROLOG and other high level languages exist.
The use of Brel in the implementation of PIAF has shown considerable advantage in the overall efficiency of the system, as intensive graph processing and optimisation algorithms are used and the availability of the "forgetting" and "memorisation" mechanism minimises the need for recalculation during backtracking.
The production of a prototype KBS CAD tool using Brel involves weeks of programming. The programmer or the

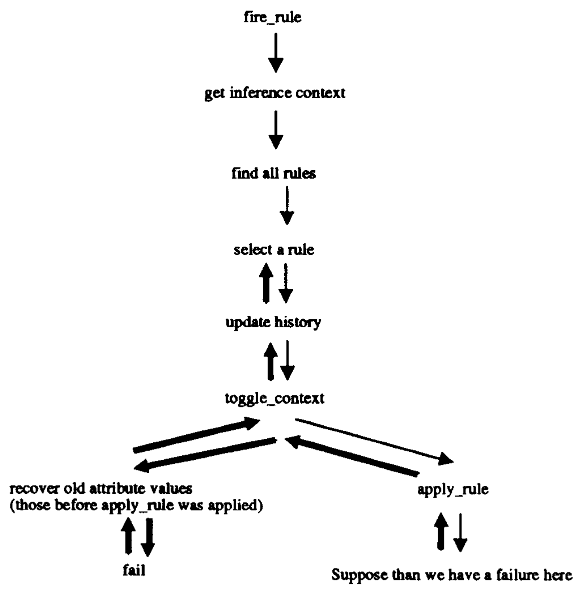
fire_rule
get inference context
find all rules
select a rule
update history
toggle_context
recover old attribute values (those before apply_rule was applied)
apply_rule
fail
Suppose than we have a failure here

Figure 7: In thick line the path taken by Brel during a context recovery process.

knowledge engineer needs only to focus on the development of rules knowing that available to him/her are a wide range of knowledge representation schemes, an efficient backtracking mechanism, a user interface and an explanation system that simplify considerably the amount of programming needed for program control.

V. Conclusion

This paper has presented the concepts behind the design and implementation of Brel. We have also presented its different sub-systems, and in particular, the memory context, the inference engine, the explanation system, and the multi-representation access of the knowledge base including the dynamic and static frames. The structure of these systems appears crucial, and the techniques we have described in this paper enable the design of efficient CAD systems for VLSI development.

References

[1] Arity PROLOG. A compiler/interpreter PROLOG system, Arity Corporation, 358 Baker Av. Ma. 01742, USA. Runs on an IBM PC/XT/AT type machine.

```
c_pl(S,Id,Field_att,Vt,Value):-
   nonvar(Vt), nonvar(Value), nonvar(S),
   ifthenelse(retrieveh(S,Id,(Id,Field_att,
      Vt,Value_list)),
      ifthenelse(member_(Value,Value_list),
         true,
         {list(modif,[Idc|_]),
         removeh(S,Id,(Id,Field_att,Vt,
            Value_list)),
         updatemodif(Idc,S,Id,Field_att,Vt,
            Value_list),
         recordh(S,Id,(Id,Field_att,Vt,[Value|
            Value_list]))
         }
      ),
      { list(modif,[Idc|_]),
      updatemodif(Idc,S,Id,Field_att,Vt,[]),
      recordh(S,Id,(Id,Field_att,Vt,[Value]))
      }
   ).
```

The above PROLOG rule assigns a value "Value" of type "Vt" to the attribute "Field_att" of an object of a class "S" defined by a frame, within the current context.

Figure 8: A PROLOG rule to assign a value to an attribute.

[2] B.G. Buchanan and E. Shortliffe. *Rule-Based Expert Systems.* Addison-Wesley Reading, Mass., 1984.

[3] W.F. Clocksin and C.S Mellish. *Programming in Prolog.* Springer-Verlag, Berlin Heidelberg, 1981.

[4] J.S. Gero and M. Balanchandran. A comparison of procedural and declarative programming languages for the computation of pareto optimal solutions. *Eng. Opt.*, 9:131–142, 1985.

[5] M.A. Jabri. *A Knowledge-Based/Algorithmic Approach to IC Floorplanning.* PhD thesis, Sydney University Electrical Engineering, 1988.

[6] M.A. Jabri. Knowledge-based system design using Prolog: the PIAF experience. *Knowledge-based systems*, 2(1):72–79, March 1989.

[7] M.A. Jabri and D.J. Skellern. A mixed knowledge-based/algorithmic approach to custom integrated circuit floorplanning. In *Proc. IEEE Custom Integrated Circuits Conference*, pages 289–292, 1986.

[8] M.A. Jabri and D.J. Skellern. PIAF: A Knowledge-Based/Algorithmic top-down floorplanning system. In *Proceeding of the 26th ACM/IEEE Design Automation Conference*, pages 582–585, Las Vegas,USA, 1989.

[9] K. Morris and A. Taylor. *Basser Prolog User's Manual.* Technical Report, Basser Department of Computer Science, University of Sydney, Australia, 1984.

[10] Quintus Prolog. A compiler/interpreter PROLOG system, Quintus Computer Systems, inc. Mountain View, California, USA. Runs on VAXs and SUN stations.