Termination Analysis of OPS5 Expert Systems*

Hsiu-yen Tsai **Albert Mo Kim Cheng**

Department of Computer Science
University of Houston
Houston, Texas 77204-3475
Email:(hsiuyen, cheng)@cs.uh.edu

Abstract

Bounded response time is an important requirement when rule-based expert systems are used in real-time applications. In the case the rule-based system cannot terminate in bounded time, we should detect the "culprit" conditions causing the non-termination to assist programmers in debugging. This paper describes a novel tool which analyzes OPS5 programs to achieve this goal. The first step is to verify that an OPS5 program can terminate in bounded time. A graphical representation of an OPS5 program is defined and evaluated. Once the termination of the OPS5 program is not expected, the "culprit" conditions are detected. These conditions are then used to correct the problem by adding extra rules to the original program.

Introduction

As rule-based expert systems become widely adopted in new application domains such as real-time systems, ensuring that they meet stringent timing constraints in these safety-critical and time-critical environments emerges as a challenging design problem. In real applications, rule firings are triggered by the changes in the environment. The computation time of an expert system is highly unpredictable and dependent on the working memory conditions. If the computation takes too long, the expert system may not have sufficient time to respond to the ongoing changes in the environment, making the result of the computation useless or even harmful to the system being monitored or controlled.

To remedy this problem, two solutions are proposed in the literature. The first one is to reduce the execution time via parallelism in the matching phase and/or firing phase of the recognize-act cycle. Several approaches (Ishida 1991; Kuo & Moldovan 1991; Schmolze 1991; Pasik 1992; Cheng 1993) have been provided to achieve this goal. The other solution is to optimize the expert system by modifying or resynthesizing the rule base if the response time is found to be inadequate(Zupan & Cheng 1994). There have been few attempts to formalize the question of whether a rule-based program has bounded response time. Some formal frameworks are introduced in (Browne, Cheng, & Mok 1988; Cheng & Wang 1990; Cheng *et al.* 1993). Their work focus on EQL (Browne, Cheng, & Mok 1988) and MRL (Wang 1990) rule-based languages, which are developed for real-time rule-based applications.

Our work in this paper is related to the second solution. In particular, we shall investigate the timing properties of programs written in the OPS5 language, which is not designed for real-time purposes although it has been widely adopted in practice. Our experience has shown that most rule-based programs are not designed for all possible data domains. Because rule-based programs are data-driven, certain input data are required to direct the control flows in the programs. Many control techniques are implemented in this manner and often require the absence of or a specific ordering of working memory elments to generate initial working memory(WM). Hence, if these WMEs are not in the expected data domain, abnormal program behavior will occur, usually leading to a cycle in the program flow. While we predict the timing bound, termination should be detected as well. Here, we focus on the following points.

- Formalize a graphical representation of rule-based programs.
- Detect the termination conditions of OPS5 programs. In (Ullman 1988), similar work focuses on the recursive relation in backward chaining programs. Here, rule-based programs which employ forward chaining are discussed.
- If an OPS5 program is not detected to terminate for all initial program states, extract the "culprit" conditions which cause non-termination to assist programmers in correcting the program.
- Modify the program to ensure program termination.

The rest of the paper is organized as follows. In Section 2 we define a graph to represent OPS5 programs.

*This material is based upon work supported in part by the National Science Foundation under Award No. CCR-9111563 and by the Texas Advanced Research Program under Grant No. 3652270.

Section 3 introduces a novel method of termination detection. Section 4 describes a technique to find the "culprit" conditions. An additional refinement phase is discussed in Section 5. Section 6 describes how the tool is constructed and provides a brief analysis of its computational complexity. Section 7 is the conclusion.

Static Analysis of Control Paths

Several graphical representations of rule-based programs have been developed for analysis, testing, and debugging purposes. An intuitive representation is a *physical rule flow graph.* In such a graph, nodes represent rules and an edge from node a to node b implies rule b is executed immediately after rule a is executed. Unlike programs written in a procedural language, the control flows of rule-based programs are embedded in the data and cannot be easily derived. Thus one cannot in general find physical paths among rules without running the program for every possible initial program state. Furthermore, since the developer and the tester of a rule-based program usually think in terms of logical paths, a physical rule flow graph is not the most appropriate abstraction. This leads to the definition of a graph called *Enable Rule (ER)* graph, which is adapted from (Cheng & Wang 1990) and (Kiper 1992). The control information among rules in OPS5 is represented by the *ER* graph. To define the *ER* graph, we need to first define the state space graph.

Definition 1 *The state space graph of an OPS5 program is a labeled directed graph G=(V,E). V is a set of nodes each of which represents a set of Working Memory Elements(WMEs). We say that a rule is* **enabled** *at node i iff its enabling condition is satisfied by the WMEs at node i. E is a set of edges each of which denotes the firing of a rule such that an edge (i,j) connects node i to node j iff there is rule R which is enabled at node i, and firing R will modify the Working Memory(WM) to have the same WMEs at node j.*

Definition 2 *Rule a is said to potentially enable rule b iff there exist at least one reachable state in the state space graph of the program where (1) the enabling condition of rule b is false, and (2) firing rule a causes the enabling condition of rule b to become true.*

Since the state space graph cannot be derived without running the program for all allowable initial states, we use symbolic pattern matching to determine the *potentially enable* relation between rules. Rule a potentially enables rule b iff the symbolic form of a WME modified by the actions in rule a matches one of the enabling conditions of rule b. Here, the symbolic form represents a set of WMEs and is of the form:

```
(classname ↑attribute1 v1 ↑attribute2 v2 ...
↑attributen vn)
```

where v1, v2 ... and vn are either variables or constant values and each attribute can be omitted. For example, `(class ↑a1 3 ↑a2 <x>)` can be a symbolic form of the following WMEs.

```
(class ↑a1 3 ↑a2 4)
(class ↑a1 3 ↑a2 8 ↑a3 4)
(class ↑a1 3 ↑a2 <y> ↑a4 <z>)
```

Example 1 illustrates the *potentially enable* relation. Rule a potentially enables rule b because the first action of rule a creates a WME `(class_c ↑c1 off ↑c2 <x>)` which symbolically matches the enabling condition `(class_c ↑c1 <y>)` of rule b. Note that the second action of rule a does not match the first enabling condition `(class_a ↑a1 <x> ↑a2 off)` of rule b because variable `<y>` ranges in `<<open close>>`.

Example 1 An example of the *potentially enable* relation

```
(p a
  (class_a ^a1 <x> ^a2 3)
  (class_b ^b1 <x> ^b2 {<y> << open close >> })
-->
  (make class_c ^c1 off ^c2 <x>)
  (modify 1 ^a2 <y>))
(p b
  (class_a ^a1 <x> ^a2 off)
  (class_c ^c1 <y>)
-->
  (modify 1 ^a2 open))
```

The symbolic matching method actually detects the enabling relation by checking the attribute ranges. This information can be found by analyzing the semantics of the rules.

Definition 3 *The enable-rule (ER) graph of a set of rules is a labeled directed graph $G = (V, E)$. V is a set of nodes such that there is a node for each rule. E is a set of edges such that an edge connects node a to node b iff rule a potentially enables rule b.*

Note that an edge from a to b in the *ER* graph does not mean that rule b will fire immediately after rule a. If rule b is potentially enabled, that only implies rule b may be added to the agenda of the rules to be fired.

The previous analysis is useful since it does not require us to know the contents of working memory, which cannot be obtained statically.

Termination Detection

The *ER* graph provides information about the logical paths of an OPS5 program. We can use this graph to trace the control flows of the program. Since we know the potentially enable relation between rules, we can detect if the firing of each rule in an OPS5 program can terminate. A rule is said to be terminating if the number of that rule's firings is always bounded.

Definition 4 *Suppose rule b potentially enables rule a. Then there is an edge from node b to node a in the ER graph. A matched condition of rule a is one of the enabling condition elements of rule a, which may be matched by executing an action of rule b. Here, rule b is called the enabling rule of the matched condition.*

Definition 5 *An unmatched condition is one of the enabling condition elements of a rule which cannot be matched by firing any rule, including this rule.*

Example 2 *Matched* and *unmatched* conditions

```
(p b                        (p a
  (c1 ^a1 5)                  (c2 ^a2 <x>)
  (c2 ^a2 <x> ^a3 2)          (c3 ^a4 <x> ^a5 <y>)
-->                         -->
  (modify 2 ^a2 3))           (modify 1 ^a2 <y>))
```

In example 2, suppose the firing of any other rule cannot match the second condition element of rule a. In the ER graph, rule b will *potentially enable* rule a. The first condition element (c2 ↑a2 <x>) of rule a is a *matched condition* because it may be matched by firing rule b. The second condition element (c3 ↑a4 <x> ↑a5 <y>) of rule a is an *unmatched condition* because it cannot be matched by firing other rules.

Next, we derive a theorem to detect the termination of a program. One way to predict the termination condition is to make sure that every state in the state space graph cannot be reached twice or more. However, since it is computationally expensive to expand the whole state space graph, we use the ER graph to detect this property.

Theorem 1 *A rule r will terminate if one of the following conditions holds:*

C1. *The actions of rule r modify or remove the unmatched conditions of rule r.*

C2. *The actions of rule r modify or remove the matched conditions of rule r. All of the enabling rules of the matched conditions can terminate in bounded time.*

C3. *Every rule, which enables rule r, can terminate in bounded time.*

Proof:

C1. Since the firing of any rule cannot match the *unmatched conditions*, the only WMEs which can match the *unmatched conditions* are the initial WMEs. Moreover, since the actions of rule r change the contents of these WMEs, the WMEs cannot match the *unmatched conditions* again after rule r is fired. Otherwise, the *unmatched condition* will be matched by firing rule r. This contradicts the definition of *unmatched conditions*. Each initial WME matching the *unmatched condition* can cause rule r to fire at most once since we have a finite number of initial WMEs. Thus rule r can terminate in bounded time.

C2. Since the enabling rules of the *matched conditions* can terminate in bounded time, by removing these rules, the *matched conditions* can be treated as *unmatched conditions*. According to condition 1, rule r can terminate in bounded time.

C3. All rules which enable rule r can terminate in bounded time. After these rules terminate, no other rule can trigger rule r to fire. Thus rule r can terminate as well.

Consider the following rule:

```
(p a
  (c1 ^a1 1 ^a2 <x>)
  (c2 ^a1 4 ^a2 <x>)
-->
  (modify 2 ^a1 3))
```

Suppose the second condition element (c2 ↑a1 4 ↑a2 <x>) cannot be matched by firing any rule, including this rule itself. Then this condition element is an *unmatched condition*. Suppose there are three WMEs in the initial working memory matching this condition element. Then this condition element can be matched by at most three WMEs. The actions of rule a modify these three WMEs when rule a fires. As a result, rule a can fire at most three times.

If there is no cycle in the ER graph or every cycle can be broken (i.e., cycle can be exited), then the firings of every rule in the OPS5 program are finite, and thus termination is detected. However, if the termination cannot be detected, we shall inspect the cycles in the ER graph.

Cycles in the ER Graph

Enabling Conditions of a Cycle

Suppose rules $p_1, p_2 \ldots, p_n$ form a cycle in the ER graph. W is a set of WMEs and W causes rules $p_1, p_2 \ldots, p_n$ to fire in that order. If after firing $p_1, p_2 \ldots, p_n$ in that order will form the WMEs W again, then W is the enabling condition of the cycle. We use symbolic tracing to find W if the data of each attribute are literal. Example 3 illustrates the idea.

Rule p_1 and p_2 form a cycle in the ER graph. To distinguish different variables in different rules, we assign different names to variables. Thus, the program is rewritten as in example 4.

Example 3 Two rules with an embedded cycle

```
(p p1
  (class1 ^a11 { <x> <> 1 } )
  (class2 ^a21 <y>)
-->
  (modify 1 ^a11 <y>))
(p p2
  (class1 ^a11 <x>)
  (class2 ^a21 { <x> << 2 3 >> } ^a22 <y>)
-->
  (modify 1 ^a11 <y>))
```

Example 4 Example 3 with modified variables

```
(p p1
  (class1 ^a11 { <x-1> <> 1 } )
  (class2 ^a21 <y-1>)
-->
  (modify 1 ^a11 <y-1>))
(p p2
  (class1 ^a11 <x-2>)
  (class2 ^a21 { <x-2> << 2 3 >> } ^a22 <y-2>)
-->
  (modify 1 ^a11 <y-2>))
```

A symbol table is built for each variable, which is bound according to the semantics of the enabling conditions. Here, the symbol table is shown in table 1.

Variable	Boundary
x-1	<>1
y-1	none
x-2	2,3
y-2	none

Table 1.

Variable	Boundary
x-1	<>1
y-1	2,3
x-2	2,3
y-2	none

Table 2.

Variable	Boundary
x-1	<>1
y-1	2,3
x-2	2,3
y-2	<>1

Table 3.

The non-terminating condition W is initially the set of all enabling conditions. Thus W is

```
(class1 ^a11 <x-1>)
(class2 ^a21 <y-1>)
(class1 ^a11 <x-2>)
(class2 ^a21 <x-2> ^a22 <y-2>)
```

Each variable is associated with the symbol table.

Now we trace the execution by firing p_1 first; p_1 enables p_2 by matching the first condition. Since the first condition of rule p_2 can be generated from rule p_1, it can be removed from W. Variable x-2 is now replaced by y-1. W is

```
(class1 ^a11 <x-1>)
(class2 ^a21 <y-1>)
(class2 ^a21 <y-1> ^a22 <y-2>)
```

Since x-2 is bound with 2 and 3, y-1 is bound with the same items. The symbol table is modified as in table 2.

After executing the action of rule p_2, W is now

```
(class1 ^a11 <y-2>)
(class2 ^a21 <y-1>)
(class2 ^a21 <y-1> ^a22 <y-2>)
```

To make this WM trigger p_1 and p_2 in that order again, the WME `(class1 ↑a11 <y-2>)` must match the first condition of p_1. Thus variable y-2 is bound with x-1's boundary. The symbol table is shown in table 3. W is

```
(class1 ^a11 <y-2>)
(class2 ^a21 <y-1>)
(class2 ^a21 <y-1> ^a22 <y-2>)
      where y-2<>1 and y-1=2,3
```

The detailed algorithm for detecting the enabling conditions of cycles is described next.

Algorithm 1 *The Detection of Enabling Conditions of Cycles*

Premise: The data domain of each attribute is literal.

Purpose: Rules $p_1, p_2 \ldots, p_n$ form a cycle in the ER graph. Find a set of WMEs W which fire $p_1, p_2 \ldots, p_n$ in that order such that these firings cannot terminate in bounded time.

1. Assign different names to the variables in different rules.
2. Initialize W to be the set of all enabling conditions of $p_1, p_2 \ldots, p_n$.
3. Build a symbol table for variables. Each variable is bound with the semantics of enabling conditions.
4. Simulate the firing of $p_1, p_2 \ldots, p_n$ in that order. Each enabling condition of rule p_i is matched from the initial WM unless it can be generated from rule p_{i-1}. If the enabling condition element w of rule p_i can be generated by firing p_{i-1}, then remove w from W. Substitute p_{i-1}'s variables v_{i-1} for corresponding variables v_i in p_i. Modify v_{i-1}'s boundary in the symbol table.
5. If p_1's enabling condition elements can be generated by p_n, substitute p_n's variables v_n for corresponding variables v_1 in p_1. Modify v_n's boundary in the symbol table.
6. In steps 4 and 5, while substituting p_{i-1}'s variables for p_is', check the intersection of the boundaries of p_i's and p_{i-1}'s variables. If the intersection is empty, then terminate the algorithm.
7. Suppose W_n is the WM after firing $p_1, p_2 \ldots, p_n$. If W_n can match W, then W is an enabling condition of the cycle $p_1, p_2 \ldots, p_n$.

Note that there can be more than one set of enabling conditions W of a cycle. Hence, by applying the algorithm, we may obtain different Ws.

Prevention of Cycles

After detecting the enabling conditions W of a cycle, we can add an extra rule r' with W as the enabling conditions of r'. By doing so, once the working memory has the WMEs matching the enabling conditions of a cycle, the control flow can be switched from the cycle to r'. In example 3, r' is

```
(p loop-rule1
  (class1 ^a11 { <y-2> <>1 } )
  (class2 ^a21 { <y-1> << 2 3 >> } )
  (class2 ^a21 <y-1> ^a22 <y-2>)
-->
  action ...
```

The action of r' is determined by the application. The simplest way is to halt in order to escape from the cycle.

To ensure the program flow switches out of the cycles, the extra rules r' should have higher priorities than the regular ones. To achieve this goal, we use the MEA control strategy and modify the enabling conditions of each regular rule.

At the beginning of the program, two WMEs are added to the WM and the MEA strategy is enforced.

```
(startup
  ......
  (strategy mea)
  (make control ^rule regular)
  (make control ^rule extra))
```

The condition (control ↑rule regular) is added to each regular rule as the first enabling condition element. (control ↑rule extra) is added to each extra rule as the first enabling condition element too. Since the MEA strategy is enforced, the order of instantiations is based on the recency of the *first* time tag. The recency of the condition (control ↑rule regular) is lower than that of the condition (control ↑rule extra). Thus, the instantiations of the extra rules are chosen for execution earlier than those of the regular rules. Example 5 is the modified result of example 3.

Example 5 The modified result of example 3

```
(startup
  (strategy mea)
  (make control ^rule regular)
  (make control ^rule extra))
(p p1
  (control ^rule regular)
  (class1 ^a11 { <x> <> 1 } )
  (class2 ^a21 <y>)
-->
  (modify 2 ^a11 <y>))
(p p2
  (control ^rule regular)
  (class1 ^a11 <x>)
  (class2 ^a21 { <x> << 2 3 >>} ^a22 <y>)
-->
  (modify 2 ^a11 <y>))
(p loop-rule1
  (control ^rule extra)
  (class1 ^a11 { <y-2> <> 1 } )
  (class2 ^a21 { <y-1> << 2 3 >> } )
  (class2 ^a21 <y-1> ^a22 <y-2>)
-->
  (halt))
```

Usually, applications do not expect cycles embedded in the control paths. Thus, once the entrance of a cycle is detected, the program can be abandoned. Hence, after all cycles in the ER graph are found and extra rules are added, we can guarantee that the program will terminate. However, we can also have exception handling on the action of the extra rules. One way to handle the exception is to remove the WMEs which match the enabling condition of a cycle. In example 5, the action of the extra rule can be (remove 2 3 4). Since the WMEs which match the enabling condition of a cycle are removed, the instantiations in the cycle are also removed. Then other instantiations in the agenda can be triggered to fire.

Program Refinement

The ER graph of a typical OPS5 program is complex and usually contains many cycles. Furthermore, even for a single cycle, there may exist more than one enabling condition to trigger the cycle. This leads to a large number of extra rules in the modified programs and thus reduces their runtime performance. To tackle this problem, redundant conditions and rules must be removed after the modification.

Redundant Conditions

In algorithm 1, after symbolic tracing, some variables will be substituted and the boundaries may be changed too. This may cause subset relationship among the enabling condition elements of a cycle. In an extra rule, if condition element C_i is a subset of condition element C_j, then C_j can be omitted to simplify the enabling condition. In example 5, the condition (class2 ↑a21 <y-1> ↑a22 <y-2>) is a subset of (class2 ↑a21). Hence, (class2 ↑a21) can be omitted.

```
(p loop-rule1
  (control ^rule extra)
  (class1 ^a11 { <y-2> <>1 } )
;   (class2 ^a21 { <y-1> << 2 3 >> } )  ;omitted
  (class2 ^a21 { <y-1> << 2 3 >> ^a22 <y-2>)
-->
  (halt))
```

Redundant Rules

Since each cycle is analyzed independently, the extra rules correspond to cycles with different enabling conditions. If the enabling condition of rule r_i is a subset of the enabling condition of rule r_j, then rule r_i can be removed since firing r_i will definitely fire r_j. The cycling information of rule r_j contains that of rule r_i. Thus, it is sufficient to simply provide more general information. In many cases, if the set of nodes P_i which form a cycle C_i is a subset of the set P_j which form a cycle C_j, then the enabling condition of C_j is a subset of C_i's enabling condition. The situation becomes apparent when the cycle consists of many nodes. Hence, we can expect to remove the extra rules whose enabling conditions are derived from larger cycles.

In the following rules, rule 3 and rule 4 can be removed because their enabling conditions are subsets of the enabling conditions of rule 1 and 2, respectively.

```
(p 1
  (class1 ^a13 { <y-1> <> 1 } )
  (class2 ^a22 <y-1>)
-->
  action ...
(p 2
  (class1 ^a13 { <x-1> <> 1 } )
  (class2 ^a22 <y-1>)
  (class4 ^a41 2 ^a42 <x-3>)
-->
  action ...
(p 3                    ; redundant rule
  (class1 ^a13 { <y-1> << 2 3 >> } )
  (class4 ^a41 { <y-4> <> 1 } ^a42 <y-1>)
  (class2 ^a22 <y-1>)
-->
  action ...
(p 4                    ; redundant rule
  (class1 ^a13 { <x-1> <> 1} )
  (class2 ^a22 { <y-1> << 2 3 >> }
  (class4 ^a41 <y-4> ^a42 <y-1>)
  (class4 ^a41 2 ^a42 <x-3>)
-->
  action ...
```

Implementation

The tool has been implemented on a DEC 5000/240 workstation. Two real-world expert systems are examined. The tool adds one extra rule to the OMS expert system(Barry & Lowe 1990) and 4 extra rules to the ISA expert system(Marsh 1988). Before extra rules are added, these two expert systems have 29 and 15 rules, respectively.

For an OPS5 program with n rules, there are potentially $O(n!)$ cycles embedded in the ER graph. However, in a real application, especially in real-time expert systems, it is unlikely that a cycle contains a large number of nodes. If it is detected that no path contains m nodes in the ER graph, there is no need to test cycles with more than m nodes. This reduces both computational complexity and memory space.

To further reduce the computation time, we can store the path information. If there is no path in the order of executing rules $p_1, p_2 \ldots, p_n$, there is no cycle containing this path. Thus we do not need to examine the cycles with the embedded path. The ER graph actually represents all possible paths between two rules. We can construct a linear list to store all impossible paths with more than two rules. Thus, it is a tradeoff between time and space. In our tool, we store impossible paths with up to nine nodes.

Conclusion

We have presented an approach to detect the termination conditions of OPS5 rule-based programs. A data dependency graph (ER graph) is used to capture all of the logical paths of a rule-based program.

Then this ER graph is used to detect if an OPS5 program can terminate in bounded time. More specifically, our technique detects rules which have a finite number of firings. Once non-termination is detected, we extract every cycle in the ER graph and find the enabling conditions of the cycles. After finding the enabling conditions W of a cycle, rule r' is added with W as the enabling conditions. By doing so, once the working memory has the WMEs matching the enabling conditions of a cycle, the control flow can be switched out of the cycle to r'. However, to ensure the program flow switches to r', the program is modified such that r' has higher priority than the regular rules. The extra rules are further refined to remove redundant conditions and rules.

By providing programmers the "culprit" conditions, extra rules can be added to correct the program. If the cycle is an abnormal situation, we can abandon the task to guarantee the termination of the program. However, if recovery from the cycle is required, these conditions can be used to guide the programmers to correct them.

Ongoing work applies the proposed technique to large rule-based systems to test its efficiency and performance. A tight estimation of execution time also must be resolved so that we can predict more precisely about the timing behavior of OPS5 and OPS5-style rule-based systems in terms of execution time.

References

Barry, M. R., and Lowe, C. M. 1990. Analyzing spacecraft configurations through specialization and default reasoning. In *Proc. of the Goddard Conf. on Space Applications of Artificial Intelligence*, 165–179. NASA.

Browne, J. C.; Cheng, A. M. K.; and Mok, A. K. 1988. Computer-aided design of real-time rule-based decision system. Technical report, Department of Computer Science, University of Texas at Austin. Also to appear in *IEEE Trans. on Software Eng.*

Cheng, A. M. K., and Wang, C.-K. 1990. Fast static analysis of real-time rule-based systems to verify their fixed point convergence. In *Proc. 5th Annual IEEE Conf. on Computer Assurance.*

Cheng, A. M. K.; Browne, J. C.; Mok, A. K.; and Wang, R.-H. 1993. Analysis of real-time rule-based system with behavioral constraint assertions specified in Estella. *IEEE Trans. on Software Eng.* 19(19):863–885.

Cheng, A. M. K. 1993. Parallel execution of real-time rule-based systems. In *Proc. IEEE Intl. Parallel Processing Symposium.*

Ishida, T. 1991. Parallel rule firing in production systems. *IEEE Trans. on Knowledge and Data Eng.* 3(1).

Kiper, J. D. 1992. Structural testing of rule-based expert systems. *ACM Trans. on Software Eng. and Methodology* 1(2).

Kuo, S., and Moldovan, D. 1991. Implementation of multiple rule firing production system on hypercube. *J. Parallel and Distr. Computing* 13(4):383–394.

Marsh, C. 1988. The isa expert system: A prototype system for failure diagnosis on the space station. Mitre report, The MITRE Corp., Houston, TX.

Pasik, A. J. 1992. A source-to-source transformation for increasing rule-based parallelism. *IEEE Trans, on Knowledge and Data Eng.* 4(1).

Schmolze, J. G. 1991. Guaranteeing serizlizable results in synchronous parallel production systems. *J. Parallel and Distr. Computing* 13(4).

Ullman, J. D. 1988. Efficient tests for top-down termination of logical rules. *J. of the ACM* 35(2).

Wang, C.-K. 1990. MRL: The language. Tech. report, University of Texas at Austin, Real-Time Lab, Department of Computer Sciences.

Zupan, B., and Cheng, A. M. K. 1994. Optimization of rule-based expert systems via state transition system construction. In *Proc. IEEE Conf. on Artificial Intelligence for Applications*, 320–326.