ANALYSIS AND SYNTHESIS OF REAL-TIME RULE-BASED DECISION SYSTEMS

Albert Mo Kim Cheng

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-14 April 1991

ANALYSIS AND SYNTHESIS OF REAL-TIME RULE-BASED DECISION SYSTEMS

by

MO KIM CHENG

B.A.C.S. with Highest Honors, M.S.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy in Computer Sciences

THE UNIVERSITY OF TEXAS AT AUSTIN

December 1990

ANALYSIS AND SYNTHESIS OF REAL-TIME RULE-BASED DECISION SYSTEMS

Publication No. ____________________

Mo Kim Cheng, Ph.D.
The University of Texas at Austin, 1990

Supervising Professors: James C. Browne and Aloysius K. Mok

Real-time decision systems (RTDS's) are computer-controlled systems that must react to events in the external environment by performing decision-intensive computation sufficiently fast to meet specified timing and safety constraints. This dissertation investigates a class of these systems where decisions are computed by an *equational rule-based program.* Two fundamental problems are identified: (1) the analysis of rule-based RTDS's in order to verify that the specified timing and safety properties are satisfied *prior to* their execution, and (2) the synthesis of rule-based RTDS's that are guaranteed to meet the specified timing constraints in addition to the safety constraints. Two complementary approaches have been developed to solve the first problem: (1) model checking of the global state transition graph representing the program, and (2) static analysis of the program. These approaches are combined to form the cornerstone of the *General Iterative Analysis Algorithm.* The applicability of this analysis technique is further enhanced by the development of a facility with which the rule-based programmer can specify domain-specific knowledge in the language **Estella** in order to validate the performance of an even wider range of programs. Two approaches also have been identified to tackle the second problem: (1) transforming the given equational rule-based program by adding, deleting, and/or modifying rules, and (2) optimizing the scheduler to select the rules to fire such that the variables in the program will always converge to stable values within the response time constraint. The complexity and size of real-time decision systems often necessitates the use of computer-aided design tools. This dissertation describes a suite of analysis tools based on our theoretical framework which have been implemented to ensure that equational rule-based programs written in the language **EQL** can indeed meet their specified timing constraints.

Table of Contents

List of Figures

Chapter 1

Introduction

Rule-Based Expert Systems are increasingly used to monitor and control the operations of complex real-time systems which require intensive knowledge-decision processing and human expertise. These *embedded AI systems* include airplane avionics (e.g., the *Pilot Associate*-driven aircraft), smart robots (e.g., the Autonomous Land Vehicle), space vehicles (e.g., the NASA Space Shuttle and the planned NASA Space Station), and many safety-critical industrial applications. In addition to functional correctness requirements, these systems often must satisfy stringent performance requirements. These rule-based systems must respond to events in the rapidly changing external environment so that the results of the expert system's computation in each monitor-respond cycle are valid in order to safely operate the real-time system. Based on input sensor values, the computer must make decisions within bounded time to respond to the external environment; the result of missing a deadline may be loss of life and property.

The added complexity of timing requirements makes the design and maintenance of these systems particularly difficult. There have been few attempts to formalize the question of whether rule-based systems can deliver adequate performance in **bounded time**. In this dissertation, we develop a formal framework for answering this important question. Even less explored is the problem of transforming a rule-based program that satisfies the integrity constraints but is not fast enough to meet the timing constraints into one which meets both the integrity and timing constraints. It is our goal to address both the theoretical foundation as well as the experimental aspects of the solutions to these two problems. We shall therefore describe a suite of computer-aided software engineering tools, based on our theoretical framework, which have been designed and implemented to ensure that programs for computing complex decisions in real time can indeed meet their specified timing constraints.

The class of real-time programs that are investigated herein are called *rule-based* **EQL** (*Equational Logic*) *programs*. An **EQL** program has a set of rules for updating variables which denote the state of the physical system under control. The firing of a rule computes a new value for one or more state variables to reflect changes in the external environment as detected by sensors. Sensor readings are sampled periodically. Every time sensor readings are taken, the state variables are recomputed iteratively by a number of rule firings until no further change in the variables can result from the firing of a rule. The equational rule-based program is then said to have reached a *fixed point*. Intuitively, rules in an **EQL** program are used to express the constraints on a system and also the goals of the controller. If a fixed point is reached, then the state variables have settled down to a set of values that are consistent with the constraints and goals as expressed by the rules.

EQL differs from the popular *expert system* languages such as OPS5 in some important ways. These differences reflect the goal of our research, which is not to invent yet another *expert system shell* but to investigate whether and how performance objectives can be met when rule-based programs are used to perform safety-critical functions in real time. Whereas the interpretation of a language like OPS5 is defined by the *recognize-act cycle* ([Forgy 81]), the basic interpretation cycle of **EQL** is defined by fixed point convergence†. It is our belief that the time it takes to converge to a fixed point is a more pertinent measure of the response time of a rule-based program than the length of the

† The fixed point semantics of **EQL** follows closely that of the language **Unity** ([Chandy & Misra 88]). Both **Unity** and the work described herein are part of a coordinated research effort to explore the foundation of programming concurrent systems at the University of Texas at Austin.

recognize-act cycle. More importantly, we do not require the firing of rules that lead to a fixed point to be implemented sequentially; rules can be fired in parallel if they do not interfere with one another. The definition of response time in terms of fixed point convergence is architecture independent and is therefore more robust.

Our work complements the *Variable Precision Logic* (VPL) approach of Michalski and Winston ([Michalski & Winston 86]) and Haddawy ([Haddawy 86], [Haddawy 87]). VPL was introduced as a tool to vary the certainty of an inference to conform to a timing constraint. An inference system for VPL can be regarded as a flexible algorithm which can trade off the certainty of an inference for the time required to achieve it. Our emphasis is, however, on ensuring *before run time* that the desired quality of the decisions made by a rule-based program can indeed be achieved within the time budget available to it. We note that a fixed point in **EQL** may correspond to an acceptable approximation of the system state and not necessarily the exact state. Our problem formulation is sufficiently general to allow for variable precision algorithms.

In view of the safety-critical functions that computers are beginning to be relied upon to perform in real time, it is incumbent upon us to ensure that some acceptable performance level can be provided by a rule-based program, subject to reasonable assumptions about the quality of the input. Research in real-time rule-based systems has started attracting attention in the last few years, e.g., [Benda 87], [Helly 84], [Koch et al 86], [O'Reilly & Cromarty 85], [Laffey et al 88]. However, there have been few attempts to formalize the question of whether rule-based systems can deliver adequate performance in bounded time and to develop formal methods for verifying performance guarantees.

1.1. Major Contributions

The major contributions of this dissertation include:

1. The formalization of real-time rule-based systems based on the following:
   - The definition of a general model of embedded real-time decision systems.
   - The development of a simple rule-based language called **Equational Logic (EQL)**.
   - The notion of the space space of an **EQL** program.

2. The formalization of the analysis problem and the synthesis problem of real-time rule-based systems by:
   - The definition of their theoretical formulations.
   - The analyses of their complexities.

3. The development of an efficient and powerful algorithm called the **General Analysis Algorithm** ([Browne, Cheng & Mok 88]) for solving a large class of analysis problems. This new algorithm incorporates the following features:
   - The identification of **special forms** of rules based on a static analysis of the **EQL** program.
   - The ability to use different methods (state-based or non-state-based) for analyzing different parts of the **EQL** program.
   - A main analysis strategy based on structural induction.
   - The capability of accepting user application-specific knowledge in order to speed up the analysis process.

4. The development of the specification language **Estella** ([Cheng, Mok & Browne 90]) for capturing behavioral constraint assertions of rule-based programs. The applicability of the analysis technique is further enhanced by this facility, with which the rule-based programmer can specify application-specific knowledge as input to the analysis algorithm in order to validate the

performance of an even wider range of programs.

5. The novel approach of applying the method of **Lagrange Multipliers** for solving the *time-budgeting problem*.

6. The implementation of a suite of computer-aided design (CAD) tools for the analysis and synthesis of real-time rule-based **EQL** systems in order to guarantee bounded response time. To show that our suite of CAD tools are practical enough to verify realistic real-time decision systems, we have used them to analyze several rule-based systems for real-time monitoring and control applications ([Cheng & Wang 90]). These rule-based systems include:
   - The Cryogenic Hydrogen Pressure Malfunction Procedure in the Pressure Control System of the Space Shuttle Vehicle ([Helly 84]).
   - The Integrated Status Assessment Expert System (ISA) ([Marsh 88]).
   - The Fuel Cell Monitoring Expert System (FCE) ([Marsh 88]).

A real-time rule-based production system called **MRL** ([Wang, Mok & Cheng 90]) also has been developed based on the framework proposed in this dissertation.

1.2. Dissertation Outline

The dissertation is organized as follows:

- Chapter 2 formulates the model of a real-time decision system, and states the problem and the research goals.
- Chapter 3 describes two examples of our equational rule-based programs.
- Chapter 4 introduces the notion of the state space of an equational rule-based program.

- Chapter 5 studies the theoretical formulation and the complexity of the analysis problem, and presents the general analysis strategy and the technique of identifying special forms of rules with bounded execution time for solving the analysis problem.
- Chapter 6 characterizes classes of **EQL** programs that are analyzable by the general analysis algorithm with recognition of special forms of rules.
- Chapter 7 describes how we increase the applicability of our analysis technique by introducing a facility with which the rule-based programmer can specify application-specific knowledge in the language **Estella** in order to validate the performance of an even wider range of programs.
- Chapter 8 presents the theoretical formulation and the complexity of the synthesis problem, and reports on concrete solution strategies for solving the synthesis problem. It shows how the analysis information derived from the general analysis algorithm can be utilized to obtain an optimal schedule for executing rules in **EQL** rule-based programs.
- Chapter 9 demonstrates the use of a set of computer-aided design (CAD) tools that have been implemented to perform timing analysis and synthesis of real-time rule-based programs.
- Chapter 10 explores possible avenues for future research.
- Chapter 11 is the conclusion of the dissertation.

Chapter 2

The Problem and the Research Goals

The problem is best formulated by first formalizing a model of a real-time decision system. A real-time decision system interacts with the external environment by taking sensor readings and computing control decisions based on sensor readings and stored state information. We can characterize a real-time decision system by the following model with 7 components:

(1) a sensor vector $\overline{x} \in X$,

(2) a decision vector $\overline{y} \in Y$,

(3) a system state vector $\overline{s} \in S$,

(4) a set of environmental constraints A,

(5) a decision map D, $D : S \times X \rightarrow S \times Y$,

(6) a set of timing constraints T, and

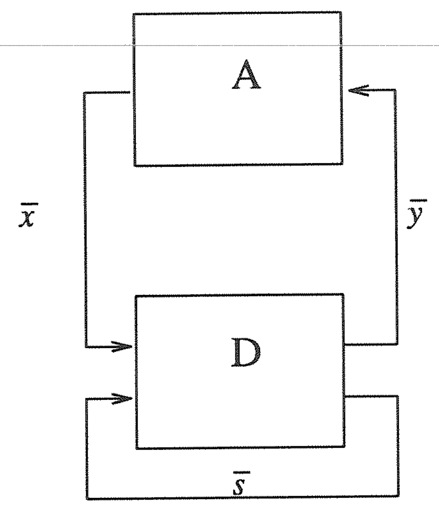
(7) a set of integrity constraints I.

In this model, X is the space of sensor input values, Y is the space of decision values, and S is the space of system state values. (We shall use $\overline{x}(t)$ to denote the value of the sensor input $\overline{x}$ at time t, etc.)

The environmental constraints A are relations over X, Y, S and are assertions about the effect of a control decision on the external world which in turn affect future sensor input values. Environmental constraints are usually imposed by the physical environment in which the real-time decision system functions.

The decision map D relates $\overline{y}(t+1), \overline{s}(t+1)$ to $\overline{x}(t), \overline{s}(t)$ i.e., given the current system state and sensor input, D determines the next decisions and system state values. For our purpose, decision maps are implemented by equational rule-based programs.

The decisions specified by D must conform to a set of integrity constraints I. Integrity constraints are relations over X, S, Y and are assertions that the decision map D must satisfy in order to ensure safe operation of the physical system under control. The implementation of the decision map D is subject to a set of timing constraints T which are assertions about how fast the map D has to be performed. Figure 2.1 illustrates the model of a real-time decision system.



- Environment Constraints:

  A relates $\bar{x}(t+1)$ with $\bar{y}(t)$

- Decision System:

  D relates $\bar{y}(t+1)$, $\bar{s}(t+1)$ with $\bar{x}(t)$, $\bar{s}(t)$

- D is subject to:
  - Integrity Constraints I: Assertions over $\bar{s}$, $\bar{y}$
  - Timing Constraints T

Figure 2.1. A Real-Time Decision System.

Let us consider a simple example of a real-time decision system. Suppose we want to automate a toy race car so that it will drive itself around a track as fast as possible. The sensor vector consists of variables denoting the position of the car and the distance of the next obstacle ahead. The decision vector consists of two variables: one variable to indicate whether to accelerate, decelerate or maintain the same speed, another variable to indicate whether to turn left, right or keep the same heading. The system state vector consists of variables denoting the current speed and heading of the car. The set of environmental constraints consists of assertions that express the physical laws governing where the next position of the car will be, given its current position, velocity, and acceleration. The integrity constraints are assertions restricting the acceleration and heading of the car so that it will stay on the race track and not to run into an obstacle. The decision map may be implemented by some equational rule-based program. The input and decision variables of this program are respectively the sensor vector and decision vectors. The timing constraint consists of a bound on the length of the *monitor-decide cycle* of the program, i.e., the maximum number of rule firings before a fixed point is reached.

There are two fundamental research problems of interest with respect to this model:

(1) Analysis problem: Does a given equational rule-based program satisfy the integrity and timing constraints of the real-time decision system?

(2) Synthesis problem: Given an equational rule-based program that satisfies the integrity constraints but is not fast enough to meet the timing constraints, can we transform the given program into one which meets both the integrity and timing constraints?

In the next chapter, we give two examples of equational rule-based programs. In chapter 4, we investigate these problems by first formulating them in terms of a state-space representation of equational rule-based programs.

Chapter 3

The Rule-Based EQL Language

An **EQL** program has a set of rules for updating variables which denote the state of the physical system under control. The firing of a rule computes a new value for one or more state variables to reflect changes in the external environment as detected by sensors. Sensor readings are sampled periodically. Every time sensor readings are taken, the state variables are recomputed iteratively by a number of rule firings until no further change in the variables can result from the firing of a rule. The equational rule-based program is then said to have reached a *fixed point.* Intuitively, rules in an **EQL** program are used to express the constraints on a system and also the goals of the controller. If a fixed point is reached, then the state variables have settled down to a set of values that are consistent with the constraints and goals as expressed by the rules. A rule-based program is said to always reach a fixed point in **bounded time** iff the number of rule firings needed to take the program from an initial state to a fixed point is always bounded by a fixed upper bound. This bound is imposed by performance constraints.

An **EQL** program consists of a finite set of rules each of which is of the form:

$$a_1 := b_1 \; ! \; a_2 := b_2 \; ! \cdots ! \; a_m := b_m \text{ IF } \textit{test}$$

where b_i is the value to be assigned to variable a_i, and $m \geq 1$. A rule has three parts:

(1) LHS: the left-hand-side of the multiple assignment statement,

(2) RHS: the right-hand-side of the multiple assignment statement, and

(3) EC: the enabling condition (also referred to as the test).

An enabling condition is a predicate on the variables in the program. (Whenever there is no ambiguity, we shall use the terms *enabling condition* and *test* interchangeably.) A rule is enabled if its test becomes true. A rule firing is the execution of the multiple assignment statement of an enabled rule. A multiple assignment statement assigns values to one or more variables in parallel. The RHS expressions must be side-effect free. The execution of a multiple assignment statement consists of the evaluation of all the RHS expressions, followed by updating the LHS variables with the values of the corresponding expressions.

For ease of discussion, we define three sets of variables for an equational rule-based program:

$L = \{ v \mid v$ is a variable appearing in LHS $\}$

$R = \{ v \mid v$ is a variable appearing in RHS $\}$

$T = \{ v \mid v$ is a variable appearing in EC $\}$

An invocation of an equational rule-based program is a sequence of rule firings (execution of multiple assignment statements whose tests are true). When two or more rules are enabled, the selection of which rule to fire is nondeterministic or up to the run-time scheduler, but any rule that stays enabled must eventually be fired. An equational rule-based program is said to have reached a fixed point when either:

(1) none of the rules is enabled, or

(2) the firing of any enabled rule will not change the value of any variable in L.

Intuitively, when a fixed point is reached, a rule-based program has arrived at a consistent evaluation of its environment. It is possible that a program can reach different fixed points starting from the same initial state, depending on which and how rules are fired. This may suggest that the correctness of the program is violated, whereas for some applications this is acceptable.

Some variables appearing in an equational rule-based program are *input variables*, and their values are determined by sensor readings from the external environment at the beginning of each invocation of the program. Input variables do not appear on the left hand side of any assignment statement. The other variables in a program will be called program variables. Program variables are either *decision variables* or *temporary variables*. Decision variables are used to control the physical system and to communicate with the outside world after a fixed point is reached, e.g., signaling system status, giving helpful advice to human operators, etc. Temporary variables are used for storing information about the environment and for communication between rules within the program.

EQL is an equational rule-based language which we have implemented to run under BSD UNIX®. A complete manual for the **EQL** language appears in Appendix A. An example of an equational rule-based program is shown below. The assignment statements are separated by the delimiter character "[]".

Example 3.1.

```
    init: sensor_a_status = sensor_b_status = good

    input: read (sensor_a, sensor_b)

(* 1. *)    object_detected := true  IF sensor_a = 1 AND sensor_a_status = good
(* 2. *) [] object_detected := true  IF sensor_b = 1 AND sensor_b_status = good
```

® UNIX is a registered trademark of AT&T Bell Laboratories.

(* 3. *) [] *object_detected* := false IF *sensor_a* = 0 AND *sensor_a_status* = good
(* 4. *) [] *object_detected* := false IF *sensor_b* = 0 AND *sensor_b_status* = good

In the above example, *sensor_a* and *sensor_b* are the input variables; *object_detected*, *sensor_a_status*, and *system_b_status* are the program variables. For this program, the three sets of variables L, R, T are:

$L = \{$ *object_detected* $\}$,
$R = \emptyset$, and
$T = \{$ *sensor_a*, *sensor_b*, *sensor_a_status*, *sensor_b_status* $\}$.

The decision variable is *object_detected*. In this example, if *sensor_a* and *sensor_b* read in values '1' and '0', respectively, then the above program will never reach a fixed point since the variable *object_detected* will be set to *true* and *false* alternatively by rules 1 and 4. Similarly, if *sensor_a* and *sensor_b* read in values '0' and '1', respectively, then the above program will never reach a fixed point since the variable *object_detected* will be set to *true* and *false* alternatively by rules 2 and 3. In a real-time system, the goal is to have the decision program converge to a fixed point within a bounded number of rule firings. To ensure that the above decision system will converge to a fixed point given any set of sensor input values, some additional information may be needed to settle the conflicting sensor readings. For example, the following rules may be added to the above program:

```
(* 5. *) [] sensor_a_status := bad
            IF sensor_a <> sensor_c  AND  sensor_b_status = good
(* 6. *) [] sensor_b_status := bad
            IF sensor_b <> sensor_c  AND  sensor_a_status = good
```

where *sensor_c* is an additional input variable. If one of the above two rules is fired, then two of the tests (either tests 1 and 3, or tests 2 and 4) in rules 1-4 will be falsified, thus permanently disabling two of those two rules. The variable *object_detected* will then have a stable value since rules 1 and 4, or rules 2 and 3, can no longer fire alternatively. Since the default scheduler of **EQL** will eventually fire an enabled rule, all the variables in the above program will converge to stable values after a finite (but unbounded) number of iterations. In chapter 8, we shall show how this program can be made to converge to stable values in bounded time.

The above example is sufficiently simple that with a little thought, one can understand its behavior and in particular, whether a fixed point can be reached or not. In general, it is non-trivial to determine the behavior of rule-based programs because there is no obvious flow of control. Even small rule-based programs can take quite a bit of work to understand, as the following example illustrates.

The following is an example of a distributed equational rule-based program for determining whether an object is detected at each *monitor-decide cycle*. The system consists of two processes and an external alarm clock which invokes the program by setting the variable *wake_up* to *true* periodically.

Example 3.2.

```
   init: arbiter = a, sync_a = sync_b = true

   input: read(sensor_a, sensor_b)

   (* process A *)
     object_detected := true ! sync_a := false
             IF (sensor_a = 1) AND (arbiter = a) AND (sync_a = true)
  [] object_detected := false ! sync_a := false
             IF (sensor_a = 0) AND (arbiter = a) AND (sync_a = true)
```

```
[] arbiter := b ! sync_a := true ! wake_up := false
            IF (arbiter = a) AND (sync_a = false) AND (wake_up = true)

 (* process B *)

[] object_detected := true ! sync_b := false
            IF (sensor_b = 1) AND (arbiter = b) AND (sync_b = true)
                  AND (wake_up = true)
[] object_detected := false ! sync_b := false
            IF (sensor_b = 0) AND (arbiter = b) AND (sync_b = true)
                  AND (wake_up = true)
[] arbiter := a ! sync_b := true ! wake_up := false
            IF (arbiter = b) AND (sync_b = false) AND (wake_up = true)
```

In this example, the input variables are $sensor\_a$ and $sensor\_b$. The three sets of variables L, R, T for process A are:

$L = \{\ object\_detected, sync\_a, arbiter, wake\_up\ \}$,
$R = \emptyset$, and
$T = \{\ sensor\_a, arbiter, sync\_a, wake\_up\ \}$.

The three sets of variables L, R, T for process B are:

$L = \{\ object\_detected, sync\_b, arbiter, wake\_up\ \}$,
$R = \emptyset$, and
$T = \{\ sensor\_b, arbiter, sync\_b, wake\_up\ \}$.

Each process runs independently of the other. An alarm clock external to the program is used to invoke the processes after some specified period of time. A rule is fired in the same way as in the non-distributed case, namely, the assignment statement is executed when the enabling condition becomes true. In this example, the shared variable *arbiter* is used as a control/synchronization variable which enforces mutually exclusive access to shared variables such as *object_detected* by different processes. The variables *sync_a* and *sync_b* are used as control/synchronization variables within process A and process B, respectively. Note that for each process, at most two rules will be fired before 'control' is transferred to the other process. Initially, process A is given the mutually exclusive access to variables *object_detected* and *sync_a*.

The reader who needs to take considerable effort in understanding the above example need not be discouraged inasmuch as the point of the example is to impress upon the reader the need for computer-aided tools to design this class of programs. We shall discuss a set of computer-aided design tools that have been implemented for this purpose in a later chapter. To investigate the analysis and synthesis problems, we shall first formulate them in terms of a state space representation of equational rule-based programs in the next chapter.

Chapter 4

State Space Representation

The state space graph of an equational rule-based program is a labeled directed graph $G = (V, E)$. V is a set of vertices each of which is labeled by a tuple: $(x_1, \ldots, x_n, s_1, \ldots, s_p)$ where x_i is a value in the domain of the i^{th} input sensor variable and s_j is a value in the domain of the j^{th} program variable. We say that a rule is **enabled** at vertex i iff its test is satisfied by the tuple of variable values at vertex i. E is a set of edges each of which denotes the firing of a rule such that an edge (i, j) connects vertex i to vertex j iff there is a rule R which is enabled at vertex i, and firing R will modify the program variables to have the same values as the tuple at vertex j. Whenever there is no confusion, we shall use the terms state and vertex interchangeably. Obviously, if the domains of all the variables in a program are finite, then the corresponding state space graph must be finite. We note that the state space graph of a program need not be connected.

A **path** in the state space graph is a sequence of vertices $v_1, \ldots, v_i, v_{i+1}, \cdots$, such that an edge connects v_i to v_{i+1} for each i. Paths can be finite or infinite. The length of a finite path $v_1, \ldots, v_k$ is $k-1$. A **simple path** is a path in which no vertex appears more than once. A **cycle** in the state space graph is a path $v_1, \ldots, v_k$ such that $v_1 = v_k$. A path corresponds to the sequence of states generated by a sequence of rule firings of the corresponding program.

A vertex in a state space graph is said to be a **fixed point** if it does not have any out-edges or if all of its out-edges are self-loops (i.e., cycles of length 1). Obviously, if the execution of a program has reached a fixed point, then every rule is either not enabled or its firing will not modify the contents of any of the variables.

An invocation of an equational rule-based program can be thought of as tracing a path in the state space graph. A **monitor-decide** cycle starts with the update of input sensor variables and this puts the program in a new state. A number of rule firings will modify the program variables until the program reaches a fixed point. Depending on the starting state, a monitor-decide cycle may take an arbitrarily long time to converge to a fixed point if at all. We say that a state in the state space graph is **stable** if all paths starting from it will lead to a fixed point. A state is **unstable** if none of the paths from it will lead to a fixed point. A state is **potentially unstable** if there is a path from it which does not lead to a fixed point. By definition, a fixed point is a stable state. It is easy to see that a state s is stable iff any path from s is simple until it ends in a fixed point. A state is potentially unstable iff a cycle is reachable from it or if there is an infinite simple path leading from it.

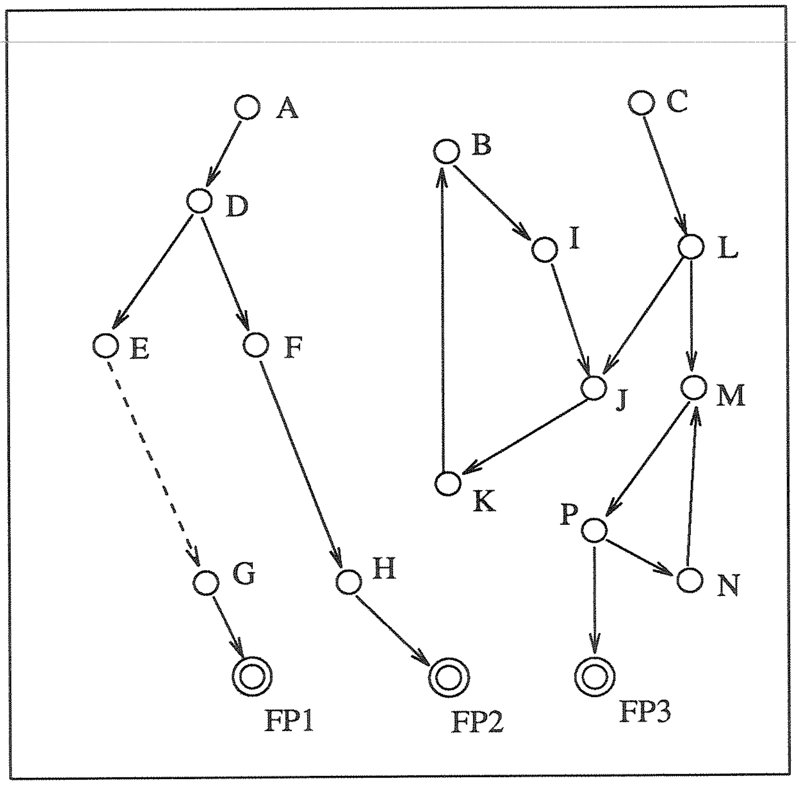


Figure 4.1. State Space Graph of a Real-Time Decision Program.

Figure 4.1 illustrates these concepts. If the current state of the program is A, then the program can reach the fixed point FP2 in 4 rule firings by taking the path (A,D,F,H,FP2). If the path (A,D,E,...,FP1) is taken, then the fixed point FP1 will be reached after a finite number of rule firings. The dotted arrow from E to G in the graph represents a sequence of an unspecified number of uniquely labeled states. State A is stable because all paths from A will lead to a fixed point. If the current state of the program is B, then the program will iterate forever without reaching a fixed point. All the states {B,I,J,K} in the cycle (B,I,J,K,B) are unstable. Note that there is no out-edge from any of the states in this cycle. Once the program enters one of these states, it will iterate forever. If the current state of the program is C, then the program may enter and stay in a cycle if the path (C,L,J,...) is followed. If the path (C,L,M,...) is taken, then the cycle (M,P,N,M) may be encountered. The program may eventually reach the fixed point FP3 if at some time the scheduler fires the rule corresponding to the edge from P to FP3 when the program is in state P. To ensure this, however, the scheduler must observe a *strong fairness* criterion: if a rule is enabled infinitely often, then it must be fired eventually. In this case, paths from state C to FP3 are finite but their lengths are unbounded. C is a potentially unstable state.

In designing real-time decision systems, we should never allow an equational rule-based program to be invoked from an unstable state. Potentially unstable states can be allowed only if an appropriate scheduler is used to always select a sufficiently short path to a fixed point whenever the program is invoked from such a state. We say that a fixed point is an **end-point** of a state s if that fixed point is reachable from s. It should be noted that not every tuple which is some combination of sensor input and program variable values can be a state from which a program may be invoked. After a program reaches a fixed point, it will remain there until the sensor input variables are updated, and the program will then be invoked again in this new state. The states in which a program is invoked are called **launch states**. Formally, we define a launch state† as follows:

(1) The initial state of a program is a launch state.
(2) A tuple obtained from an end-point (which is a tuple of input and program variables) of a launch state by replacing the input variable components with any combination of input variable values is a launch state.
(3) A state is a launch state iff it can be derived from rule (1) and (2).

The above definition of a launch state is conservative in the sense that not all combinations of input variables need to be considered in the construction of launch states because future sensor readings are restricted by the environmental constraints. However, since environmental constraints are necessarily approximations of the external world, it seems prudent not to include them in the definition of a launch state. We emphasize that it is possible to take advantage of environmental constraints in cutting down the number of launch states for analysis purposes.

In this dissertation, the timing constraint of interest is a deadline which must be met by every monitor-decide cycle of an equational rule-based program. In terms of the state space representation, the timing constraint imposes an upper bound on the length of paths from a launch state to a fixed point. Integrity constraints are assertions that must hold at the end-points of launch states. Given a program which meets the integrity constraints but violates the timing constraint, the synthesis problem is to transform this program into one that also meets the timing constraint. This can be done by program transformation techniques and/or by customizing the scheduler to fire the rules selectively so that an invocation always fires the rules corresponding to the shortest path from the launch state to an end-point.

Chapter 5

The Analysis Problem

In this chapter, we study the analysis problem and discuss efficient techniques for tackling the problem.

5.1. Introduction

The analysis problem is to decide whether a given real-time equational rule-based program meets the specified timing constraints as well as integrity constraints. Since the formulation of our problem is in terms of state-space graphs, our approach is compatible with the semantics of temporal logic: in spite of the many variations of temporal logic, their semantics are usually defined in terms of Kripke (state space) structures. Hence, the issue of verifying that an equational rule-based program meets a set of integrity constraints can be treated by appealing to temporal logic techniques in a straightforward manner. Since the application of temporal logic to program verification is quite well understood, it suffices to note that we have integrated a model checker for the temporal logic CTL [Clarke, Emerson & Sistla 86] into our suite of computer-aided design tools. The focus of this research is on meeting timing constraints.

There are many types of timing constraints in real-time decision systems. The fundamental requirement is to be able to bound the response time of the decision system. We capture the response time of an equational rule-based program by the length of the monitor-decide cycle, i.e., the time it takes for all the program variables to settle down to stable values. Technically, the analysis problem of interest is to decide whether a fixed point can always be reached from a launch state on any sufficiently long but finite path. In general, the analysis problem is undecidable if the program variables can have infinite domains, i.e., there is no general procedure for answering all instances of the decision problem.

5.1.1. The Complexity of the Analysis Problem

The undecidability result follows from the observation that any two-counter machine can be encoded by an equational rule-based program that uses only '+' and '–' as operations on integer variables and '>', '=' as atomic predicates such that a two-counter machine accepts an input if and only if the corresponding equational rule-based program can reach a fixed point from an initial condition determined by the input to the two-counter machine. Since two-counter machines can simulate arbitrary Turing machines, our analysis problem is equivalent to the Turing machine halting problem. We illustrate the idea of the proof by exhibiting a two-counter machine (Figure 5.1) and the corresponding equational rule-based program.

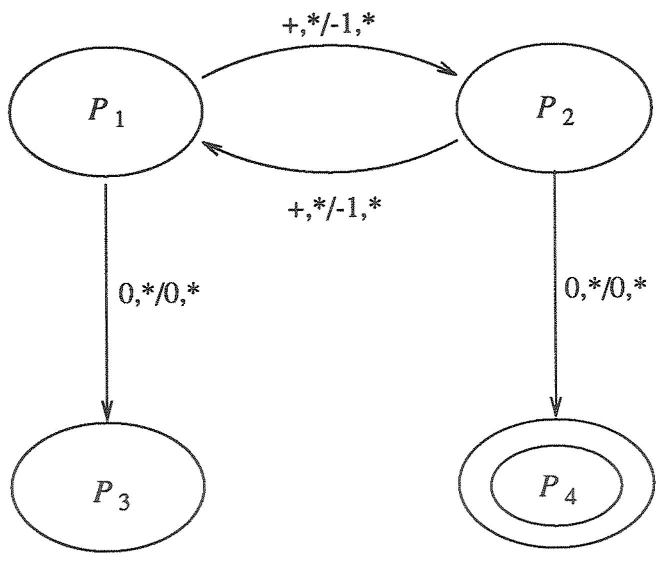


Figure 5.1. A Two-Counter Machine for Testing Odd Input.

This two-counter machine accepts the integer input in its first register iff it is odd. The same input integer is used to initialize the variable c_1 in the program below. The variables s, and f are used to keep track of respectively the current state of the two-counter machine and whether the two-counter machine has entered an accepting state. Notice that the program below reaches a fixed point iff only rule 5 is enabled.

Equational rule-based program for simulating the two-counter machine of Figure 5.1.

init: $s = 1, c_1 = \text{INPUT}, c_2 = 0, f = 0$

```
(* 1. *)    $s := 2 ! c_1 := c_1 - 1 ! f := f + 1$  IF $s = 1$ AND $c_1 > 0$
(* 2. *) [] $s := 3 ! c_1 := c_1 ! f := f + 1$  IF $s = 1$ AND $c_1 = 0$
(* 3. *) [] $s := 3 ! c_1 := c_1 ! f := f + 1$  IF $s = 3$
(* 4. *) [] $s := 4 ! c_1 := c_1 ! f := f + 1$  IF $s = 2$ AND $c_1 = 0$
(* 5. *) [] $s := 4 ! c_1 := c_1 ! f := f$  IF $s = 4$
(* 6. *) [] $s := 1 ! c_1 := c_1 - 1 ! f := f + 1$  IF $s = 2$ AND $c_1 > 0$
```

5.1.2. Finite Domains

Even though the analysis problem is undecidable in general, it is trivially true that the analysis problem is decidable if all the variables of an equational rule-based program range over finite domains. In this case, the state space graph of the program must be finite and can thus be analyzed by an algorithm which performs an exhaustive check on the finite graph. In chapter 9, we will describe a suite of tools for analyzing equational rule-based programs. The default approach there is to generate the reachability graph from the initial (launch) state and use the model checker to determine whether a fixed point is always reachable on any path from the initial state. (Fixed points are expressed by an atomic predicate on a state which is true if and only if out-edges from the state are self-loops.) This approach is viable if the state space graph is reasonably small, but in the worst case may require exponential computation time as a function of the number of variables in the program. More precisely, it can be shown that the computational complexity of the analysis problem restricted to finite graphs is PSPACE-complete ([Mok 89]).

5.1.2.1. Analysis Example

We now describe how our state-based tools can be applied to the distributed **EQL** program of example 3.2. A complete description of these analysis tools will be presented in chapter 9. With the EQL-to-C translator **eqtc**, we can translate the equational rule-based program in example 3.2 into a C program by invoking the command:

eqtc < example3.2 > example3.2.c.

This program can be compiled using a C compiler (the **cc** command in Unix) and then executed to obtain the stable output values if a fixed point is reachable in a finite number of iterations. The current version of **eqtc** simulates the reading of external sensors by initializing the input variables before any rule is fired. The C program generated by the **eqtc** translator is shown below.

```
#include <stdio.h>
#include "scheduler.c"

#define maxseq 24

#define false 0
#define true 1
#define a 0
#define b 1
int znext,
    randseq[maxseq],
    counter;

main() {
      extern int znext,
          randseq[maxseq],
          counter;
int i;

int sync_a, sync_b, wake_up, object_detected;
int arbiter;
int sensor_a, sensor_b;

             sync_a = true;
             sync_b = true;
             wake_up = true;
             arbiter = a;
             sensor_a = 1;
             sensor_b = 0;

init_random_seq(randseq, &znext, z0, &counter);
while (!fixed_point())
{      i = schedule(randseq, &znext, 6);
       switch(i) {
       case 1:
             if ((sensor_a == 1) && (arbiter == a) && (sync_a == true)
                   && (wake_up == true)) {
                   object_detected = true;
                   sync_a = false;
             }
             break;
       case 2:
             if ((sensor_a == 0) && (arbiter == a) && (sync_a == true)
                   && (wake_up == true)) {
                   object_detected = false;
                   sync_a = false;
             }
             break;
       case 3:
             if ((arbiter == a) && (sync_a == false) && (wake_up == true)) {
                   arbiter = b;
                   sync_a = true;
                   wake_up = false;
             }
             break;
       case 4:
             if ((sensor_b == 1) && (arbiter == b) && (sync_b == true)
                   && (wake_up == true)) {
                   object_detected = true;
                   sync_b = false;
             }
             break;
       case 5:
             if ((sensor_b == 0) && (arbiter == b) && (sync_b == true)
                   && (wake_up == true)) {
                   object_detected = false;
                   sync_b = false;
             }
             break;
       case 6:
             if ((arbiter == b) && (sync_b == false) && (wake_up == true)) {
                   arbiter = a;
                   sync_b = true;
                   wake_up = false;
```

```
                        }
                        break;
                }
                printf(" object_detected = %d\n", object_detected);
        }
        printf(" object_detected = %d\n", object_detected);
}
```

The **EQL** program with the initial input values can be translated into a finite state-space graph by using the **ptf** translator with the command:

ptf < example3.2.

ptf generates the following output for user reference:

```
Finite State Space Graph Corresponding to Input Program:
---------------------------------------------------------

state  next states
-----  -----------

rule #1 2 3 4 5 6
0:     1 0 0 0 0 0
1:     1 1 2 1 1 1
2:     2 2 2 2 2 2

State Labels:
-------------

state  (sync_a, sync_b, wake_up, object_detected, arbiter, sensor_a, sensor_b)

0      1 1 1 0 0 1 0
1      0 1 1 1 0 1 0
2      1 1 0 1 1 1 0
```

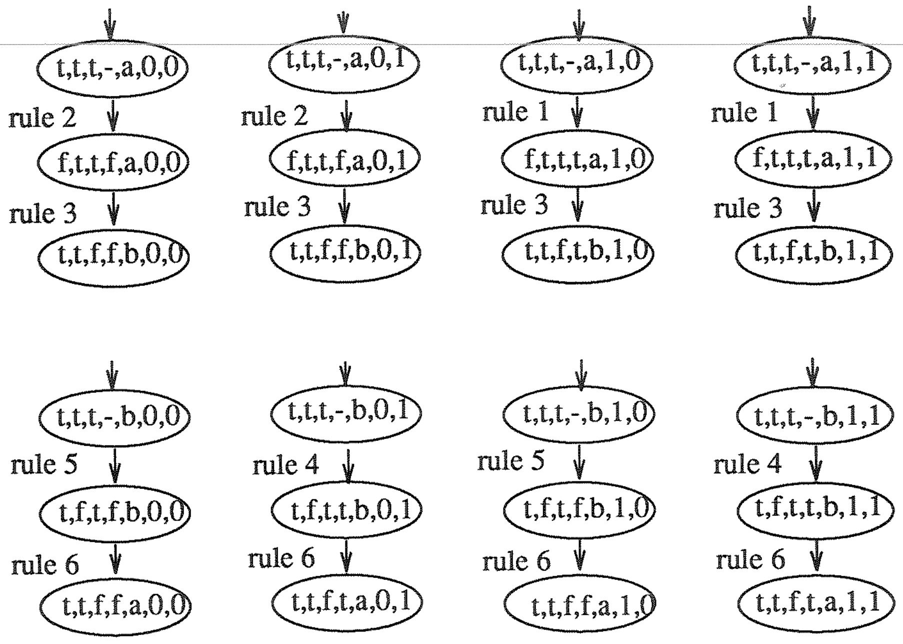
ptf also generates a CTL temporal logic formula for checking whether this program will reach a fixed point in finite time from the launch state corresponding to the initial input and program variable values. This formula is stored in the file *mc.in* which is generated as input to the model checker and the timing analyzer. *mc.in* contains the adjacency matrix representation of the labeled state space graph.

```
3
1 1 0
0 1 1
0 0 1
0 n1 ;
1 n1 ;
2 f1 ;
(au n1 f1)
0
```

The temporal logic model checker **mcf** can then be used to determine whether a fixed point is always reachable in a finite number of iterations by analyzing this finite state-space graph with the given launch state:

mcf < mc.in.

To verify that the program will reach a fixed point from any launch state, the (finite) reachability graph of every launch state must be analyzed by the model checker. The complete state-space graph of the example **EQL** program which consists of eight separate finite reachability graphs, one for each distinct launch state, is shown in Figure 5.2. The graph with launch state (t,t,t,-,a,0,1), corresponding to the combination of input values and initial program values specified in the C program, is one of $2^3 = 8$ possible graphs that must be checked by the model checker.



state = ($sync\_a$, $sync\_b$, $wake\_up$, $object\_detected$, $arbiter$, $sensor\_a$, $sensor\_b$)

t = TRUE, f = FALSE, a = name of process A, b = name of process B, - = don't care

Figure 5.2. Complete State-Space Graph of Program example3.2.

In general, for a finite-domain **EQL** program with n input variables and m program variables, the total number of reachability graphs that have to be checked in the worst case (i.e., all combinations of the values of the input and program variables are possible) is $(\prod_{i=1}^{i=n} |X_i| \cdot \prod_{j=1}^{j=m} |S_j|)$ where $|X_i|$, $|S_j|$ are respectively the size of the domains of the i^{th} input and j^{th} program variable. If all variables are binary, then this number is 2^{n+m}. In practice, the number of reachability graphs that must be checked is substantially less because many combinations of input and program variable values do not constitute launch states. Other techniques are also available that do not require examination of the entire state-space graph. They will be discussed in the next section.

Finally, the timing analyzer **fptime** can be invoked to determine the longest sequence of rule firings leading to a fixed point, if at least one exists, by the command:

fptime < mc.in.

The following is the partial output of the **fptime** module corresponding to the reachability graph with launch state (t,t,t,-,a,0,1):

```
> initial state:     0
> fixed-point(s):
> 2
> initial state: 0     fixed-point: 2
> maximum number of iterations: 2
> path:      0    1    2
```

The module **ptaf** performs the above translation and analysis on the complete state-space graph of the example **EQL** program automatically. The command:

ptaf < example3.2

produces the following messages:

```
> The program always reaches a fixed point in finite time.
> The maximum number of iterations to reach a fixed point is 2.
> 8 FSMs checked.
```

5.1.3. Special Forms of Rules with Bounded Execution Time

It should be emphasized that in practice, it is often not necessary to check the complete state space in order to solve the analysis problem. Under appropriate conditions, efficient procedures exist which can be applied to reduce the size of the state space by a simple textual analysis of the program. In particular, rules of certain forms are always guaranteed to reach a fixed point in a finite number of iterations. Four of these special forms which are especially useful are presented in this chapter. We shall show later that it is unnecessary for all the rules of a program to be in a special form in order to be able to reduce the state space. Techniques exist that can be applied recursively to fragments of a program and the result used to transform the whole program into a simpler one. First, some definitions are in order.

We have defined in chapter 3 three sets of variables for an equational rule-based program. They are repeated below for convenience.

$L = \{ v \mid v$ is a variable appearing in LHS $\}$

$R = \{ v \mid v$ is a variable appearing in RHS $\}$

$T = \{ v \mid v$ is a variable appearing in EC $\}$

Let $T = \{ v_1, v_2, ..., v_n \}$ and let $\bar{v}$ be the vector $<v_1,v_2,...,v_n>$. With this definition, each test (enabling condition) in a program can be viewed as a function $f(\bar{v})$ from the space of $\bar{v}$ to the set $\{ \mathit{true}, \mathit{false} \}$. Let f_a be the function corresponding to the test a and let V_a be the subset of the space of $\bar{v}$ for which the function f_a maps to *true*. Let $V_{a,i}$ be the subset of the values of v_i for which the function f_a *may* map to *true*; that is, if the value of the variable v_i is in the subset $V_{a,i}$, then there exist an assignment of values to the variables in the set $T - \{v_i\}$ such that the function f_a maps to *true*. Note that if the variable v_k does not appear in the test a, then $V_{a,k}$ is the entire domain of v_k. We say that two tests a and b are **mutually exclusive** iff the subsets V_a and V_b of the corresponding functions f_a, f_b are disjoint. Obviously, if two tests are mutually exclusive, then only one of the corresponding rules can be enabled at a time.

For some rules, it is straightforward to determine if two tests are mutually exclusive. For example, consider tests of the form:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each C_i is a predicate of the form:

$$<variable> \ <relational\ operator> \ <constant>$$

For a test a of this form, it is easy to see that the subset V_a of the space of $\bar{v}$ for which f_a maps to *true* can be expressed as the cross product:

$$V_{a,1} \times V_{a,2} \times \cdots \times V_{a,n}$$

such that $f_a(\bar{v})$ maps to *true* iff the i^{th} component of $\bar{v}$ is in $V_{a,i}$, for $i=1,...,n$. Note that if the variable v_k does not appear in the test a, then $V_{a,k}$ is the entire domain of v_k. To verify that two tests a and b are mutually exclusive, it suffices to find at least one variable v_i for which $V_{a,i} \cap V_{b,i} = \emptyset$. If no such v_i is found, then the two tests are not mutually exclusive.

Let L_x denote the set of variables appearing in LHS of rule x. Two rules a and b are said to be **compatible** iff at least one of the following conditions holds:

(CR1) Test a and test b are mutually exclusive.
(CR2) $L_a \cap L_b = \emptyset$.
(CR3) Suppose $L_a \cap L_b \neq \emptyset$. Then for every variable v in $L_a \cap L_b$, the same expression must be assigned to v in both rule a and b.

Let T_x denote the set of variables appearing in the test (enabling condition) of rule x. We are now ready to present several special forms of rules with bounded execution time for which the analysis problem can be solved efficiently.

5.2. Special Form A: Compatible Assignment to Constants, L and T Disjoint

A set of rules are said to be in special form **A** if all of the following three conditions hold.

(A1) Constant terms are assigned to all the variables in L, i.e., $R = \varnothing$.
(A2) All of the rules are compatible pairwise.
(A3) $L \cap T = \varnothing$.

Theorem 5.2.1.

An **EQL** rule-based program whose rules are in special form **A** will always reach a fixed point in at most n iterations, where n is the number of rules in the program.

Proof:

The fact that L and T are disjoint means that the logical value of every test will remain constant throughout an invocation once all sensor readings have been taken and assigned to the input variables. Thus condition A3 implies that a rule is either enabled or disabled throughout an invocation of the program. So we only need to focus on the set of rules which are enabled. If condition CR1 holds for every pair of rules, then at most one of the rules is enabled at any invocation, and since assignments are always to constants, the program will reach a fixed point in one iteration. If condition CR2 holds, then every variable appears at most once in LHS of the enabled rules. Hence, there is at most one constant that can be assigned to any particular variable, and the program must reach a fixed point after all the enabled rules have been fired. If two or more enabled rules can be fired which assign to the same variable, condition CR3 guarantees that they will assign the same value to the variable, and again the program must reach a fixed point after all the rules have been fired. Obviously, the number of iterations before reaching a fixed point is bounded by the number of rules in the program. (This assumes that the scheduler must not execute a rule more than once if no variable will be changed by executing the rule.) Tighter bounds are possible by taking into account rules whose tests are mutually exclusive.

$\square$

5.2.1. Recognition Procedure for Special Form A

Given a set S of rules and a set A (used to hold rules in special form A) initialized to the empty set, the current implementation of the recognition procedure incrementally adds a rule i to A if the set $A \cup \{i\}$ is in special form A. To check the satisfiability of the condition A1, the recognition algorithm checks the expressions assigned to each variable in the set L_i of the rule i against the table of internal and declared constants. To check the satisfiability of the condition A2, the recognition algorithm checks rule i with every rule already in the set S for compatibility (by conditions CR1, CR2 or CR3). Finally, to check the satisfiability of the condition A3, the recognition algorithm checks whether L_i and T_A are disjoint and whether L_A and T_i are disjoint, where L_A and T_A are respectively the sets of variables in the LHS and the test parts of those rules already in A.

In order for a rule i to be added to the set A, all three conditions A1-A3 must be satisfied. The recognition algorithm terminates when all rules in the set S are checked. The output from the algorithm is the set A.

5.2.2. Complexity of the Recognition Procedure for Special Form A

It is easy to see that the complexities of the recognition algorithms for checking a set of rules for the satisfiability of the conditions A1, A2:CR2 and A2:CR3, and A3 are respectively $O(n)$, $O(n^2)$, and $O(k^2)$, where n is the number of rules in the set, and k is the number of variables appearing in set L. Note that in the worst case, the checking of condition A2:CR1 (mutual exclusion) may require exponential time. However, programmers do not write unstructured tests in practice and many pairs of rules are compatible by conditions CR2 or CR3 (which are checked first by the recognition procedure), the checking of condition CR1 is usually not a problem.

5.2.3. Application of Special Form A

To illustrate the application of the special form A, consider the programs in examples 5.2.3.1-5.2.3.4 below. In example 5.2.3.1, the two rules in the program are compatible by CR1 and thus the program is always guaranteed to reach a fixed point in bounded time.

In example 5.2.3.2, even though the two tests in this program are not mutually exclusive because ($b = c = true$) is true in both test 1 and test 2, the fact that all LHS variables are distinct makes the rules compatible (condition CR2 is satisfied) and thus is sufficient to guarantee that this program will reach a fixed point in a bounded number of iterations.

In example 5.2.3.3, test 1 and test 3 are not mutually exclusive. However, rule 1 and rule 3 are compatible by condition CR2. Rule 2 and rule 3 are compatible because their tests are mutually exclusive, and so are rule 1 and rule 2. Thus, this program will reach a fixed point in bounded time.

Finally, consider example 5.2.3.4. Note that the same value (the constant *true*) is assigned to the variable $a1$ which appears in both the LHS of rules 1 and 2. Hence, condition CR3 is satisfied and thus the rules are compatible; hence, this program is guaranteed to reach a fixed point in bounded time.

Example 5.2.3.1. (The two rules in this program satisfy condition CR1.)

```
        input: read($b$, $c$)

1.    $a1$ := true  IF $b$ = true AND $c$ = false
2.  [] $a2$ := false IF $b$ = true AND $c$ = true
```

Example 5.2.3.2. (The two rules in this program satisfy condition CR2.)

```
        input: read($b$, $c$)

1.    $a1$ := true  IF $b$ = true
2.  [] $a2$ := false IF $c$ = true
```

Example 5.2.3.3. (Rule 1 and rule 3 are compatible by condition CR2.)

```
        input: read($b$, $c$)

1.    $a1$ := true  IF $b$ = true AND $c$ = true
2.  [] $a1$ := true  IF $b$ = true AND $c$ = false
3.  [] $a2$ := false IF $c$ = true
```

Example 5.2.3.4. (The two rules in this program satisfy condition CR3.)

```
        input: read($b$, $c$)

1.    $a1$ := true  IF $b$ = true
2.  [] $a1$ := true  IF $c$ = true
```

5.2.4. Deriving Tight Response Time Bound

We define the **response time** of an EQL rule-based program to be the maximum number of rule firings to take the program from a launch state to a fixed point. Some additional definitions are needed for describing the derivation of the response time bound for executing special form A rules.

Definition 5.2.4.1.

The **mutual exclusion** (ME) graph of a set of rules is a labeled undirected graph $G = (V, E)$. V is a set of vertices each of which represents a rule. E is a set of edges such that an edge (a, b) connects vertex a to vertex b iff rule a and rule b are compatible by mutual exclusivity (condition CR1).

We are now ready to derive a formula for a tighter response time bound for programs in special form A. For a program in special form A, rules that are enabled (disabled) before the firing of any rule remain enabled (disabled) during the course of rule firings. Theorem 5.2.1 states that an **EQL** program consisting of n rules all of which are in special form A will always reach a fixed point in at most n rule firings. To obtain a more accurate response time bound, we subtract from n the number of rules that are (and remain) disabled during the course of rule firings. Observe that at most one of the rules in a complete **ME** subgraph is enabled at any invocation since condition CR1 holds for every pair of rules in the subgraph. Two complete subgraphs G_a and G_b are said to be **independent** iff $V(G_a) \cap V(G_b) = \emptyset$. Let m_i be the number of nodes in the independent complete subgraph i in the **ME** graph.

Note that an **ME** subgraph may contain only one node since, by definition, a rule is CR1-compatible with itself. Also note that two rules, each from a different, independent complete **ME** subgraph, may be enabled at the same time since these two rules may not be mutually exclusive. For the purpose of deriving the response time bound on the firing of a set of rules, we are interested in finding the *minimal* number of independent complete **ME** subgraphs with at least two vertices. Let k be this number. The reason for minimizing the number of independent complete **ME** subgraphs for consideration is to obtain a tighter response time bound. Following the above reasoning, we derive the following formula:

$$T_A = n - \sum_{i=1}^{k} (m_i - 1)$$

Next, observe that if two or more rules have the same LHS and all these rules assign the same expressions to these same variables, then only one of these rules will fire since firing a second rule in this rule set will not change the content of any variable in LHS of the rule. Let S_j be the set of rules with isomorphic assignments of the form j such that no pair of rules in S_j are compatible by mutual exclusivity (condition CR1) and none of these rules are CR1-compatible with rules not in S_j (e.g. each rule vertex in S_j has no incident **ME** edges). Let p_j be the number of rules in S_j. Note that set S_j may be a singleton. Let l be the number of sets S_j. From this observation, we can refine the above formula to obtain:

$$T_A = n - (\sum_{i=1}^{k} (m_i - 1) + \sum_{j=1}^{l} (p_j - 1))$$

Since

$$\sum_{i=1}^{k} m_i + \sum_{j=1}^{l} p_j = n,$$

therefore,

$$T_A = n - ((\sum_{i=1}^{k} m_i - \sum_{i=1}^{k} 1) + (\sum_{j=1}^{l} p_j - \sum_{j=1}^{l} 1))$$

$$T_A = n - (n - k - l) = k + l$$

Theorem 5.2.4.1.

The response time bound T_A of an n-rule program in special form **A** is given by the following formula:

$$T_A = k + l$$

where k is the minimal number of independent complete **ME** subgraphs each of which contains at least two vertices, and l is the number of sets S_j of rules with isomorphic assignments.

Example 5.2.4.1.

If condition CR1 holds for every pair of rules, then at most one of the rules is enabled at any invocation, and since assignment expressions consist of constants only, the program will reach a fixed point in one rule firing. We obtain the same response time bound by using the above formula,

$$T_A = k + l = 1 + 0 = 1$$

since the minimal number of independent complete **ME** subgraphs is one -- the entire **ME** graph.

Example 5.2.4.2.

```
(* 1 *)    $a$ := 10 IF $x = 1$
(* 2 *) [] $a$ := 20 $x = 2$
(* 3 *) [] $a$ := 30 IF $x = 3$
(* 4 *) [] $a$ := 40 IF $x = 4$
(* 5  *)  [] b := 10 IF y = 1
(* 6  *)  [] b := 20 IF y = 2
(* 7  *)  [] c := 10 IF z = 1
(* 8  *)  [] c := 20 IF z = 2
(* 9  *)  [] c := 30 IF z = 3
(* 10 *)  [] d := 0 IF a = 10
(* 11 *)  [] d := 0 IF b = 20
```

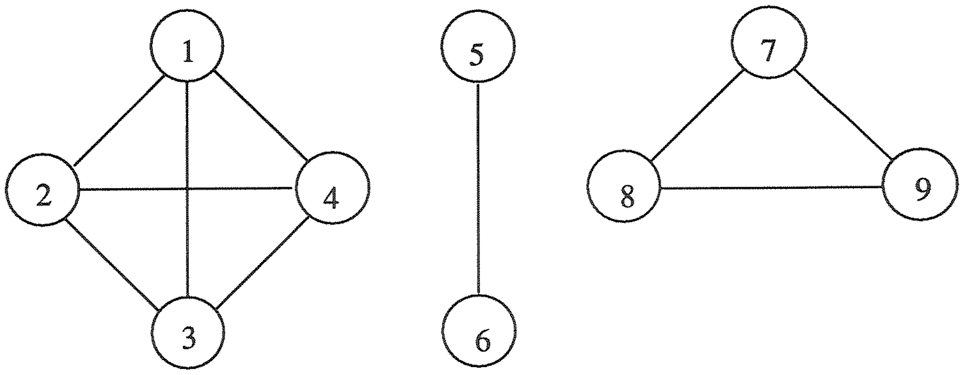


Figure 5.3. ME Graph Corresponding to the Program of Example 5.2.4.2.

In this program, rules 1, 2, 3, and 4 form a complete ME subgraph m_1, so do rules 5 and 6 (subgraph m_2) and rules 7, 8, and 9 (subgraph m_3). Rule 10 and rule 11 assign the same value to the same variable. Using the response time formula for special form **A**,

$$T_A = k + l = 3 + 1 = 4.$$

5.3. Special Form B: Compatible Assignment to Constants, L and T Not Disjoint

Before we can describe special form **B**, we need some additional definitions.

Definition 5.3.1.

Rule a is said to **potentially enable** rule b if there exist at least one reachable state where (1) rule b is disabled, and (2) firing rule a enables rule b.

Definition 5.3.2.

Rule a is said to **disable** rule b if for all reachable program states where rule a and rule b are both enabled, firing rule a disables rule b.

Definition 5.3.3.

Given two distinct rules a and b. Let x_i, $i = 1, \ldots, n$, be variables in the set $L_a \cap T_b$. Rule a is said to **approximately potentially enable** rule b if: rule a assigns the value m_i to variable x_i such that $m_i \in V_{b,i}$, $i = 1, \ldots, n$, and $n \geq 1$.

Note that definition 5.3.1 implies definition 5.3.3. The approximately potentially enable relation can be easily checked in polynomial time whereas the checking of the potentially enable relation may require exhaustive state space search.

Definition 5.3.4.

The **enable-rule (ER)** graph of a set of rules is a labeled directed graph $G = (V, E)$. V is a set of vertices such that there is one vertex for each rule. E is a set of edges such that an edge connects vertex a to vertex b iff rule a

potentially enables rule b.

Definition 5.3.5.

A **cycle** in the **ER** graph is path $v_1, ..., v_k$ such that $v_1 = v_k$. A **simple cycle** is a cycle $v_1, v_2, v_3, ..., v_k$ such that $v_1 = v_k$ and for $i, j = 1, ..., k$ and $i \neq j: v_i \neq v_j$ (i.e., every vertex in the cycle is distinct).

A set of rules are said to be in special form **B** if all of the following four conditions hold.

(B1) Constant terms are assigned to all the variables in L, i.e., $R = \varnothing$.
(B2) All of the rules are compatible pairwise.
(B3) $L \cap T \neq \varnothing$.
(B4) For each cycle in the **ER** graph corresponding to this set of rules, no two rules in the cycle assign different expressions to the same variable. (A cycle is said to be bad if at least two rules in it assign different expressions to the same variable.)
(B5) Rules in disjoint simple cycles (with at least two vertices) in the **ER** graph do not assign different expressions to a common variable appearing in their LHS.

Theorem 5.3.1.

An **EQL** program whose rules are in special form **B** will always reach a fixed point in a bounded number of iterations.

Let **SSG** denote a state-space graph and let **ERG** denote an enable-rule graph. In the ERG, we shall use the terms *rule* and *vertex* interchangeably.

Lemma 1.

If a set of rules satisfy conditions B1, B2 and B3, then each rule in a SSG cycle, if any, must be disabled in some state in the SSG cycle.

Proof.

Assume that there exist a rule r_i in a SSG cycle S that is not disabled in any state in S. There are two cases to consider. (1) After firing r_i, r_i will not fire again unless there is a variable v, $v \in L_{r_i}$, such that v has changed its content since the last firing of r_i. Let e be the expression assigned to v in rule r_i. Let r_j be the rule which assigns an expression e' to v such that $e \neq e'$, and r_j fires sometime after the firing of r_i. Since all rule pairs must be compatible but r_i and r_j do not satisfy compatibility conditions CR2 or CR3, r_i and r_j must be compatible by mutual exclusion (condition CR1). This implies that when r_j is enabled for firing, r_i must be disabled. (2) If r_i does not fire again after firing once, then it cannot be in a SSG cycle. This follows from the fact that when there is an edge (corresponding to a rule r_k) from a state s to another state s' in a cycle or path in the SSG, at least one variable changes its content by firing r_k. Thus it can be concluded that each rule in a SSG cycle must be disabled in some state in the cycle.

$\square$

Let $C_1, C_2, ..., C_m$ be the strongly connected components of $G(V, E)$. Define $\overline{G}(\overline{V}, \overline{E})$ as follows:

$$\overline{V} = \{C_1, C_2, ..., C_m\}$$
$$\overline{E} = \{ (C_i, C_j) \mid i \neq j, (x, y) \in E, x \in C_i \text{ and } y \in C_j \}$$

Lemma 2. $\overline{G}$ is a directed acyclic graph.

Proof.

Assume that $\overline{G}$ is not an acyclic graph. Then $\overline{G}$ has a directed circuit. However, all strongly connected components on it should have been one strongly connected component. Thus $\overline{G}$ must be a directed acyclic graph.

$\square$

Theorem.

If the conditions B1, B2 and B3 of special form **B** are satisfied by a set of rules, if there are no bad cycles in the **ER** graph (condition B4 is satisfied), and if rules in disjoint simple **ER** cycles do not assign different expressions to a common variable appearing in their LHS (condition B5 is satisfied), then there is no cycle in the corresponding state-space graph.

Proof.

To prove this theorem, we shall prove its contrapositive. Given any set of rules, we shall show that if conditions B1, B2, and B3 are satisfied, and there is a SSG cycle, then condition B4 or condition B5 must be violated. Consider a SSG cycle. Label the rules represented by the edges in the SSG cycle $r_1, r_2, ..., r_m$. If a rule is represented by more than one edge in the SSG cycle, it is assigned the same label so that each r_i, $i = 1, ..., m$, corresponds to one distinct rule. Consider the ERG subgraph G' defined as follows:

$$V' = \{r_1, r_2, ..., r_m\}.$$
$$E' = \{(r_i, r_j) \mid r_i \in V' \wedge r_j \in V' \wedge (r_i, r_j) \in E\}.$$

E is the set of all **ER** edges. We shall show that there is at least one strongly connected component in G'. First, we show that vertices with out-edge(s) only cannot exist in G'. Let r_{out} be a vertex with only out-edge(s). There are two cases to consider. (1) If r_{out} is disabled at the start of rule firings, it remains disabled permanently. This follows from the fact that there is no in-edge incident to r_{out} and thus no other rules can enable it. However, each rule in the SSG cycle must become enabled infinitely often so that it can fire infinitely often. Therefore, there is no edge corresponding to r_{out} in the SSG cycle and thus r_{out} cannot be in V'. (2) If r_{out} is enabled at the start of rule firings, then there are two subcases to consider.

(2.1) Suppose r_{out} is compatible with another rule by condition CR1 (mutual exclusion). Let v_i, $i = 1, 2, ..., p$, be the variables in $L_{r_{out}}$, and $\exp_i$ be the expression assigned to v_i. Then there may be a variable v_a, $v_a \in L_{r_{out}}$, such that v_a is assigned $\exp_{a'}$ by another rule r_j, $\exp_a \neq \exp_{a'}$, and r_{out} and r_j are mutually exclusive. Suppose r_{out} fires first. Then r_j becomes enabled. At this point, r_{out} is disabled and stays disabled permanently since it does not have incident in-edge and thus cannot be re-enabled by another rule. Hence, r_{out} can fire at most once and thus cannot appear in the SSG cycle. Thus r_{out} and its incident out-edge(s) cannot exist in G'.

(2.2) If r_{out} is not mutually exclusive with another rule, then it must be compatible with every other rule by condition CR2 or by condition CR3. Then there is no variable v_a, $v_a \in L_{r_{out}}$, such that v_a is assigned $\exp_{a'}$ by another rule and $\exp_a \neq \exp_{a'}$. Since r_{out} is enabled at the start, it stays enabled until it is disabled by another rule, from this point it stays disabled permanently since there is no in-edge incident to r_{out}. Hence, r_{out} can fire at most once and thus cannot appear in the SSG cycle. Thus r_{out} and its incident out-edge(s) cannot exist in G'.

Using the above argument, we can easily show that vertices with no incident edges cannot exist in G'.

Next, we show that vertices with in-edge(s) only can be removed from G' and the corresponding edge(s) in the SSG cycle can be bypassed (defined later) for the purpose of the proof. Let r_{in} be a vertex with only in-edge(s).

Then r_{in} cannot enable any other rule. Thus the only scenario in which the non-firing of r_{in} might break the SSG cycle is: there is a rule r, there is a variable v_1, $v_1 \in L_{r_{in}}$, $v_1 \in L_r$, and r_{in} and r assign different expressions to v_1. This is so because a rule will fire again if at least one variable in LHS of that rule has changed its value since the last firing of that rule. We shall show that in this scenario, r_{in} must also have at least one out-edge.

Since r_{in} and r do not satisfy condition CR2 or CR3, they must be mutually exclusive (condition CR1 is satisfied). From lemma 1, r_{in} must be disabled in some state in the SSG cycle. Thus it must be re-enabled by another rule before it can fire again. There must be at least one variable v_2 which appears in the enabling conditions of these rules and the value of v_2 changes infinitely often among a finite number of values, enabling these CR1-compatible rules alternatively. Note that now the variable v_2 is in the same situation as the variable v_1 is in above. We repeat this argument to include more variables $v_1, v_2, v_3, v_4, \cdots$ Since there is a finite number of variables, we will eventually reuse a variable v_k which is the same variable v_i, $1 \le i \le k-1$, that has been included already. Thus a 'path' or braid is traced from v_1 to v_k.

Suppose $v_k = v_1$. Since r_{in} does not have out-edges, it cannot enable any other rule and thus cannot, together with r, alternatively enable the rules whose tests contain variable v_1. Rules other than r_{in} must modify v_1 or another variable in the braid in order to change the value of v_1 (and the other variables in the braid) infinitely. Thus not firing r_{in} does not break the SSG cycle. Suppose $v_k \ne v_1$, and v_k is modified by r_{in} or by r. Again, since r_{in} does not have out-edges, it cannot enable any other rule and thus cannot, together with r, alternatively enable the rules whose tests contain variable v_k. Rules other than r_{in} must modify v_k or another variable in the braid in order to change the value of v_1 (and the other variables in the braid) infinitely. Thus not firing r_{in} does not break the SSG cycle. If $v_k \ne v_1$ and v_k is not modified by r_{in} or by r, then it is obvious that a SSG cycle still exists even if r_{in} is not fired.

The above reasoning implies that the SSG cycle still exists if r_{in} is not fired. We bypass the edge corresponding to r_{in} in the SSG cycle as follows. Let (s, s', s'') be a subpath of the SSG cycle. Let edge (s, s') corresponds to r_{in} and let edge (s', s'') corresponds to r_a. r_{in} needs not fire and thus state s' can be bypassed by firing r_a at state s, putting the program in state s'''. State s''' is the same as state s'' except possibly for those variables whose contents are changed by r_{in} if fired at state s. However, since the changing of the contents of these variables by r_{in} will not enable any rule and because of the argument above, state s' can be bypassed. We remove all vertices with in-edge(s) only from G' and bypass the corresponding rule edges in the SSG cycle in the manner described above. We now have a ERG subgraph G'' defined as follows:

$$V'' = V' - \{ r_{in} \mid r_{in} \text{ has in-edges only} \}.$$
$$E'' = E' - \{ (r_i, r_j) \mid r_j \text{ has in-edges only} \}.$$

Every vertex in G'' has at least one incident in-edge and at least one incident out-edge. Thus there is at least one strongly connected component in G'' as well as in G'. Let $C_1, C_2, ..., C_m$ be the strongly connected components of $G''(V'', E'')$. There are three cases to consider.

(1) There is only one strongly connected component. In this case, there must be a cycle (possibly non-simple) which contains all vertices in G''. Since the expressions in R consist of constants only and the number of rules in the program is finite, none of the variables in the program would change in value infinitely among an infinite number of values. Consequently, a set of rules will fire forever (creating a SSG cycle) and will not reach a fixed point *only if* there is at least one variable v_1 in L whose value is changed infinitely often among a finite number of values. Then there exist at least two rules in the SSG cycle which assign different expressions to v_1. Since each rule in the SSG cycle (after

bypassing those states as described above) is also represented by a vertex in ERG subgraph G'', there must be at least two rules which assign conflicting expressions in G''. Since these conflicting rules are in the cycle described above, it can be concluded that there is a bad ERG cycle. Next, we show that there must be a simple cycle with at least two rules which assign different expressions to the same variable, or rules in disjoint simple cycles assign different expressions to the same variable. There are two basic cases to consider.

(1.1) 8-circle: two or more simple cycles connected by a single common vertex r. If r and a rule r_1 in one of these simple cycles assign different expressions to the same variable, then there is simple cycle which contains two conflicting rules. Now suppose that for every rule r_i ($r_i \neq r$) in these simple cycles, r and r_i do not assign different expressions to the same variable. If the content of a variable v_2 in LHS of r is changed after firing r, then there is another rule r_2 not in this 8-circle which assigns a conflicting expression to v_2 and the argument for case (1.2) applies. Rule r_2 cannot be in another SCC since there is only one SCC. If the contents of the variables in LHS of r do not change after firing r, then r can fire at most once. Then successors of r can be re-enabled at most once. If the successors are mutually exclusive, then only one can be enabled at the same time. This means that all simple cycles with vertex r in common are broken. These cycles effectively become acyclic paths without repeated firings of rule r. Even if there are two conflicting rules, r_a and r_b, r_a is in simple cycle S_1, r_b is in simple cycle S_2, and both cycles are in the 8-circle, no infinite rule firings can occur. Thus there must be two conflicting rules in one simple cycle or in disjoint simple cycles.

(1.2) Overlapping simple cycles: two or more simple cycles with at least one edge (and at least two vertices) in common. This is in fact a generalization of the 8-circle where there are two or more simple cycles with one vertex (and no edges) in common. Let r be a vertex common in these simple cycles. If r and a rule r_1 in one of these simple cycles assign different expressions to the same variable, then there is simple cycle which contains two conflicting rules. Now suppose that for every rule r_i ($r_i \neq r$) in these simple cycles, r and r_i do not assign different expressions to the same variable. If the content of a variable v_2 in LHS of r is changed after firing r, then there is another rule r_2 not in these overlapping cycles which assigns a conflicting expression to v_2. (Rule r_2 cannot be in another SCC since there is only one SCC.) Then, one of the following scenarios may occur: (a) r_2 is in another simple cycle disjoint from the overlapping simple cycles we are considering, or (b) r_2 is in one of the simple cycles in a 8-circle, and the common vertex in this 8-circle is a vertex that is common in the simple cycles of the overlapping cycles we are considering (otherwise, there are disjoint simple cycles). In scenario (a), condition B5 is violated. In scenario (b), r and r_2 cannot fire forever because of argument 1.1. Thus rules in the path common in the simple cycles in this overlapping-cycles figure cannot fire forever. Then successors of r cannot be re-enabled infinitely often. If the successors are mutually exclusive, then only one can be enabled at the same time. This means that all simple cycles with vertex r in common are broken. Even if there are two conflicting rules, r_a and r_b, r_a is in simple cycle S_1, r_b is in simple cycle S_2, and both cycles are in this overlapping-cycles figure, no infinite rule firings can occur. Thus there must be two conflicting rules in one simple cycle or in disjoint simple cycles.

(2) G'' is connected and there are more than one strongly connected component (SCC) in G''. Consider two of the SCCs C_a and C_b, each of which has at least two vertices, connected by at least one one-way directed path consisting of one or more edges. This implies that the vertices and edges in C_a and those in C_b are disjoint. If the path has zero edges, then a vertex must be common in both C_a and C_b, and thus C_a and C_b must be in the same SCC. Let $r_1 \in V(C_a)$ and let $r_2 \in V(C_b)$. Suppose there is a directed path from r_1 to r_2, and no SCC (besides C_a and C_b) with more than one vertex lies in the path from r_1 to r_2. There is no path from a vertex in C_b to a vertex in C_a; otherwise,

C_a and C_b would be in the same SCC. From the argument in case 1, we know that there are at least two rules in G'' which assign conflicting expressions to the same variable. Let r_a and r_b be these conflicting rules. Suppose $r_a \in V(C_a)$ and $r_b \in V(C_b)$. Since C_a and C_b each contain at least two vertices, both contain simple cycles. One simple cycle in C_a must contain r_a, and similarly, one simple cycle in C_b must contain r_b. These two simple cycles are disjoint because C_a and C_b are disjoint. Thus condition B5 is violated.

If both r_a and r_b are in C_a or both are in C_b, then condition B4 is violated (see case (1)). Therefore, we only need to show that the following scenario cannot occur: r_a is in a one-vertex SCC in the path from C_a to C_b, r_b is in either C_a or in C_b, and there are no conflicting rules in either C_a or in C_b. As shown earlier, r_a and r_b must be mutually exclusive.

(Argument 2.1) Suppose r_b is in C_b. Suppose r_a fires first. Then following a sequence of zero or more rule firings, r_b becomes enabled and fires. At this point, r_a must be disabled because of the mutual exclusion condition. Suppose there are no conflicting rules in C_a; otherwise, the argument for case (1) applies. Also suppose there does not exist a rule r_c in another SCC C_c with at least two vertices, the LHS of r_c contains a variable v_1, and this variable also appears in LHS of a rule r_d in C_a such that the expressions assigned to v_1 are different. (If rule r_c exists, then the argument given earlier for the violation of condition B5 applies.) Then each rule in C_a fires at most once. We shall treat the subcase (*) in which rule r_c appears in a one-vertex SCC shortly. For now, assume that no such rule r_c exists. Then rules in C_a can fire at most once and will reach a fixed point. Thus r_a can be re-enabled at most once. After firing r_b a second time, r_a cannot fire again since r_a must become disabled before the second firing of r_b. Therefore, no infinite rule firings is possible in this scenario. Suppose r_b fires first. Then following a sequence of zero or more rule firings, r_a becomes enabled and fires. At this point, r_b must be disabled because of the mutual exclusion condition. The above argument for the case in which r_a fires first applies.

Suppose r_b is in C_a instead of in C_b. A similar argument as the one above applies here.

Now we consider subcase (*) where rule r_c appears in a one-vertex SCC C_c. (Argument 2.2) Since G'' is connected, there must be either a one-way directed path from C_c to C_a or a one-way directed path from C_a to C_c. If both paths exist at the same time, then C_c and C_a are in the same SCC with two conflicting rules and the argument for case (1) applies. If the path is from C_a to C_c, then the firing of r_c cannot enable rules in C_a. We consider the case where the path is from C_c to C_a. Now we treat r_c as r_a, treat r_d as r_b, and then apply argument 2.1. If there is a rule in the role of r_c within argument 2.1, then we apply argument 2.2 again. If we 'trace' a reverse path back to C_b (this path consists of disjoint SCCs connected to one another by one-way directed paths), then there is one big cycle containing at least two rules which assign different expressions to the same variable and thus condition B4 is violated. If no such paths can be traced, applying argument 2.2 repeatedly will eventually lead to a situation where rule r_c does not exist in argument 2.1. Then each rule in C_a fires at most once in argument 2.1 and the rest of the argument follows.

(3) G'' is disconnected and thus there are more than one strongly connected component in G''. Since vertices with no incident edges cannot exist in G' (shown earlier), every SCC which is not connected to another SCC has at least two vertices. The case in which there is a one-vertex SCC but connected to another SCC is treated in case (2) above. From the argument in case (1), if there is a SSG cycle, then there must be at least two rules which assign conflicting expressions to the same variable. If condition B4 is satisfied, then there are no conflicting rules in any SCC. Therefore, rules in different SCCs and thus rules in disjoint simple cycles assign conflicting expressions to the same LHS variable, thereby violating condition B5. Suppose there are two

disconnected components (not necessarily SCCs) C_1 and C_2, there are two rules r_a and r_b which assign different expressions to the same variable, r_a is in a one-vertex SCC in C_1, r_b is in a SCC with at least two vertices in C_2. Since r_a and r_b must be mutually exclusive, when r_a is enabled to fire, r_b must be disabled, and vice versa. In order for these two rules to fire alternatively, there must be two other rules r_c and r_d, r_c enables r_a and is in C_1, r_d enables r_b and is in C_2. r_c and r_d are mutually exclusive since they also assign different expressions to a common variable in order to enable r_a and r_b alternatively. If r_c is in a SCC with at least two vertices and if r_d is in a SCC with at least two vertices, then condition B5 is violated. If not, we can trace those rules which alternatively enable r_c and r_d. Rules in a disconnected acyclic path cannot fire forever. In a finite number of steps, we will find two rules, one in a SCC with at least two vertices in C_1 and one in a SCC with at least two vertices in C_2, which assign conflicting expressions to a common variable. This is due to the fact that G'' is finite and not all SCCs in a disconnected component contain a single vertex. Otherwise, there is a vertex with no in-edge or a vertex with no out-edge. However, this contradicts the fact that every vertex in G'' has in-edge(s) and out-edge(s). Therefore, condition B5 is violated. The case in which r_a is in a one-vertex SCC in C_1 and r_b is in a one-vertex SCC in C_2 can be treated similarly. If condition B5 is satisfied, then there must be at least two rules which assign conflicting expressions in one of the SCCs in G''. Since these conflicting rules are in the cycle described in case (1), it can be concluded that there is a bad ERG cycle, thereby violating condition B4.

$\square$

5.3.1. A Greedy Recognition Procedure for Special Form B

The ER graph G corresponding to the set of rules is constructed. Given a set S of rules and a set B initialized to the empty set, a simple implementation of the recognition procedure incrementally adds a rule i to B such that the set

$B \cup \{i\}$ is in special form **B**. The algorithms for checking the satisfiability of the conditions B1 and B2 are exactly the same as those for checking the satisfiability of the conditions A1 and A2. For checking the satisfiability of the condition B3, the algorithm is the negation of the output from the algorithm for checking the satisfiability of the condition A3.

To check the satisfiability of condition B4, the greedy algorithm performs a depth-first search, starting from the vertex corresponding to the rule i being checked for inclusion in set B, on the **ER** subgraph G' consisting of the vertices corresponding to those rules already in B and the rule i, and the edges connecting these vertices. If a cycle with the vertex i in it is found (the search visits the initial vertex i corresponding to the rule i again), the algorithm invokes a procedure to check whether the cycle is good. If the cycle is good, then the algorithm proceeds to check for other cycles involving vertex i until a bad cycle is found or all cycles in G' have been checked. If a bad cycle involving the vertex i is found, rule i is not included in B.

In order for a rule i to be added to set B, all five conditions B1-B5 must be satisfied. The recognition algorithm terminates when all rules in S are checked. The output from the algorithm is the set B.

5.3.1.1. Complexity of the Greedy Recognition Procedure

As in the case of special form A, it is easy to see that the complexities of the recognition algorithms for checking a set of rules for the satisfiability of the conditions B1, B2 (CR2 and CR3), and B3 are respectively $O(n)$, $O(n^2)$, and $O(k^2)$, where n is the number of rules in the set, and k is the number of variables in the set. Note that in the worst case, the checking of condition B2:CR1 (mutual exclusion) may require exponential time. However, as noted earlier, programmers do not write unstructured tests in practice and thus the checking of condition CR1 (which is needed only when the other two compatibility criteria are not satisfied) is usually not a problem. The complexity for checking conditions B4 and B5, however, is exponential in the worst-case. It is well known that an algorithm for determining all directed circuits (cycles) in a directed graph has an exponential run-time complexity ([Garey & Johnson 79]), owing to the combinatorial number of distinct directed circuits that may exist in the directed graph in the worst case. A simple extension to this recognition algorithm is actually a solution to the *directed Hamiltonian circuit problem*, which is known to be NP-complete.

In practice, the number of cycles in the **ER** graph corresponding to an **EQL** program that have to be checked is small. The algorithm used for identifying the set of rules in special form **B** does not need to analyze all distinct cycles in the graph unless none of cycles are bad. Thus it is still practical to use such a recognition algorithm for checking realistic **EQL** programs. In the following section, we present an efficient, polynomially-bounded algorithm for checking condition B4.

5.3.2. An Efficient Algorithm for Checking Condition B4

The greedy algorithm given in section 5.3.1 may require exponential computation time to determine the satisfiability of the condition B4 when given a worst-case **ER** graph. Although it is observed that the number of distinct cycles in the **ER** graphs corresponding to actual programs is small, the number of such cycles in the **ER** graphs corresponding to large programs is still large. Thus it is necessary to develop a more efficient algorithm that has polynomial time complexity. This new algorithm does not need to check all cycles and is not based on the *fundamental cycles* of the **ER** graph with respect to one of its spanning trees. (The addition of a chord to a spanning tree of a graph creates precisely one cycle. The collection of these cycles with respect to a particular spanning tree is called a set of fundamental cycles and these cycles form a basis for the cycle space.) The algorithms for checking the satisfiability of conditions B1-B3 remain the same.

The algorithm outlined below operates as follows. First, it determines all distinct pairs of rules that have conflicting assignments to at least one variable common in both rules. Then it checks each pair of these rules to determine whether they are both in a cycle. Let p and q be a pair of rules with conflicting assignments to the same variable. To determine whether they are in the same cycle, the algorithm attempts to find two vertex-disjoint (except the initial and final vertices) and edge-disjoint paths, one from p to q, and the other from q to p. This is equivalent to finding whether the two vertices are biconnected. If a pair of rules are in the same cycle is detected, then a cyclic sequence of rule firings may occur.

Efficient Algorithm for Checking Condition B4:

```
$C := \varnothing$;
for $j := 1$ to $n$ do
    for $k := 2$ to $n$ do
        if $j \neq k$ and conflicting_assign($j, k$)
            then $C := C \cup \{ \{j, k\} \}$
for each rule pair $\{p, q\}$ in $C$ do
    if vertex $p$ and vertex $q$ are biconnected
        then there is a bad cycle
```

Given a set S of rules, in order to incrementally add a rule i to the set B (which contains rules that are in special form **B**) such that the set $B \cup \{i\}$ is also in special form **B**, the recognition procedure checks the new rule i against every rule already in B. Then, if there exist conflicting assignments to the same variable(s) in rule i and in rule $j \in B$, the procedure checks whether these two rules lie on the same cycle. If they are not in the same cycle, the procedure adds rule i to B. Otherwise, rule i is not included in B.

5.3.2.1. Complexity of the Efficient Recognition Procedure

The efficient algorithm described in the preceding section has time complexity $O(MAX(n^2, m \times MAX(n,e)))$, where m is the number of rule pairs with conflicting assignments to a common variable, n and e are respectively the number of vertices (rules) and the number of edges in the **ER** graph corresponding to the set of rules. Note that $m \leq n(n-1)/2$.

Given a program with n rules, all pairs of rules with conflicting assignments to at least one common variable can be found in $O(n^2)$-time since there are at most $n(n-1)/2$ distinct pairs of rules. For each pairs of rules with conflicting assignments, the algorithm checks whether the vertices corresponding to these two rules are biconnected (both lie on the same cycle). This can be achieved in $O(MAX(n,e))$-time using the depth-first search strongly connected components algorithm of Tarjan ([Tarjan 72]). Since there are m pairs of such rules, the second loop has time complexity $O(m \times MAX(n,e))$. Thus the algorithm has overall time complexity $O(MAX(n^2, m \times MAX(n,e)))$, and in the worst case where every pair of rules have conflicting assignments, $O(n^2 \times MAX(n,e))$-time.

5.3.3. Application of Special Form B

To illustrate the application of special form **B**, consider the programs in the two examples below. Note that all rules in the programs of example 5.3.3.1 and example 5.3.3.2 satisfy conditions B1, B2 and B3. The **ER** graph corresponding to the program of example 5.3.3.1 is shown in Figure 5.4. Observe that there is a bad cycle in this graph -- there are three pairs of rules in it that assign different values to the variables a, b and c, respectively. Therefore, condition B4 is violated and thus this program is not in special form **B**.

Example 5.3.3.1. (This program is not in special form B.)

```
(* 1 *)    b := 0  IF  a = 1
(* 2 *) [] b := 1  IF  a = 0
(* 3 *) [] c := 0  IF  b = 0
(* 4 *) [] c := 1  IF  b = 1
(* 5 *) [] a := 0  IF  c = 0
(* 6 *) [] a := 1  IF  c = 1
```

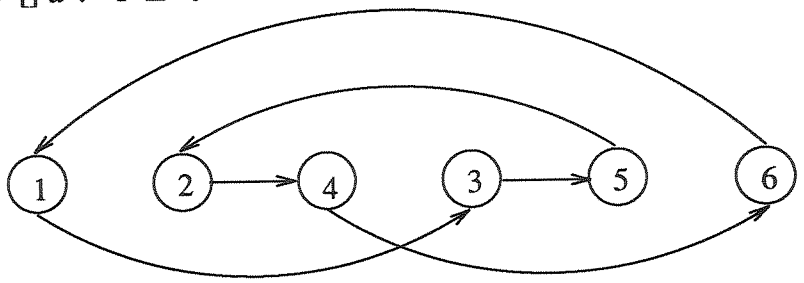


Figure 5.4. ER Graph Corresponding to the Program of Example 5.3.3.1.

Now consider the program of example 5.3.3.2, whose **ER** graph is shown in Figure 5.5. Observe that there is one cycle in this graph but rules in this cycle do not assign different expressions to the same variable. Conditions B4 and B5 are satisfied. Therefore, this program is in special form **B** and thus it is always guaranteed to reach a fixed point in bounded time.

Example 5.3.3.2. (This program is in special form B.)

```
(* 1 *)    b := 0  IF  a = 1
(* 2 *) [] b := 1  IF  a = 0
(* 3 *) [] c := 0  IF  b = 0
(* 4 *) [] c := 1  IF  b = 1
(* 5 *) [] a := 1  IF  c = 0
(* 6 *) [] a := 2  IF  c = 1
```

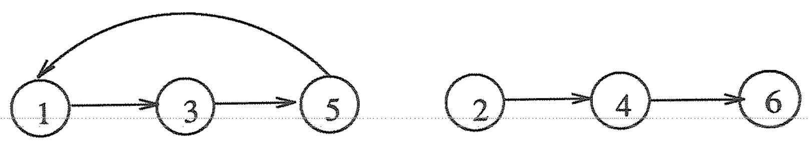


Figure 5.5. ER Graph Corresponding to the Program of Example 5.3.3.2.

5.3.4. Deriving Tight Numeric Response Time Bound for Special Form B

The response time formula for special form **A** cannot be used to compute the response time bound for programs in special form **B** since the formula is based on the assumption that rules which are enabled (disabled) before the firing of any rule remain enabled (disabled) during the course of rule firings, but this is not the case for rules in special form **B**. To compute the response time bound for special form **B** rules, we present an algorithm that performs the analysis on the data dependency graph called the **access-modify (AM)** graph ([Wang, Mok & Cheng 90]) and the **ER** graph corresponding to these rules. This response time analysis algorithm is an extension of the one proposed in [Wang, Mok and Cheng 90] and it computes a more precise bound.

Definition 5.3.4.1.

An **access-modify graph** (**AM** graph) is a simplified representation of a set of rules. It consists of two kinds of vertices. Each variable vertex represents one variable and each rule vertex represents one rule. A directed edge from a rule vertex to a variable vertex means that this variable appears in the LHS of the rule. A directed edge from a variable vertex to a rule vertex means that this variable appears in the enabling condition of the rule.

Before we present the algorithm, observe that there is a close relationship between the **AM** graph and the **ER** graph. If there exist an edge (i,j) in the **ER** graph, then there exist a variable x such that $x \in L_i \wedge x \in T_j$, i.e., there exist an edge (i,x) and an edge (x,j) in the **AM** graph.

Algorithm for Computing Response Time Bound for Special Form B

We define two values, NC and NF, used by the following algorithm. Let i be a variable, and NC_i an upper bound on the number of changes that may happen to the content of variable i before the program reaches a fixed point. Let C be a set of rules in special form **B**, and NF_C an upper bound on the number of firings of rules in the set C.

The following response time analysis algorithm consists of two subprocesses. The first subprocess determines the NC value for each variable. The basic idea of this subprocess is to set up NC values of some special variable vertices initially, then *diffuse* NC values to other variable vertices based on the **AM** graph. The second process uses these NC values to determine, again based on the **AM** graph, the NF value of each set of rules in special form **B**.

(1) If the set of rules is in special form **B**, then proceed. Otherwise terminate this algorithm with the upper bound on the number of firings set to infinity.

(2) Subprocess 1:

(a) Initially assign values to the NCs of some variable vertices according to the following rules:

(i) $\forall$ variable i: variable i is not modified by any rule, then $NC_i := 0$.

(ii) $\forall$ variable i: variable i is modified by only one rule, *or* variable i is modified by more than one rule and all the assignments are the same, then $NC_i := 1$.

(b) Then compute NCs of other variable vertices according to the following function:

$$NC_i = 1 + \sum_j NC_j$$

where (i) variable j is accessed by at least two rules that disagree on variable i, and (ii) it is not the case that there exist a **disable** in-edge incident to each of the vertices representing these conflicting rules.

(c) If the value of NC_i cannot be computed because at least one NC_j is undefined, then

$$NC_i = 1 + \sum_k NC_k + C_i$$

where each NC_k is defined as above but with a known value (computed in step (a)), and C_i is the number of simple **ER** cycles which, when superimposed on the **AM** graph, contain variable vertex i.

We call this process the *diffusion* of NC values because NC_is can be computed only after all the NC_js have been computed.

(3) Subprocess 2:

The NF value of a set C of special form **B** rules is computed by the following function:

$$NF_C = \sum_{i : i \in L_C} NC_i$$

with the following exception: $\forall$ variables $x_1, \ldots, x_k$, $k > 1$, which appear in the LHS of every rule in a subset S of rules in C, but do not appear in the LHS of any rule not in S, only $\max(NC_{x_1}, ..., NC_{x_k})$ should be included in the sum of the above formula.

In the subprocess 2, we compute NF values with regard to a set of rules instead of individual rules for two reasons. First of all, sometimes it is impossible to determine the exact NF value for each rule at compile time. The exact NF values depends on the contents of variables. Second, it is usually good enough to know the NF values of sets of rules instead of the NF value for each rule because the basic units selected at each step of the general analysis algorithm are independent sets of rules, and these sets of rules are treated as unit modules when scheduling the rules to fire.

Proof of the Correctness of the Algorithm:

The correctness of the basic algorithm is proved in [Wang, Mok & Cheng 90]. We shall prove the correctness of the new algorithmic steps not found in the basic algorithm.

Step 2(b)(ii): If variable j is accessed by at least two rules that disagree on variable i, but there is a disable in-edge incident to each of the vertices corresponding to these rules, then these conflicting rules would not fire more than once. This single firing may still modify the content of variable i, but this modification is counted by variable C_i as explained in step 2(c) below. Thus the value of NC_j should not be included when computing the value of NC_i.

Step 2(c): If the value of NC_i cannot be computed because at least one NC_j is undefined, then there must be a good **ER** cycle. When this **ER** cycle is superimposed on the **AM** graph, one can trace an **AM** cycle which contains variable vertex i. This cycle cannot be bad since a program with bad **ER** cycle(s) is not in special form **B** and thus step 1 would have terminated the algorithm by setting the upper bound on the number of firings to infinity. A disagreeable braid ([Wang, Mok & Cheng 90]) exists but this braid must be harmless since the special form **B** recognition algorithm is able to distinguish a good **ER** cycle from a bad one, both of which may indicate the existence of a braid. Variable C_i counts the number of changes to the content of variable i by the rules appearing in each simple and good **ER** cycles each of which contains variable i.

Step 3(exception part): If the condition appearing in this exception holds, then whenever a rule modifies the content of one of the variables xs, it modifies the contents of all variables xs. Therefore, only one rule firing (instead of k) should be counted.

□

5.4. More General Conditions for Fixed Point Reachability

This section extends the notion of the enable-rule relationship developed in the previous section and provides a set of more general conditions for guaranteeing bounded-time fixed-point reachability. Specifically, we state a theorem which gives more general sufficient conditions, relative to those given by the special form **B**, for bounded-time fixed-point reachability. This theorem thus generalizes the potentially-enable relation used in special form **B**. We first introduce some definitions that will be used in the remainder of this section. We use the terms *rule* and *vertex* (used to represent a rule in the **ER** graph) when there is no ambiguity.

As we shall describe in chapter 7, the rule-based programmer can supply application-specific knowledge about a rule-based program in the form of enable-rule and disable-rule tables. These tables specify the enable sets and disable sets for each rule in the program. Since the contents of some variables cannot be determined before runtime, the analysis algorithm may not be able to detect all enable-rule and disable-rule relationships among rules. With the programmer-supplied information and the general theorem described below about these enable-rule and disable-rule relationships, the analysis algorithm can recognize a larger class of rule-based programs with bounded response time.

Definition 5.4.1.

The **precedent set** of a rule vertex v is defined as the set of rule vertices v_j that are connected to v by edge (v_j, v).

Definition 5.4.2.

An **enable set** of a rule r is defined as a set of rules whose firing in some order will enable rule r if rule r is not already enabled. A **non-enable set** of a rule r is a set of rules whose firing in any order will not enable rule r if rule r is not already enabled. Note that a rule may have more than one distinct enable set.

Definition 5.4.3.

An enable set M of a rule r is said to be **minimal** iff the removal of any rule from M would make M a non-enable set. Note that a rule may have more than one distinct minimal enable set.

Definition 5.4.4.

A **disable** edge (r,s) connects vertex r to vertex s iff the firing of rule r always disables the enabling condition of rule s.

Definition 5.4.5.

The **disable** graph of a set of rules is a labeled directed graph

$G = (V, E)$. V is a set of vertices each of which is labeled by a rule number i corresponding to a distinct rule i. E is a set of disable edges.

Consider the rules in the following example.

Example 5.4.1.

```
(* 1. *)    $a := 1 ! n := 0$  IF $m = 1$
(* 2. *) [] $b := 0 ! p := 0$  IF $n = 1$
(* 3. *) [] $c := 2 ! q := 1$  IF $p = 1$
(* 4. *) [] $d := -2 ! r := 1$  IF $q = 0$
(* 5. *) [] $e := 1 ! s := 1$  IF $r = 0$
(* 6. *) [] $f := 1 ! q := 1$  IF $s = 0$
(* 7. *) [] $g := 0 ! n := 0$  IF ($a = 1$ AND $b = 0$ AND $c > 0$ AND $d < 0$)
                                  OR ($d < 0$ AND $e = 1$ AND $f > 0$)
```

Rules 1-5 constitute an enable set for rule 7, so is the set of rules 3-6. The minimal enable sets for rule 7 are $S_{7,1}$ (rules 1-4) and $S_{7,2}$ (rules 4-6). The **ER/disable** graph corresponding to this program appears in Figure 5.6. In the graph, solid arrows denote **ER** edges and dotted arrows denote disable edges.

Theorem 5.4.1.

For a set W of rules satisfying the conditions B1, B2, B3 and B5 of special form **B**, there exist at least one infinite firing sequence (in which the set of rules does not reach a fixed point in bounded time) only if all of the following conditions hold:

(1) There exist a strongly connected component C with at least 2 vertices in the **ER** graph corresponding to W.

(2) There exist at least one cycle of length m $(m \geq 2)$ in C in which each rule r_i, $i = 1,..., m$ (represented by vertex v_i) is enabled by the precedent set P (that is, there exist an enable set N for rule r_i such that $N \subseteq P$);

(3)(a) There exist at least one minimal enable set $S_{i,j}$ for each rule vertex r_i in the above cycle whose disable subgraph $G_D(V_D, E_D)$ is acyclic, where V_D is a minimal enable precedent set of rule vertex r_i and E_D is the set of **disable** edges connecting vertices in V_D, for $i = 1, ..., m$; *or*

(3)(b) there exist at least one vertex r_k in a disable-cycle such that r_k has a minimal enable set which satisfies condition (3)(a) or condition (3)(b).

(4) There are at least two rules which assign conflicting values to the same variable.

The intuition behind the derivation of these general conditions can be illustrated by the following examples. A proof of a variant of this theorem targeted for showing the bounded-time fixed-point reachability of rule-based **EQL** programs will then follow.

Condition (1): Since the number of rules in a program is finite, at least one rule must be fired infinitely often for the program not to reach a fixed point in bounded time. However, since the expressions in R consist of constants only, a rule a does not fire again once it has already fired once unless at least one variable in L_a has changed its value as a result of the firing of another rule. Thus at least two rules must be involved in a cyclic firing sequence. Since a cycle with at least two vertices is a connected component, a strongly connected component must exist. The reason for using the term *strongly connected component* to make the condition statement more general will become clear later.

Condition (2): The key observation here is that in order for a rule to fire more than once, it either must stay enabled or if disabled, must be enabled by a set of rules constituting an enable set. A more precise argument will appear in the proof steps of theorem 5.4.1' below.

Condition (3): Consider the partial **ER/disable** graph in Figure 5.6. Let solid arrows denote potentially enable edges and let dotted arrows denote disable edges. The rule vertices in each rectangle represent a minimal enable set for rule 7. This **ER/disable** graph corresponds to the program segment of example 5.4.1.

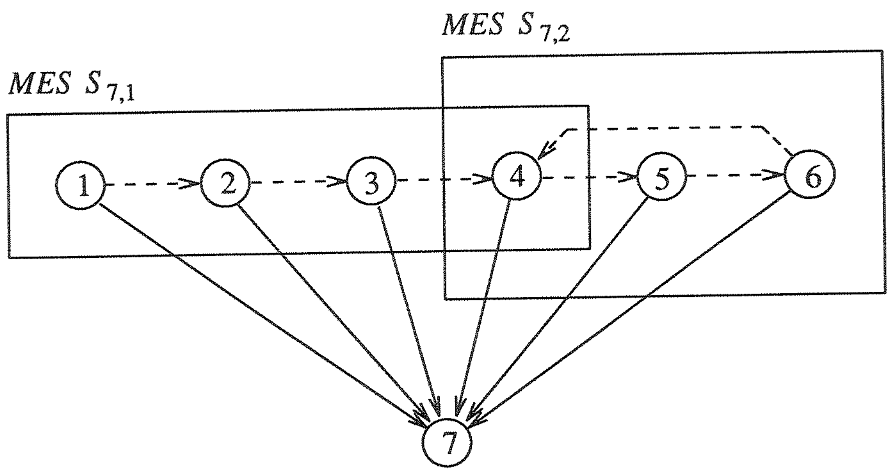


Figure 5.6. Situation 1 for Condition 3 of Theorem 5.4.1.

The disable subgraph of the minimal enable set $S_{7,2}$ contains a cycle. However, the disable subgraph of the minimal enable set $S_{7,1}$ is acyclic and thus the rules in it can be fired in the following order: 4, 3, 2, 1. In fact, for any acyclic graph, there is a corresponding topological ordering of its vertices. Thus all rules in a minimal enable set for a rule r whose **disable** graph is acyclic can be fired in some order, thus enabling rule r. Now consider the partial **ER/disable** graph in Figure 5.7.

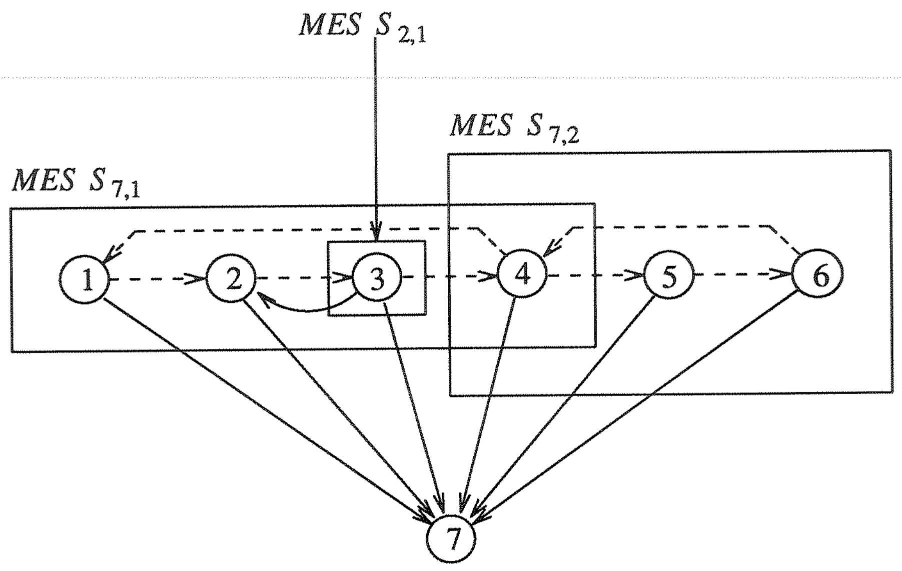


Figure 5.7. Situation 2 for Condition 3 of Theorem 5.4.1.

The disable subgraphs of both enable sets of rule 7 contain cycles. However, rule 2 in the disable cycle in the subgraph of the minimal enable set $S_{7,1}$ has a minimal enable set consisting of rule 3 whose disable subgraph is obviously acyclic. Thus the rules in $S_{7,1}$ can be fired in the following order: 1, 4, 3, 2. The existence of the minimal enable set $S_{2,1}$ effectively breaks the cycle in the disable subgraph corresponding to the minimal enable set $S_{7,1}$, thus allowing rule 7 to be enabled. Next we consider the partial **ER/disable** graph in Figure 5.8.

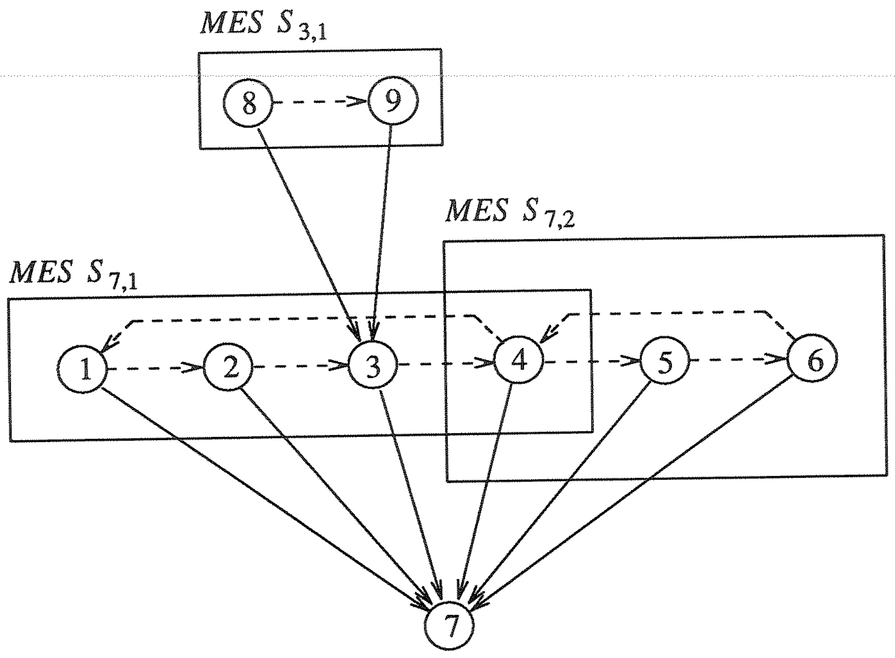


Figure 5.8. Situation 3 for Condition 3 of Theorem 5.4.1.

The disable subgraphs of both enable sets of rule 7 contain cycles. However, rule 3 in the disable cycle in the subgraph of the minimal enable set $S_{7,1}$ has a minimal enable set $S_{3,1}$ consisting of rule 8 and rule 9 whose disable subgraph is acyclic. Thus the rules in $S_{7,1}$ can be fired in the following order: 2, 1, 4, 3, and the rules 8 and 9 are fired in the order 9, 8 anytime after the firing of rule 2 and before the firing of rule 3. Since all rules in the enable set $S_{7,1}$ can be fired, rule 7 can be enabled.

Condition (4): Since the expressions in R consist of constants only and the number of rules in the program is finite, none of the variables in the program would change in value infinitely among an infinite number of values and thus a cycle in the enable-rule graph of the program corresponds to a cycle in the state space graph of the program. Consequently, a set of rules will not reach a fixed point *only if* there is at least one variable v_1 in L whose value is changed infinitely often among a finite number of values. Then there exist at least two rules which assign different values to the variable v_1.

Theorem 5.4.1 can be rephrased as follows to reflect our goal which is to show that a set of rules satisfying a set of conditions is guaranteed to reach a fixed point in bounded time. The proof of theorem 5.4.1' thus can be easily modified as a proof for theorem 5.4.1 above.

Theorem 5.4.1'.

For a set S of rules satisfying the conditions B1, B2, B3 and B5 of special form **B**, a fixed point is reachable in bounded time (there does not exist an infinite firing sequence) if at least one of the four conditions of theorem 5.4.1 is false.

Proof of theorem 5.4.1'.

(1) Proof of [B1, B2, B3, B5 and condition (1) is false $\rightarrow$ a fixed point is always reachable in bounded time].

Assume that the first condition is false; that is, there does not exist a strongly connected component with at least two vertices in the **ER** graph. Assume that a fixed point is not always reachable in bounded time. Then by the definitions given in chapter 4, in the state space graph corresponding to the program, there must be (1) at least one cycle with at least two vertices, or (2) at least one simple path of unbounded length. Since a rule does not fire again after firing once unless the subsequent firing changes the value of at least one variable on its left-side and the expressions in R consist of constants only, cycles with one vertex (self-loops) pose no problems for bounded-time termination. Since the expressions in R consist of constants only and the number of rules in the program is finite, none of the variables in the program would change in value infinitely among an infinite number of values. A cycle corresponding to situation (1), but not situation (2), in the state space graph of the program must exist in the **ER** graph of the program. A cycle is a connected component of the graph; it may be itself a strongly connected component or it may be part of a strongly connected component. Since the cycle has at least two vertices, there exist a strongly connected component with at least 2 vertices in the **ER** graph, a fact that violates the first assumption. Therefore, if condition (1) is false, the set of rules is guaranteed to reach a fixed point in bounded time.

(2) Proof of [B1, B2, B3, B5 and condition (2) is false $\rightarrow$ a fixed point is always reachable in bounded time].

Assume that the second condition is false; that is, there does not exist one cycle of length m $(m \geq 2)$ in C in which each rule r_i, $i = 1,..., m$ (represented by vertex v_i) is enabled by the set P of preceding adjacent rules (represented by vertices preceding and adjacent to v_i in the **ER** graph). That is, in every cycle in C, there exist one rule a such that the set P of rule vertex a does not constitute an enable set N for rule a $(N \not\subseteq P)$. Assume that a fixed point is not always reachable in bounded time.

Assume that rule a stays enabled and hence does not need an enable set. Rule a will fire again after firing once if and only if at least one variable in L_a has changed its value as a result of the firing of another rule since the last firing of rule a. Thus at least two rules must be involved in a cyclic firing sequence. For rule a to stay enabled, it must be compatible with every other rule inside and outside the strongly connected component C by condition CR2 or by condition CR3. Since rule a stays enabled, it may fire anytime provided the firing will change the value of at least one variable in L_a. From previous reasoning, this occurs if and only if at least one variable in L_a has been modified by another rule b. Rule b cannot be compatible with rule a by neither condition CR2 or condition CR3 and hence they must be compatible by condition CR1. This leads to a contradiction since rule a cannot stay enabled, at least not enabled when rule b is enabled. Furthermore, rule a must be re-enabled by an enable set of rules, a fact that contradicts our assumption.

(3) Proof of [B1, B2, B3, B5 and and condition (3) is false $\rightarrow$ a fixed point is always reachable in bounded time].

Assume that the third condition is false; that is, the disable subgraphs $G_D(V_D, E_D)$ of all minimal enable sets $S_{i,j}$ of rule r_i, where V_D is a minimal enable set of parallel vertices preceding and adjacent to rule vertex v_i and E_D is the set of **disable** edges connecting vertices in V_D, are not acyclic, *and* there does not exist one vertex r_k in a disable-cycle having at least one minimal enable set satisfying condition (3)(a) or condition (3)(b). Assume that a fixed point is not always reachable in bounded time.

From above, at least two rules must be involved in a cyclic firing sequence. Since the subgraph of each minimal enable set is not acyclic, the firing of any one rule in it will disable at least one other rule b (equivalent to removing the rule b from the minimal enable set), thus effectively making it a non-enable set. Furthermore, since there does not exist one vertex in a disable-cycle having at least one minimal enable set satisfying condition (3)(a) or condition (3)(b), rule b cannot become re-enabled again. Thus no rule in each of the cycles in the strongly connected component C can fire more than once, a fact that contradicts our assumption. The case in which a rule stays enabled is treated in the proof step 2.

(4) Proof of [B1, B2, B3, B5 and condition (4) is false $\rightarrow$ a fixed point is always reachable in bounded time].

Assume that the fourth condition is false; that is, there are no two rules in C which assign conflicting values to the same variable. Assume that a fixed point is not always reachable in bounded time. From the proof of special form **B** theorem, a set of rules satisfying conditions B1, B2 and B3 will not reach a fixed point *only if* there is at least one variable v_1 in L whose value is changed infinitely often among a finite number of values. Then there exist at least two rules which assign different values to the variable v_1, a fact that contradicts our assumption.

□

5.5. Special Form C

Before we can describe special form **C**, we need to define the notion of a **subrule** and the concept of a **variable-modification graph**. A **subrule** of rule y with m LHS variables is of the form:

$$c_1 := d_1 \,!\, c_2 := d_2 \,!\, \cdots \,!\, c_p := d_p \;\text{ IF } \textit{test}$$

where each $c_i \in L_y$, d_i is the expression to be assigned to variable c_i in the original rule, and $p \leq m$. A **single-assignment subrule** of rule y is then of the form:

$$c := d \;\text{ IF } \textit{test}$$

where $c \in L_y$, and d is the expression to be assigned to variable c in the original rule. Let $y1$ be a subrule of rule y. Then L_{y1} denote the set of variables in LHS of subrule $y1$, and T_{y1} denote the set of variables in the test of subrule $y1$. Note that $T_{y1} = T_y$. Let $\exp_w$ denote the expression to be assigned to variable w.

Definition 5.5.1.

The **variable-modification (VM)** graph of a set of rules is a labeled directed graph $G = (V, E)$. V is a set of vertices each of which is labeled by a tuple (i, j) corresponding to a distinct single-assignment subrule, where i is the rule number and j is the single-assignment subrule number within rule i (counting from left to right). E is a set of edges each of which denotes the interaction between a pair of single-assignment subrules such that an edge (m, n) connects vertex m to vertex n iff $L_m \cap R_n \neq \emptyset$, i.e., the variable appearing in LHS of single-assignment subrule m also appears in RHS of single-assignment subrule n.

The **VM** graph corresponding to the program segment of example 5.5.1 is shown in Figure 5.9. Note that there are 11 vertices each of which corresponds to a distinct single-assignment subrule found in the program segment. For example, the vertex labeled (2,2) corresponds to single-assignment subrule $b := d - 1$ and the vertex labeled (6,1) corresponds to single-assignment subrule $f := g + d$. There is an edge from the vertex labeled (2,2) to the vertex labeled (1,1) since the variable b appears in $L_{(2,2)}$ and in $R_{(1,1)}$. Similarly, there is an edge from the vertex labeled (5,2) back to itself (a self-loop) since the variable e appears in $L_{(5,2)}$ and in $R_{(5,2)}$.

Example 5.5.1.

```
var
      a, b, c, d, e, f, g, h : integer;

rules
    (* 1. *)     a := b + c + 1   IF   test1
    (* 2. *)  [] a := c + 1 ! b := d - 1   IF   test2
    (* 3. *)  [] b := d + 1 ! c := 0   IF   test3
```

(* 4. *) $[]\, c := b + d + 1$ IF $test_4$

(* 5. *) $[]\, d := a - 1 \,!\, e := e + 1$ IF $test_5$

(* 6. *) $[]\, f := g + d \,!\, g := g + h \,!\, h := f$ IF $test_6$

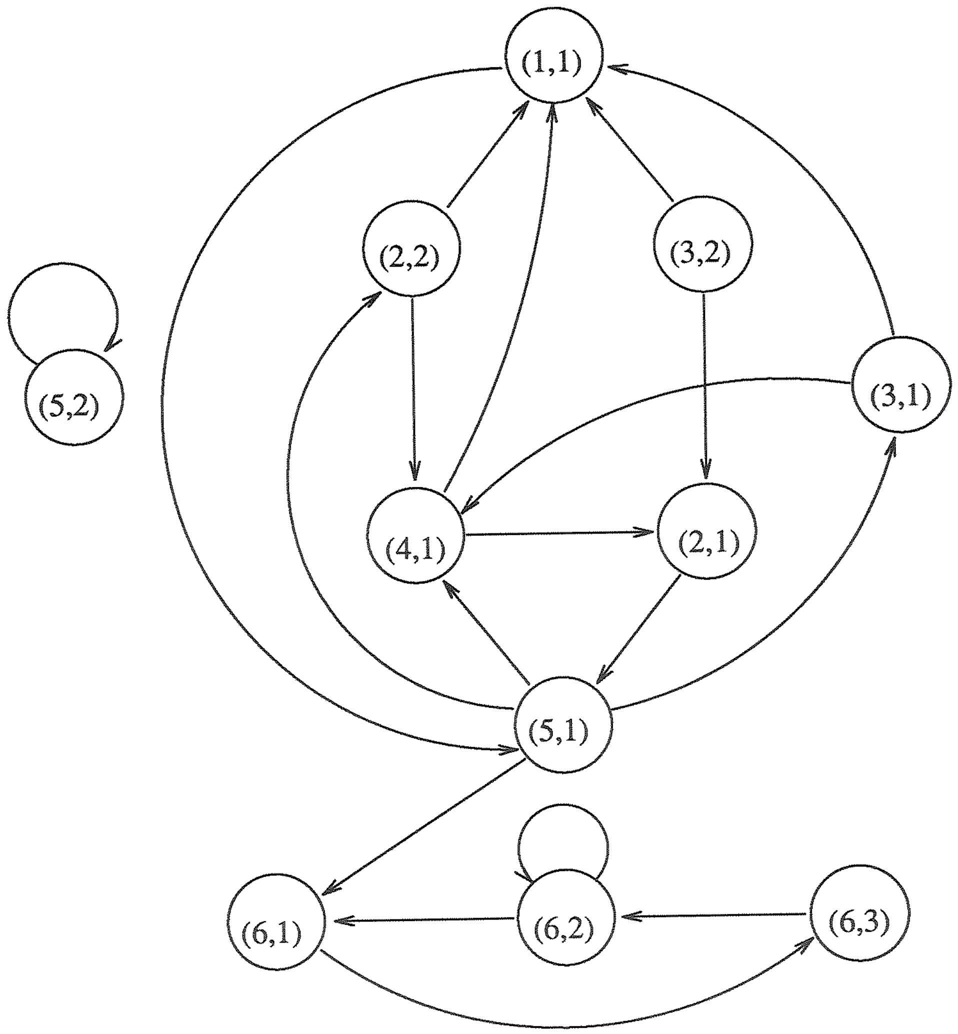


Figure 5.9. VM Graph Corresponding to Program of Example 5.5.1.

5.5.1. Special Form C: Compatible Assignment to Variables, L and T Disjoint

A set of rules are said to be in special form C if all of the following four conditions hold.

(C1) Expressions with variables are assigned to the variables in L, i.e., $R \neq \emptyset$.
(C2) All of the rules are pairwise compatible.
(C3) $L \cap T = \emptyset$.
(C4) For each simple cycle in the variable-modification graph corresponding to this set of rules, there is at least a pair of rules (subrules) in the cycle that are compatible by condition CR1 (mutual exclusivity).

Theorem 5.5.1.

An **EQL** program whose rules are in special form C will always reach a fixed point in a bounded number of iterations.

Note that all rules in a set automatically satisfy the condition C4 iff all rules in this set are compatible pairwise by condition CR1 (mutual exclusivity). Moreover, the domains of the variables may be finite or infinite.

Observe that if the cycle is a self-loop, then there does not exist a pair of rules (subrules) that are compatible by condition CR1 (mutual exclusivity). This occurs if $L_x \cap R_x \neq \emptyset$ for subrule x, i.e., the variable appearing in LHS also appears in RHS of subrule x. This follows from the fact that the cycle consists of a single vertex corresponding to one single-assignment subrule. In the variable-modification graph of Figure 5.9 corresponding to the program segment of example 5.5.1, there are self-loops around vertices labeled (5,2) and (6,2) corresponding to the single-assignment subrules (5,2) and (6,2). Thus the program of example 5.5.1 is not in special form $\mathbb{C}$. In fact, it can be shown that a nontrivial program satisfying conditions C1, C2 and C3 whose variable-modification graph contains self-loop(s) (condition C4 is violated) may never reach a fixed point. A nontrivial program is one which does not contain rules such as: $x := x$ IF *test*, $x := x + 0$ IF *test*, or $x := x - x$ IF *test*.

Furthermore, for a cycle whose component vertices all correspond to the subrules of the same rule, there does not exist a pair of rules (subrules) that are compatible by condition CR1 (mutual exclusivity). This follows from the fact that different subrules of the same rule cannot have mutually exclusive tests since they all have the *same* test. Referring to the variable-modification graph of Figure 5.9 (corresponding to the program segment of example 5.5.1), there is a cycle ((6,1), (6,3), (6,2), (6,1)) whose component vertices all correspond to subrules of the same rule, namely rule 6. Thus the program of example 5.5.1 is not in special form $\mathbb{C}$. It can be shown that a nontrivial program satisfying conditions C1, C2 and C3 whose variable-modification graph contains cycle(s) of this type (condition C4 is violated) may never reach a fixed point.

We now exhibit a program that falls into special form $\mathbb{C}$.

Example 5.5.1.1.

```
var
      a, b, c, d, e, m, n, p : integer;

rules
    (* 1. *)     a := b + c + 1    IF    m > 0 AND n = 0
    (* 2. *)   [] a := c + 1    IF    m <= 0 AND n = 0
    (* 3. *)   [] b := d + 1 ! e := 0    IF    n <> 0 AND p = 5
    (* 4. *)   [] c := b + d + 1 ! e := 1    IF    n = 0 AND p = 5
    (* 5. *)   [] d := a - 1    IF    p <> 5
```

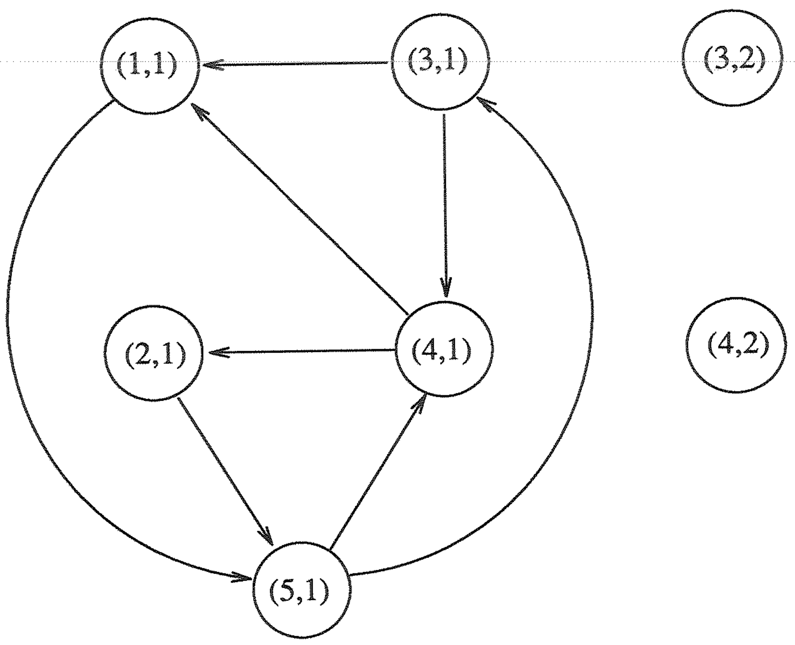


Figure 5.10. VM Graph Corresponding to Program of Example 5.5.1.1.

It is obvious that this set of rules satisfies the conditions C1 and C3 of special form C. It is easy to see that there are five distinct cycles in the VM graph corresponding to the program segment in this example:

cycle 1: ((1,1), (5,1), (3,1), (1,1))
cycle 2: ((2,1), (5,1), (4,1), (2,1))
cycle 3: ((1,1), (5,1), (4,1), (1,1))
cycle 4: ((1,1), (5,1), (3,1), (4,1), (1,1))
cycle 5: ((2,1), (5,1), (3,1), (4,1), (2,1))

Note that every cycle except cycle 1 contains the single-assignment subrules

(4,1) and (5,1), and that rules 4 and 5 are compatible by condition CR1 (mutual exclusivity). Cycle 1 contains the single-assignment subrules 3 and 5, which are also compatible by condition CR1. Since there is at least one pair of rules (subrules) that are compatible by condition CR1 in each cycle, condition C4 of special form **C** is satisfied. Rule 1 and rule 2 do not satisfy condition CR2 since the same variable a appears in both rules. However, these pair of rules are compatible by condition CR1. Rule 3 and rule 4 do not satisfy condition CR2 or CR3 since the same variable e appears in both rules and is assigned conflicting values. However, these pair of rules are compatible by condition CR1. All other rule pairs are compatible by condition CR2 and thus the condition C2 is satisfied. Hence, we can conclude that the set of rules in this program segment is in special form **C** and thus is always guaranteed to reach a fixed point in bounded time.

Proof of Theorem 5.5.1.

Given a set of rules in special form **C**, assume that a fixed point is not always reachable in bounded time. Then by the definitions given in chapter 4, in the state space graph corresponding to the program, there must be (1) at least one cycle, or (2) at least one simple path of unbounded length. Since the expressions in the set R consist of both constant and variable terms, the variables in the program may change in value infinitely among an infinite number of values (for variables with infinite domains) and thus a cycle in the variable-modification graph of the program may correspond to either situation (1) or situation (2) in the state space graph of the program.

Since $L \cap T = \varnothing$ (condition C3), the firing of any rule will not affect the value of any test, i.e., will not enable or disable any rule. Hence, the values of all tests are fixed once the values of all input variables have been read in and remain constant throughout the execution of the program. It can be concluded from this fact that cycles of rule firings cannot be created nor destroyed during the course of rule firings and thus all cycles in the VM graph corresponding to the program, if any, are persistent during the course of rule firings. There are two cases of cyclic firing sequence to consider. A cyclic sequence of rule firings $(r_1, r_2, r_3, \ldots, r_s, r_1)$ occurs if

(Case 1) the firing of rule r_i modifies at least one variable in the RHS of rule $r_{(i \bmod s)+1}$, for $i = 1, \ldots, s$, and changes the value of the expression containing the modified variable(s), and all rules in this cycle are enabled; or

(Case 2) the length of the cycle is at least two, i.e., $s \geq 2$, and there are at least two rules in it which assign different values to the same variable(s).

Let us first consider case 1. The first condition of case 1 is necessary for a cycle to occur since an enabled rule cannot fire again once it has already fired once unless at least one expression on the right side of the assignment has changed its value since the last firing of that rule. The second condition of case 1, that all rules in the cycle are enabled, is also necessary for a cycle to occur since the firing of any rule cannot enable any rule that is not already enabled. This implies that no pair of rules in it are compatible by condition CR1 (mutual exclusivity). Since the firing of any rule cannot possibly enable rules that have not been enabled at the start of the execution of the program, all rules in the cycle must be enabled at the start. However, this requirement violates our assumption (condition C4) that there is at least one pair of rules that are compatible by condition CR1 in each distinct simple cycle. Therefore, a cyclic sequence of rule firings satisfying the conditions of case 1 cannot exist for rules in special form C. It should be noted that rule(s) in this non-cycle may fire more than once, but the number of such firings for any given rule in it is bounded by a finite number that is a function of the length of the non-cycle.

We now consider the second case. For this cycle to occur, at least two of the rules having at least one variable in common in L are not compatible by condition CR1 and this variable is assigned conflicting values in these rules.

However, this violates our compatibility assumption (condition C2) that all rules are pairwise compatible. Hence, it can be concluded that a set of rules in special form **C** is always guaranteed to reach a fixed point in bounded time.

□

5.5.2. Simplifying the Analysis When L and R Disjoint

Given a set S of rules in special form **C**, if $L \cap R = \varnothing$, then all variables in the set R can be treated as constants since these variables do not appear in the set L and thus are never modified during the course of rule firings. As a result, the special form **A** can be applied instead of the special from **C**, reducing the complexity of the recognition procedure as well as allowing the analysis procedure to obtain a more precise bound on the number of rule firings for the set of rules to reach a fixed point. The recognition procedure should therefore first determines whether $L \cap R = \varnothing$ so that it can decide whether to apply the special form **C** or the special form **A**.

5.6. Special Form D

In section 5.5, we have introduced the concept of a variable-modification (VM) graph. To define special form **D**, we extend this graph to include a new type of edges: the **disable** edge. To distinguish the original type of edge from the new **disable** edge, we call the original edge an **LR** (set L $\rightarrow$ set R) edge to reflect the fact that an **LR** edge (m,n) connects vertex m to vertex n iff $L_m \cap R_n \neq \varnothing$, i.e., the variable appearing in the set L of single-assignment subrule m also appears in the set R of single-assignment subrule n. A **disable** edge (r,s) connects vertex r to vertex s iff the firing of rule r always disables the enabling condition of rule s. Special form **D** also makes use of the **ER** graph defined in section 5.3 for the recognition procedure of special form **B**. Let L_x, R_x, and T_x denote respectively the sets of variables appearing in LHS, RHS, and the test (enabling condition) of rule x.

The extended **VM** graph corresponding to the program segment of example 5.6.1 is shown in Figure 5.11. **LR** edges are represented by solid arrows whereas **disable** edges are represented by dashed arrows in the graph. Note that there are 11 vertices each of which corresponds to a distinct single-assignment subrule found in the program segment. For example, the vertex labeled (2,2) corresponds to single-assignment subrule $b := d - 1$ and the vertex labeled (6,1) corresponds to single-assignment subrule $f := g + d$. There is an **LR** edge from the vertex labeled (2,2) to the vertex labeled (1,1) since the variable b appears in $L_{(2,2)}$ and in $R_{(1,1)}$. Similarly, there is an **LR** edge from the vertex labeled (5,2) back to itself (a self-loop) since the variable e appears in $L_{(5,2)}$ and in $R_{(5,2)}$. There is a **disable** edge from the vertices labeled (3,i), $i = 1,2$, to the vertices labeled (6,j), $j = 1,2,3$, since the firing of rule 3 disables the enabling condition of rule 6.

Example 5.6.1.

```
var
      a, b, c, d, e, f, g, h, m, n, p, q : integer;

rules
      (* 1. *)     a := b + c + 1   IF   m > 5
      (* 2. *)  [] a := c + 1 ! b := d - 1   IF   h < 0
      (* 3. *)  [] b := d + 1 ! c := 1   IF   a > 10
      (* 4. *)  [] c := b + d + 1   IF   n < 10
      (* 5. *)  [] d := a - 1 ! e := e + 1   IF   p > 0 AND q > 0
      (* 6. *)  [] f := g + d ! g := g + h ! h := f   IF   c = 0
```

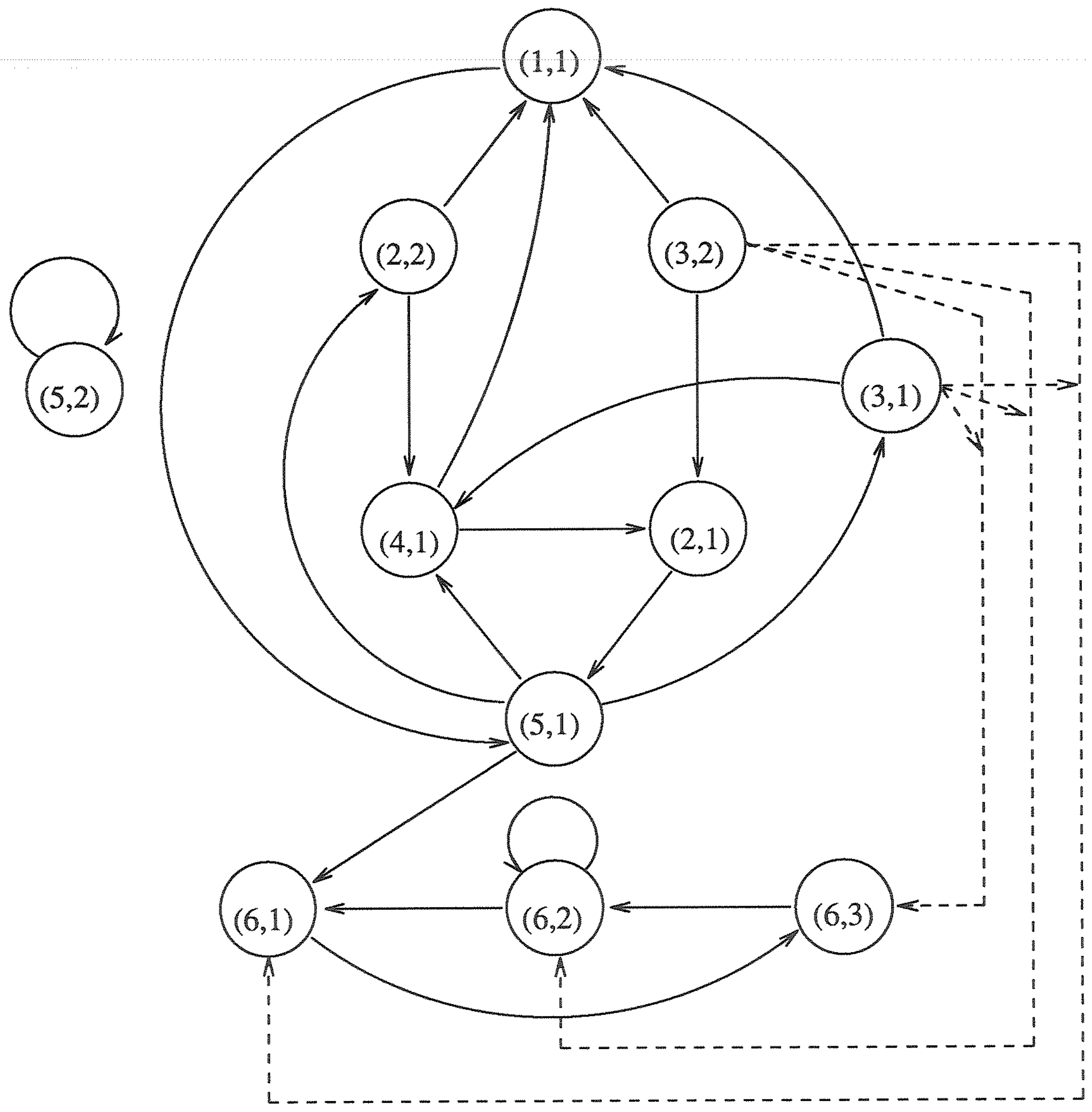


Figure 5.11. VM Graph Corresponding to Program of Example 5.6.1.

5.6.1. Special Form D: Compatible Assignment to Variables, L and T Not Disjoint

A set of rules are said to be in special form **D** if all of the following five conditions hold.

(D1) Expressions with variables are assigned to the variables in L, i.e., $R \neq \emptyset$.
(D2) All of the rules are compatible pairwise.
(D3) $L \cap T \neq \emptyset$.
(D4) For each distinct cycle consisting of **LR** edges only in the variable-modification graph corresponding to this set of rules, there is at least a pair of rules (subrules) in the cycle that are compatible by condition CR1 (mutual exclusivity), or there is at least one **disable** edge in the cycle. (If a **disable** edge connects one *rule* vertex a to another *rule* vertex b, then there is one **disable** edge connecting every *subrule* vertex of rule a to every *subrule* vertex of rule b.)
(D5) For each cycle in the **ER** graph corresponding to this set of rules, no two rules in the cycle assign different expressions to the same variable.
(D6) Rules in disjoint simple cycles (with at least two vertices) in the **ER** graph do not assign different expressions to a common variable appearing in their LHS.

Theorem 5.6.1.

An **EQL** program whose rules are in special form **D** will always reach a fixed point in a bounded number of iterations.

Note that all rules in a set satisfy condition D4 iff all rules in this set are compatible pairwise by condition CR1 (mutual exclusivity). Moreover, the domains of the variables may be finite or infinite.

Observe that if the cycle is a self-loop, then there does not exist a pair of rules (subrules) that are compatible by condition CR1 (mutual exclusivity). This occurs if $L_x \cap R_x \neq \emptyset$ for subrule x, i.e., the variable appearing in LHS also appears in RHS of subrule x. This follows from the fact that the cycle consists of a single vertex corresponding to one single-assignment subrule. In the variable-modification graph of Figure 5.11 (corresponding to the program segment of example 5.6.1), there are self-loops around vertices labeled (5,2) and (6,2) corresponding to single-assignment subrules (5,2) and (6,2). Thus the program of example 5.6.1 is not in special form **D**.

Furthermore, for a cycle whose component vertices all correspond to subrules of the same rule, there does not exist a pair of rules (subrules) that are compatible by condition CR1 (mutual exclusivity). This follows from the fact that different subrules of the same rule cannot have mutually exclusive tests since they all have the same test. Referring to the variable-modification graph of Figure 5.11 corresponding to the program segment of example 5.6.1, there is a cycle ((6,1), (6,3), (6,2), (6,1)) whose component vertices all correspond to subrules of the same rule, namely rule 6. Thus the program of example 5.6.1 is not in special form **D**.

We now exhibit a program that falls into special form **D**.

Example 5.6.1.1.

```
var
        a, b, c, d, e, m, n, p, q : integer;

rules
        (* 1. *)     a := b + c + 1    IF   m > 0 AND n = 0
        (* 2. *)   [] a := c + 1    IF   m <= 0 AND n = 0 AND c = 10
        (* 3. *)   [] b := d + 1 ! e := 0    IF   n <> 0 AND p = 5
(* 4. *)    [] c := b + d + 1 ! e := 1    IF    p = 5 AND d < 0
(* 5. *)    [] d := a - 1    IF    p <> 5 AND q < 0
(* 6. *)    [] q := 2    IF    a > 10
```

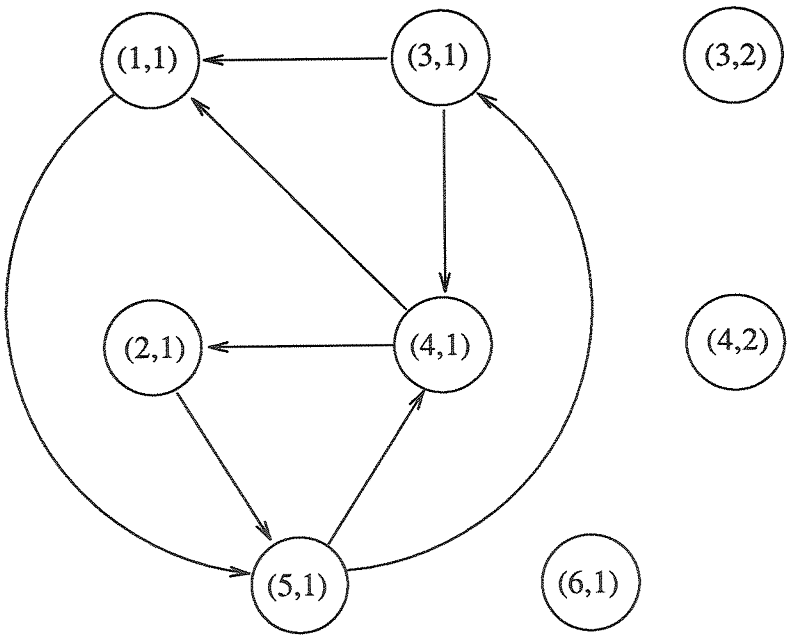


Figure 5.12. VM Graph Corresponding to Program of Example 5.6.1.1.

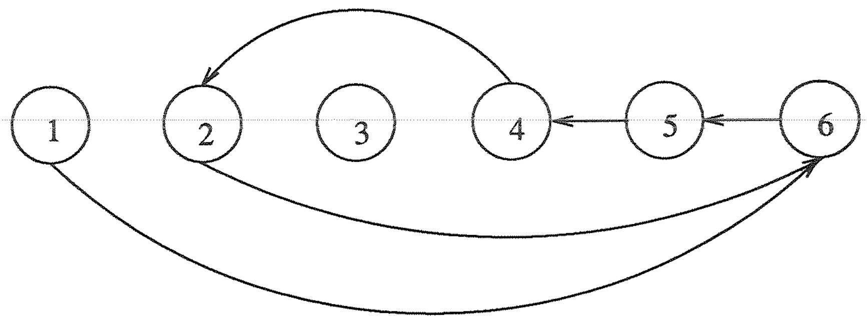


Figure 5.13. ER Graph Corresponding to Program of Example 5.6.1.1.

It is obvious that this set of rules satisfies conditions D1 and D3 of special form **D**. It is easy to see that there are five simple cycles each of which consists of **LR** edges only in the variable-modification graph corresponding to the program segment in this example:

cycle 1: ((1,1), (5,1), (3,1), (1,1))
cycle 2: ((2,1), (5,1), (4,1), (2,1))
cycle 3: ((1,1), (5,1), (4,1), (1,1))
cycle 4: ((1,1), (5,1), (3,1), (4,1), (1,1))
cycle 5: ((2,1), (5,1), (3,1), (4,1), (2,1))

Note that every cycle except cycle 1 contains the single-assignment subrules (4,1) and (5,1), and that rules 4 and 5 are compatible by condition CR1 (mutual exclusivity). Cycle 1 contains the single-assignment subrules 3 and 5, which are also compatible by condition CR1. Since there is at least one pair of rules (subrules) that are compatible by condition CR1 in each cycle, condition D4 of special form **D** is satisfied. The rule pair 1 and 2, and the rule pair 3 and 4 do not satisfy condition CR2 or CR3 since the same variable *e* appears in both rules and is assigned conflicting values. However, these pairs of rules are compatible by condition CR1. All other rule pairs are compatible by condition CR2 and thus the condition D2 is satisfied. In the **ER** graph (Figure 5.13) corresponding to this program, there is one cycle (2,6,5,4,2) but there are no two rules in it that assign conflicting values to the same variable and thus condition D5 is also satisfied. Condition D6 is also satisfied since there is only one cycle. Hence, we can conclude that the set of rules in this program segment is in special form **D** and thus is always guaranteed to reach a fixed point in bounded time.

Proof of Theorem 5.6.1.

Given a set of rules in special form **D**, assume that a fixed point is not always reachable in bounded time. Then by the definitions given in chapter 4, in the state space graph corresponding to the program, there must be (1) at least one cycle, or (2) at least one simple path of unbounded length. Since the expressions in the set R consist of both constant and variable terms, the variables in the program may change in value infinitely among an infinite number of values (for variables with infinite domains), or among a finite number of values (for variables with finite domains). Hence, the state space graph may have an infinite number of states and thus the lengths of some paths may be infinite. Therefore, a cycle in the **VM** graph or a cycle in the **ER** graph of the program may correspond to either situation (1) or situation (2) in the state space graph of the program.

We refer to a cycle consisting of **LR** edges only as an **LR**-cycle, and a cycle consisting of **ER** edges only as an **ER**-cycle. A cycle in the **ER** graph is said to be **good** if and only if no two rules in it assign different values to the same variable. A cycle in the **ER** graph is said to be **bad** if and only if at least two rules in it assign different values to the same variable. An **assignment conflict** is said to have occurred if two rules in a cycle assign different values to the same variable.

The part of the proof that deals with the cycles of firing sequences which consist of **LR** edges only is more complicated than that for special form **C**, owing to the fact that $L \cap T \neq \emptyset$ (condition D3) and hence not all of the rules in a cycle need to be enabled at a given time for this type of cyclic firing sequences to occur.

Since $L \cap T \neq \emptyset$, the firing of a rule may affect the logical value of the enabling condition associated with this rule or other rule(s), i.e., may enable or disable this rule and/or other rule(s). Hence, the logical values of some enabling conditions are not fixed even after the values of all input variables have been read in. It can be concluded from this fact that cycles of rule firings can be created or destroyed dynamically during the course of rule firings. Different patterns of cyclic rule firings may be exhibited given different initial input values and given different orders in which the enabled rules are fired. This greatly complicates the analysis of the rules. At first glance, this may seem to give rise to an extremely large number of cases of cycles and rule interactions. But after a careful case analysis, it is found that there are three general cases of cyclic firing sequences to consider. A cyclic sequence of rule firings $(r_1, r_2, r_3, ..., r_s, r_1)$ may occur if one of the following conditions becomes true.

Case 1: **LR**-cycle. (1) The firing of rule r_i modifies at least one variable in the RHS of rule $r_{(i \; mod \; s)+1}$, for $i = 1, ..., s$, and changes the value of the expression containing the modified variable(s), and [(2a) all rules in this cycle are permanently enabled, or (2b) not all rules are enabled at one time but they become enabled and disabled in such a way that makes this cycle possible].

Case 2: **ER**-cycle. The firing of rule r_i potentially enables rule $r_{(i \; mod \; s)+1}$, for $i = 1, ..., s$, and there are at least two rules in it which assign different values to the same variable.

Case 3: **LR**/**ER**-cycle. A combination cycle consisting of **LR** and **ER** edges.

We next show that these 'cycles' do not lead to infinite rule firings provided that all of the rules in the set satisfy all conditions of the special form **D**.

Case 1: **LR**-cycle.

The first condition of case 1 is necessary for this type of cycles to occur since an enabled rule cannot fire again once it has already fired once unless another firing of that rule will change the value of at least one variable on the left side of the assignment. This may occur if:

> Situation 1: at least one expression on the right side of the assignment has changed its value since the last firing of that rule, and/or
>
> Situation 2: at least one variable on the left side of the assignment has changed its value since the last firing of that rule.

We first consider situation 1. (Situation 2 is treated in the proof steps for case 2 below.) For this type of cycles to occur, one of the following conditions also must be true. We consider each condition ([1.1]-[1.5]) separately.

[1.1] Condition 2a of case 1: All rules in the cycle are permanently enabled. This implies that no pair of rules in it are compatible by condition CR1 (mutual exclusivity) or there is no disable edge in the cycle. However, this requirement violates our assumptions (1) (condition D4) that there is at least one pair of rules that are compatible by condition CR1 in each simple cycle, or (2) there is at least one **disable** edge in each distinct simple cycle. The first assumption implies that there is at least one disabled rule in the cycle. The second assumption implies: suppose there is a **disable** edge (i, j), then firing rule r_i will disable rule r_j and thus not all rules can be enabled permanently. Therefore, a cyclic sequence of rule firings satisfying these conditions cannot exist. The situation in which rule r_j is enabled again by another rule is treated in [1.4] and [1.5] below.

Condition 2b of case 1: Since the firing of any rule can potentially enable (and disable) rules that have not been enabled (disabled) at the start of the execution of the program, it is possible that not all rules are enabled at one time but they become enabled and disabled in such a way that make this **LR**-cycle possible. We shall show that this kind of cycles is not possible for rules in special form **D** in the following proof steps.

[1.2] With at least one pair of rules compatible by condition CR1, it is still possible that they become alternatively enabled so that they can alternatively fire since $L \cap T \neq \emptyset$. However, there must be a bad cycle of **ER** edges in the **ER** graph with at least two rules in it that assign conflicting values to the same variable(s), as will be shown in the proof steps for case 2 below. This violates condition D5 and thus a cyclic sequence of rule firings satisfying these conditions cannot exist. If there does not exist a bad cycle in the **ER** graph and all rules are compatible pairwise, then only one of the rules compatible pairwise by condition CR1 only can be enabled and it stays enabled during the course of rule firings.

[1.3] Suppose that there is at least one pair of rules compatible by condition CR1 in the **LR**-cycle, but there is no bad **ER**-cycle. If there are at least two rules in the **LR**-cycle with conflicting assignments to the same variable, these rules must be compatible by condition CR1. Since there is no bad **ER**-cycle, these rules cannot become alternatively enabled. In fact, only one of these CR1-compatible rules (r_m) stays enabled permanently while the other rules which are compatible with this rule r_m by condition CR1 only are disabled, assuming no other rule interferences. Therefore, no cyclic rule firings can occur under these conditions.

Suppose that there is at least one **disable** edge (j, k) in the **LR**-cycle, but there is no bad **ER**-cycle. Assume that there does not exist a pair of rules in the **LR**-cycle that are compatible by condition CR1. Suppose rule r_k is in this

LR-cycle and is disabled by rule r_j. Suppose rule r_1 enables rule r_k sometime later. There are two cases of possible cyclic rule firings to consider.

[1.4] (1) Rule r_1 is in the **LR**-cycle. Then there must be a variable v in both set L_1 and set L_j, and rule r_1 assigns the value m to v such that $m \in V_{k,v}$, and rule r_j assigns a value n to v such that $n \notin V_{k,v}$. This implies that $m \neq n$ and thus rule r_1 and rule r_j must be compatible by condition CR1. Since there is now at least one pair of rules compatible by condition CR1 in the cycle, the proof arguments [1.1]-[1.3] above apply.

[1.5] (2) Rule r_1 is not in the **LR**-cycle. If rule r_k is enabled repeatedly by rule r_1, rule r_k is also disabled repeatedly by rule r_j. Rule r_k can be enabled infinitely often by rule r_1 if rule r_1 can fire infinitely often -- possible if rule r_1 is itself in an **LR**-cycle or an **ER**-cycle. Rules satisfying all conditions of the special form **D** which are in an **ER**-cycle cannot fire infinitely often, as will be shown in the proof steps for case 2 below. If rule r_1 is in an **LR**-cycle, then it cannot fire infinitely often provided that one of the above conditions [1.1]-[1.4] is true. If rule r_1 is enabled infinitely often by another rule r_2, then rule r_1 is in the same situation as rule r_k above and thus the proof steps above apply. Now suppose rule r_2 is also in *another* **LR**-cycle. We repeat this argument for rules $r_1, r_2, r_3,\ldots.$ Since the number of rules in the set is finite, we will eventually reuse a rule r_i that has already been considered. Since all rules are compatible pairwise and all rules in every cycle satisfy conditions D4 and D5, rules cannot be enabled infinitely often under this condition.

Case 2: **ER**-cycle.

In this case, a set of rules will not reach a fixed point *only if* there is at least one variable v_1 in L whose value is changed infinitely often among a finite number of values (for variables with finite domains), or among an infinite number of values (for variables with infinite domains). Then one of the following conditions is true.

(1) There must be an **LR** cycle (of length one or more) in the **VM** graph.

(2) There exist at least two rules which assign different values to the variable v_1.

The first condition is treated in case 1 above, and the second condition is treated in the proof of theorem 5.3.1 relevant to special form **B**.

Case 3: Cycles with **ER** and **LR** edges.

There are three situations to consider.

[3.1] Joint **ER**- and **LR**- cycles. Both **ER**-cycles and **LR**-cycles have been separately shown not to cause infinite rule firings in case 1 and case 2 above, provided that the set of rules satisfies all conditions of the special form **D**. For each case, all possible interactions among rules in and out of the cycles have been shown not to affect the validity of the proof argument.

[3.2] Partially joint **ER**-cycle and **LR**-cycle. Both **ER**-cycles and **LR**-cycles have been separately shown not to cause infinite rule firings in case 1 and case 2 above, provided that the set of rules satisfies all conditions of the special form **D**. For each case, all possible interactions among rules in and out of the cycles have been shown not to affect the validity of the proof argument.

[3.3] The **ER** edge(s) and the **LR** edge(s) form a cycle with no overlapping **ER** and **LR** edges. Since the **LR** edges do not form an **LR**-cycle, the value(s) of the expression(s) in the set R of at least one rule r_m is(are) fixed and can be treated as constant(s) during the course of rule firings. Consequently, no cyclic rule firings of the type in case 1 can occur. Since all rules are compatible pairwise, even if there are conflicting assignments to the same variable(s) by two or more rules, these rules must be compatible pairwise by condition CR1.

Since the **ER** edges do not form an **ER**-cycle, there cannot be alternative enabling of these CR1-compatible rules, as shown in the proof steps for case 2 above. Thus no cyclic rule firings can occur in this case.

Hence, it can be concluded that a set of rules in special form **D** is always guaranteed to reach a fixed point in bounded time.

□

5.6.2. Simplifying the Analysis When L and R Disjoint

Given a set S of rules in special form **D**, if $L \cap R = \varnothing$, then all variables in the set R can be treated as constants since these variables do not appear in the set L and thus are never modified during the course of rule firings. As a result, special form **B** can be applied instead of special from **D**, reducing the complexity of the recognition procedure as well as making it possible to derive a more precise bound on the number of rule firings to reach a fixed point. The recognition procedure should therefore first determines whether $L \cap R = \varnothing$ in order to decide whether to apply the special form **D** or the special form **B**.

5.7. The General Analysis Strategy

The utility of the special forms described in preceding sections might seem quite limited since all conditions of a special form must be satisfied by the *complete* set of rules in a program. However, the main use of the special forms in our analysis tools is not to identify special-case programs. The leverage of a special form comes about when we can apply it to a subset of rules and conclude that at least some of the variables must attain stable values in bounded time. It is unnecessary for all the rules of a program to be in a special form in order to be able to reduce the state space. The exploitation of the special forms in a general strategy is explained in this section. Our general strategy for tackling the analysis problem is best understood by an example. Suppose only special form

A (see section 5.2) is available in our general analysis tool.

Example 5.7.1.

```
     input: read($b$, $c$)

(* 1. *)    $a1$ := true  IF $b$ = true AND $c$ = true
(* 2. *) [] $a1$ := true  IF $b$ = true AND $c$ = false
(* 3. *) [] $a2$ := false IF $c$ = true
(* 4. *) [] $a3$ := true  IF $a1$ = true AND $a2$ = false
(* 5. *) [] $a4$ := true  IF $a1$ = false AND $a2$ = false
(* 6. *) [] $a4$ := false IF $a1$ = false AND $a2$ = true
```

For this program, $L \cap T \neq \varnothing$ and thus the rules are not in special form **A**. However, observe that rules 1, 2 and 3 by themselves are of the special form and that all the variables in these rules do not appear in LHS of the rest of the rules of the program and thus will not be modified by them. We can readily conclude that the variables $a1$ and $a2$ must attain stable values in bounded time, and these two variables can be considered as *constants* for rules 4, 5 and 6 of the program. We can take advantage of this observation and *rewrite* the program into a simpler one, as shown below.

```
     input: read($a1$, $a2$)

(* 4. *) [] $a3$ := true  IF $a1$ = true AND $a2$ = false
(* 5. *) [] $a4$ := true  IF $a1$ = false AND $a2$ = false
(* 6. *) [] $a4$ := false IF $a1$ = false AND $a2$ = true
```

Note that $a1$ and $a2$ are now treated as input variables. This reduced program is of the special form since all assignments are to constants, L and T are disjoint, and all tests are mutually exclusive. Hence this program is always guaranteed to reach a fixed point in bounded time. This guarantees that the original program must reach a fixed point in bounded time.

The other special forms presented in preceding sections can be exploited in the above fashion. Our general strategy for tackling the analysis problem is as follows.

(1) Identify some subset of the rules which are of a special form (determined by looking up a catalog of special forms) and which can be treated independently. We call a subset of rules **independent** iff its fixed-point convergence can be determined without considering the behavior of the rest of the rules in the program. In other words, the fixed-point convergence of an independent subset of rules does not depend on the behavior of the rest of the rules in the program. Rewrite the program to take advantage of the fact that some variables can be treated as constants because of the special form.

(2) If none of the special forms applies, identify an independent subset of the rules and check the state space for that subset to determine if a fixed point can always be reached. Rewrite the program as in (1) to yield simpler ones if possible.

(3) Perform an analysis on each of the programs resulting from (1) or (2).

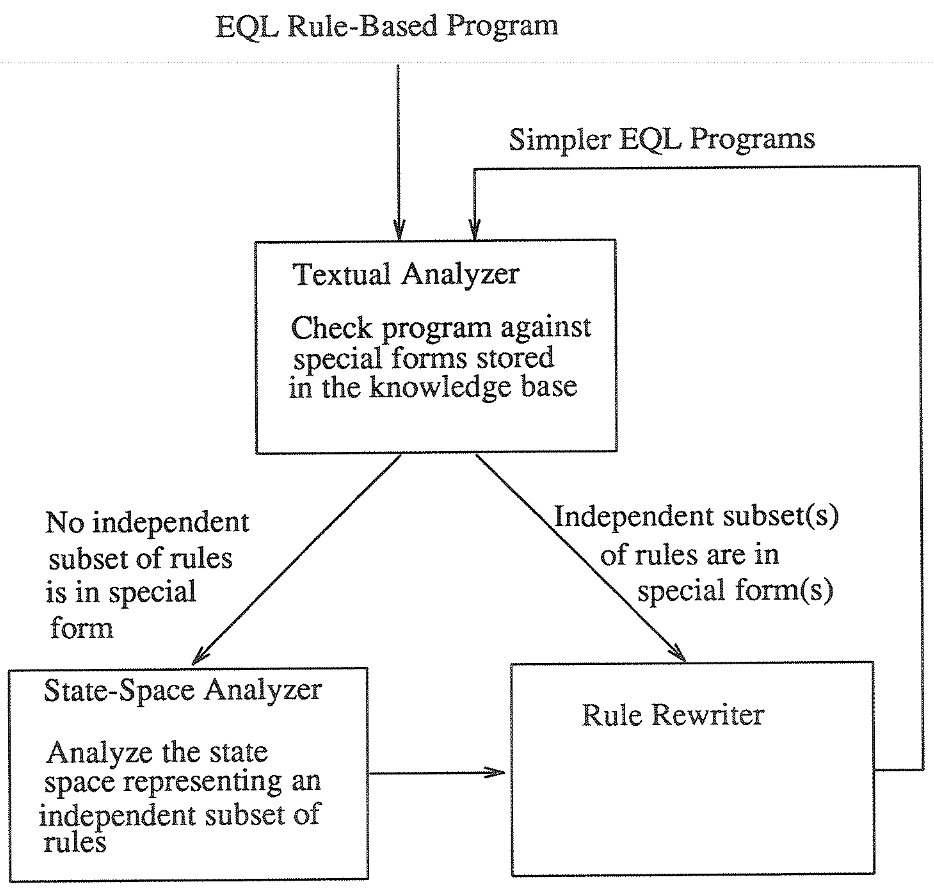

EQL Rule-Based Program
Simpler EQL Programs
Textual Analyzer
Check program against special forms stored in the knowledge base
No independent subset of rules is in special form
Independent subset(s) of rules are in special form(s)
State-Space Analyzer
Analyze the state space representing an independent subset of rules
Rule Rewriter

Figure 5.14. The General Analysis Strategy.

There are several ways by which the subsets of rules are selected in each iteration of the analysis algorithm. We shall outline the simplest selection strategy here. Suppose R is the set of rules at a particular iteration step of the analysis algorithm. Let S be the set of rules in special form **F**, where **F** is one of the special forms catalogued. We first initialize S to be the empty set. A simple rule set selection strategy is to incrementally add a rule i ($i \in R$) to S if the rule set $S \cup \{i\}$ is in special form **F**. Note that special forms which are computationally faster to check are applied first. Starting with the first rule in the program, each rule is checked for inclusion in S in the order it appears in the program. If no rule is in a special form after this first pass, the second rule is checked first, and the above procedure is repeated. In order for a rule i to be added to S, all conditions of special form **F** must be satisfied by the rules in the set $S \cup \{i\}$. The selection algorithm terminates when all rules in the set R are checked in the way described above. The output from the selection algorithm is the set S.

Intuitively, this general analysis strategy allows us to use a special form in the induction step of a proof, by structural induction, that an **EQL** program always takes a bounded number of rule firings to reach a fixed point. Thus relatively restricted special forms may be exploited to analyze a much larger class of programs. We next show how information extracted by this analysis algorithm can be used to compute the numeric response time bound for an **EQL** program.

5.7.1. Derivation of the Program Response Time Bound

A numeric bound on the number of rule firings for a program to reach a fixed point can be easily derived as follows. Note that the analysis algorithm breaks the **EQL** program into smaller rule sets, and the rules in the rule sets selected in iteration i are independent from those in the rule sets selected in iteration j, for $i < j$. This in fact suggests an optimal schedule for firing the rules in the program. Rules in the rule sets selected during the first iteration should be fired first until a fixed point is reached, followed by the firing of those rules in the rule sets selected during the second iteration and later iterations. With this scheduling policy and a uniprocessor system, the bound on the number of rule firings is the sum of all the bounds (associated with the special forms identified or computed by the state space analyzer for non-special-form rule sets) on the number of rule firings of each rule set. A different upper bound can also be easily derived if a different scheduling policy and an alternative run-time architecture are assumed.

Our analysis strategy is general enough to accomodate different scheduling policies and execution architecture, yet it can be easily tailored to specific run-time environment. In addition to the numeric bound computation method described above, there is also a timing analyzer in our analysis system which provides an exact upper bound on the number of rule firings by determining the longest path from an initial state to a fixed point in the state space graph corresponding to the **EQL** program.

In [Browne, Cheng & Mok 88], we reported on the application of a preliminary version of our suite of computer-aided design tools based on this general analysis strategy to an analysis of the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure Control System. The analysis tools employing this general analysis strategy shows a drastic improvement in terms of time and space over tools that primarily rely on exhaustive state-space checking, as evidenced by the two-second time it took for our tools to complete the analysis of the aforementioned rule-based program as compared to a two-week time it took for a state-space-based version of our tools to complete the analysis of only a subset (20 out of 36 rules) of the same program.

5.8. A Note on the Special Forms of Rules

We now take a closer look at the identification of special forms of rules as a technique for speeding up the analysis process. In sequential, control-driven programs, the analysis problem can be solved by the use of bound functions and the recognition of well-founded ordering (e.g., [Dershowitz & Manna 79], [Gries 81]). These techniques can be applied to Prolog and Prolog-style programs owing to the fact that such programs are still control-driven. Rule-based programs, however, are in general nondeterministic in the sense that there is no explicit control flow. The control flow of rule-based programs is embedded in the data and cannot be easily derived. It can also be implementation-dependent. One way to prove bounded-time termination of rule-based programs is to apply model-checking on the state transition graph corresponding to the program. This approach is not viable for large rule-based programs since the analysis procedure has at least exponential time complexity. In section 5.1.1, we have shown that the general analysis problem is undecidable by exhibiting a simple rule-based program that simulates a two-counter machine. Thus any test to guarantee bounded-time termination is a sufficient condition only. This analysis problem has been shown to be PSPACE-complete for rule-based programs with finite domains ([Mok 89]). To overcome this computational complexity problem and still be able to determine whether a fixed point is reachable in bounded time for a large class of rule-based programs, we have developed the technique of identifying special forms of rules for which bounded-time fixed-point reachability is guaranteed. In this section, we examine the techniques employed by bound functions for control-driven programs in relation to our technique of special form identification for rule-based programs. Our aim is to provide a formal framework behind the technique of special form identification.

5.8.1. Bound Functions

In traditional proof of program correctness, a bound function serves the following two purposes: (1) to show that a program (usually with at least one loop) terminates, and (2) to give an upper bound on the number of useful statement executions (or state changes) before the program terminates. Suppose that we are given $\{p\}$ *program* $\{q\}$ and we want to show that q will hold within at most n state changes. A bound function f is a metric that maps program states to natural numbers smaller than n such that, any state change results in q, or preserves p and decreases the value of f. Since a lower bound is associated with the bound function, *program* is guaranteed to terminate within a bounded number of n state changes.

Several strategies (heuristics) have been suggested for finding bound functions:

(1) Use the form of the invariant of a loop as a suggestion for a bound function.

(2) Try to derive the bound function from the notation of the problem and its solution.

(3) Use the lexicographic ordering of certain variables in the program as the bound function.

5.8.2. Special Forms of Rules

Instead of attempting to derive and establish an invariant and a bound function for a program (with loops), the method of identifying special forms of rules tackles the problem of proving progress toward termination by checking the sets of rules in the program to see if they fall into special classes that satisfy the conditions of some special forms. A set of rules found to be in one of the special forms are always guaranteed to reach a fixed point in bounded time.

It is not necessary to use a heuristic or an ad-hoc method to derive an invariant and a bound function appropriate for proving the progress toward termination of a program; only a mechanical application of the recognition algorithm of special forms is needed. The burden of deriving the invariant and the bound function has been transferred to a one-time development of the special forms of rules encompassing a large class of programs and the derivation of the corresponding proofs of correctness.

5.8.2.1. Why Do We Need Special Forms of Rules Instead of Bound Functions?

Although several theorems concerning ordered tuples have been derived to allow the construction of standard bound functions for program loops satisfying certain specifications as specified by the theorems, the bound functions still must be constructed in order to show that an arbitrary program will terminate execution in bounded time. The recognition of lexicographic ordering has been used to mechanize the problem of proving termination of a restricted class of programs. It also has been shown to be useful in certain classes of Prolog or Prolog-style programs in which there is still a basic control flow, namely, top-down left-to-right evaluation of the set of clauses or logical rules ([Ullman & van Gelder 88]). In their method, for rules satisfying certain properties, a set of inequalities whose satisfaction is sufficient for termination of the rules is generated in polynomial time. Then a polynomial test for satisfaction of these inequality constraints is constructed. Their basic technique is to find a well-founded ordering on the procedure calls, that is, a nonnegative integer function of the procedure arguments that is guaranteed to decrease at each recursive call. Note that this integer function is basically a bound function. The question of termination of recursions also has been thoroughly investigated in the context of proving correctness of control-driven programs (e.g., [Dershowitz & Manna 79]).

A sequential, control-driven finite program will not terminate if and only if there is an infinite loop (the testing loop condition for (re)entering the loop is always true). This implies that a program without a loop (in the form of iteration or recursion) will always terminate execution in bounded time. This property also applies to Prolog and Prolog-style programs, with the infinite loop replaced by infinite recursive calling caused by an unbounded argument size ([Ullman 85], [Ullman & van Gelder 88]). This is not the case in rule-based programs, where the absence of infinite looping in the traditional sense does not imply bounded-time termination. A rule-based program may not reach a fixed point in bounded time if in the state space graph corresponding to the program, there is (1) at least one cycle, or (2) at least one simple path of unbounded length. Since the expressions in the set R consist of both constant and variable terms, the variables in the program may change in value infinitely among an infinite number of values (for variables with infinite domains), or among a finite number of values (for variables with finite domains).

Since there is no readily discernible structure or flow of control in rule-based programs, it is extremely difficult, if not impossible, to construct an appropriate bound function for an arbitrary program. It is often the case in rule-based programs that no such bound function can be found since there does not exist a variable whose value is bounded from above (below) and is monotonically increasing (decreasing). The technique of using lexicographic ordered tuples may not be applied since it is often the case that no such ordered tuples can be found. Bound functions are thus not appropriate for proving termination of rule-based programs since there are no control-driven loops or recursive procedure calls in such programs.

One may argue that the method of bound functions still can be applied to prove bounded-time termination for rule-based programs by considering a bound function as defined by either of the following metrics.

(1) f_1: The number of variables which have not reached stable values.

(2) f_2: The number of rules which are enabled (the enabled rules whose firing do not change the value(s) of the variable(s) in LHS are considered disabled for this purpose).

Consider the first bound function f_1. The bound function is bounded by 0, which is the number of variables that have not reached stable values at fixed point, assuming there is one. For programs that will reach a fixed point in a bounded number of rule firings, the value of f_1 is nonincreasing, but may not be monotone nonincreasing. The problem is how to prove or disprove that the value of f_1 decreases for an arbitrary program, which may or may not terminate.

Now consider the second bound function f_2. The bound function is also bounded by 0, which is the number of enabled rules at fixed point, assuming there is one. The value of f_2 may not be monotone nonincreasing, even for programs that will reach a fixed point in a bounded number of rule firings, owing to the fact that rules may become enabled and disabled alternatively during the course of execution of the program. Again the problem is how to prove or disprove that the value of f_2 decreases for an arbitrary program, which may or may not terminate.

Thus these two candidate 'bound functions' are not suitable for showing bounded-time termination of rule-based programs unless the stated problems can be solved systematically. Even though appropriate bound functions can be derived for some very simple rule-based programs with more regular structures (such rule-based programs often can be easily translated into control-driven programs), these bound functions can only be used for these particular programs and must be derived in an often heuristic way. The method of identifying rules in special forms, however, is more amenable to mechanization and thus it is more general.

It is our belief that the application of bound functions which only deal with the value(s) of certain variable(s) and/or argument sizes are not powerful enough to answer the question of whether a rule-based program can be guaranteed to terminate in bounded time. Thus it is necessary to examine the syntactic form as well as the semantic form of the rules in the program. This is accomplished by the identification of rules in special forms, which analyze the relation among all rules in the program. For example, one of the conditions of both special form **A** and special form **B** requires that every pair of rules be compatible (see section 5.2 and section 5.3). The recognition procedures for both special forms thus check the relation among every pair of rules in the program. As another example, the fourth condition (B4) of special form **B** requires that there are no conflicting assignments in any distinct cycles in the **ER** graph corresponding to the program. The recognition procedure for special form **B** thus needs to check the relation among not only pairs of rules, but also sequences and sets of rules.

As indicated earlier, bound functions are also used to show a bound on the number of useful statement executions (or state changes) before the program terminates. For each special form, a bound on the number of rule firings can also derived and thus a program that falls in one of these special forms is guaranteed to reach a fixed point within the time bound specified for that particular special form of rules.

In conclusion, the identification of an arbitrary program (or a set of rules) as in one of the special forms of rules is equivalent to the use of the appropriate bound function to show that the program will terminate execution in bounded time, but the identification of special forms of rules is more appropriate for rule-based programs and is more amenable to mechanization.

5.9. Analysis of Two Expert System Modules

The analysis is in two steps. First, rules in the Space Station Expert System are translated into equivalent rules written in **EQL**. Second, the equivalent set of rules in **EQL** is analyzed using the general analysis strategy discussed in section 5.7. In the following subsections, we report on the translation into **EQL** programs and the analyses of two rule-based modules (originally written in OPS5, CLIPS and Lisp) of the Space Station Expert System:

(1) the Integrated Status Assessment Expert System (ISA), and

(2) the Fuel Cell Monitoring Expert System (FCE).

The EQL versions of both programs appear in appendix D and appendix E.

5.9.1. Translating the Integrated Status Assessment Expert System into an EQL Program

The purpose of the Integrated Status Assessment Expert System (ISA) program is to determine the faulty components in a network. A component can be either an *entity* (node) or a *relationship* (link). A relationship is a directed edge connecting two entities. Components are in one of three states: nominal, suspect, or failed. A failed entity can be replaced by an available backup entity. This expert system makes use of simple strategies to trace failed components in a network. The ISA Expert System consists of:

- 35 EQL rules,
- 46 variables (29 of which are input variables), and
- 12 constants.

Consider the first rule in the ISA program.

```
(P FIND-BAD-RELATIONSHIPS
  (ENTITY ^NAME <NAME> ^STATE << SUSPECT FAILED >>)
  (RELATIONSHIP ^FROM <NAME> ^STATE { <> SUSPECT }
    ^TYPE DIRECT ^MODE ON)
  -->
  (MODIFY 2 ^STATE SUSPECT))
```

This rule says that if entity *A* is in the suspect or failed state and the relationship *L* connects *A* to some other entity, then change the state of *L* to suspect.

The original ISA program is written in the OPS5 rule-based language.

It contains 15 production rules and some Lisp function definitions. Two problems arise during the translation step. The first problem is that actions of several rules call Lisp input/output functions because the original ISA program is an interactive system. For output functions, we simulate them in **EQL** by writing results to extra variables defined for this purpose. Upon reaching fixed point, the `PRINT` command of **EQL** is used to report these results. Input functions which modify some variables during the execution are ignored and values of these variables are fixed initially by defining them as input variables.

The second problem is that the current version of **EQL** has only unstructured variables. However, **OPS5** uses pattern variables to access structured data. This complicates the task of faithfully simulating the original ISA program by an **EQL** program. In order to translate the program, we impose two restrictions on the ISA system so that the use of pattern variables can be avoided. The first restriction concerns the topology of the network. The connectivity of entities and relationships have to be decided at compile time and it cannot be changed during the execution. The other restriction is related to the backup entities. The original ISA program can assign any available backup entity to replace a faulty entity and reconstructs that part of the network. In the translated program, the backup entities are assigned at compile time and cannot be reassigned to replace any other entities. The current **EQL** version of the ISA program is based on the following topology. There are four entities (`entity1` to `entity4`) and three relationships (`rel1` to `rel3`). `rel1` is from `entity1` to `entity3`, `rel2` is from `entity2` to `entity3`, and `rel3` is from `entity3` to `entity4`. Each entity may have zero or one backup. Whether entities have backups are specified by input variables `backup1` to `backup4` respectively.

All the attributes of class *entities* and *relationships* of **OPS5** are now represented by variables in **EQL**. For example, the working memory class

```
(ENTITY ^NAME ^STATE ^MODE)
```

is decomposed into variables `state1` to `state4` and `mode1` to `mode4` with the attribute NAME being "1" to "4". Actually, there are no variables corresponding to NAME explicitly. By using `state2`, for example, we actually refer to the state of the entity of the name "2". Again, we use the first rule from original ISA program as an example to illustrate how it is translated into **EQL** rules.

```
rel1_state := suspect IF (state1 = suspect OR state1 = failed)
      AND rel1_state <> suspect AND rel1_mode = on AND rel1_type
      = direct
[] rel2_state := suspect IF (state2 = suspect OR state2 =
      failed) AND rel2_state <> suspect AND rel2_mode = on AND
      rel2_type = direct
[] rel3_state := suspect IF (state3 = suspect OR state3 =
      failed) AND rel3_state <> suspect AND rel3_mode = on AND
      rel3_type = direct
```

Note that it is translated into three **EQL** rules. Each rule checks the status of one *relationship*. This can be done only when the topology of the network is fixed.

In order to handle pattern variables and structured data, a new language called **MRL** which is an extension to **EQL** has been developed in [Wang, Mok & Cheng 90] and it has the expressive power of **OPS5**. However, the analysis of an **MRL** program is nearly as easy as the analysis of an **EQL** program.

5.9.2. Translating the Fuel Cell Expert System Program into an EQL Program

The purpose of the Fuel Cell Expert System (FCE) program is to determine the status of the different components of the fuel cell system based on current sensor readings and previous system state values. Then it displays the corresponding diagnostics according to the evaluation of the status of the different components of the system.

The FCE program is organized into three main sections: (1) the meta rule section (testFCE1.eql), (2) 12 ordinary rule classes (testFCE2.*.eql), and (3) the output section (testFCE3.eql). This expert system contains:

- 101 EQL rules,
- 56 program variables,
- 130 input variables, and
- 78 constants.

The description of the rule template on page 1 of the program listing and the way the program is organized into different rule classes suggest that the execution of the program proceeds by first firing the set of meta rules until a fixed point is reached and thus a rule class is selected for firing, and then firing those rules in the selected rule class until a fixed point is reached there. The template for the meta rules is:

```
( [ premise(s) ]
  [ consequence 1 - load a rule file ]
  [ consequence 2 - add to text string ]
  [ consequence 3 - set new rule base ] )
```

The corresponding **EQL** program thus reflects the organization and the firing sequences of the original FCE program.

5.9.2.1. Statements Translated

We translate into **EQL** rules only those statements that are relevant to the question of whether the program is always guaranteed to reach a fixed point in a bounded number of rule firings. For example, we translate assignment statements such as

```
(setq STATUS1 'COOL.LP.PROB)
```

and enabling conditions such as

```
(and (= STK.T.STATUS3 'HIGH.START)
     (= STK.T.DISCONN3 'OK))
```

However, we ignore most output statements such as

```
(princ (sys:time))
```

and

```
(princ "FC3 COOLANT LOOP PROBLEM.  FC3 CAN NOT BE USED WITH")
```

The rationale behind this selective translation policy is that the assignment statements and the enabling conditions in those rules in which at least one assignment statement appears may affect the outcome of the execution of the program and thus may determine whether a fixed point is reachable or not in bounded time. However, output and formatting statements such as those exhibited above do not affect the fixed-point reachability of the program.

5.9.2.2. Translating Quoted Values and Variables

Each distinct quoted value in the original FCE program is declared as a distinct constant in the CONST section of the corresponding **EQL** program so that is can be referred to in the remainder of the **EQL** program. Constants in the **EQL** program are further classified into two types: *rule class* constants and *regular* constants. A constant of the former type is used to name one of the many rule classes appearing in the FCE program whereas a constant of the latter type is used to identify an ordinary quoted value. All non-quoted values, which do not appear to be declared in the original FCE program listing, are treated as variables in the corresponding **EQL** program. To allow the **EQL** program to execute, we consider as input variables those variables which do not appear on the left side of any assignment statements in the original program.

5.9.2.3. Translating Premises and Consequences

The translation from the rule-based language used in the Fuel Cell Expert System into **EQL** is described by the following example. Rule #117 in the Fuel Cell Expert System

```
; #117
;
; premises
;
(and (= STATUS 'DISCONN.V.LOW)
     (= SU.HTR 'INH)
     (or (= VOLTAGE 'OK)
         (= VOLTAGE 'LOW1)))
;
; consequences
;
(princ "FC")
(princ FC.ID)
(princ "VOLTS LOW WITH STARTUP HTR INHIBITED.
         POSSIBLE FC")
(terpri)
(keep_line_count)
(princ "INTERNAL SHORT.  ADVISE CRE TO CLOSE FC")
(princ FC.ID)
(princ "REAC VLVS UNTIL"))
```

```
(terpri)
(keep_line_count)
(princ "COOL P <15, THEN SHUTDN FC")
(princ FC.ID)
(terpri)
(keep_line_count)
(princ ">>>>>>")
(princ (sys:time))
(terpri)
(page_pause)
(terpri)
(page_pause)

(setq LOAD.STATUS 'INTERNAL.LD)
```

is translated into the corresponding **EQL** rule

```
load_status := internal_ld
   IF status = disconn_v_low AND
       su_htr = inh AND
       (voltage = ok OR voltage = low1)
       AND work_rule_base = fc_stack_t7_lb
```

The translation of the enabling condition of the FCE rule into the corresponding **EQL** enabling condition is a straightforward exercise in translating syntax. `INTERNAL.LD` is a quoted value in the original rule and thus it is replaced by a constant in the corresponding **EQL** rule. The '`setq`' statement in the original rule is translated into the assignment statement in **EQL**.

5.9.3. The Analysis

The analysis makes use of the special form **A** and the special form **B** defined earlier.

5.9.3.1. The Analysis of the Integrated Status Assessment Expert System

During the first iteration of the general analysis procedure, two bad cycles have been identified: (10→18→34→10) and (10→18→35→10). This indicates that the program may not reach a fixed point given a particular initial state. The rules involved in these two cycles are reproduced below.

```
(* 10 *)
[] state3 := failed IF find_bad_things = true AND state3 =
     suspect AND NOT (rel1_state = suspect AND rel1_mode = on
     AND rel1_type = direct) AND NOT (rel2_state = suspect AND
     rel2_mode = on AND rel2_type = direct)
(* 18 *)
[] state3 := nominal ! reconfig3 := true IF state3 = failed AND
     mode3 <> off AND config3 = bad
(* 34 *)
[] sensor3 := bad ! state3 := suspect IF state1 = suspect AND
     rel1_mode = on AND rel1_type = direct AND state3 = nominal
     AND rel3_mode = on AND rel3_type = direct AND state4 =
     suspect AND find_bad_things = true
(* 35 *)
[] sensor3 := bad ! state3 := suspect IF state2 = suspect AND
     rel2_mode = on AND rel2_type = direct AND state3 = nominal
     AND rel3_mode = on AND rel3_type = direct AND state4 =
     suspect AND find_bad_things = true
```

5.9.3.2. The Analysis of the Fuel Cell Expert System

The Fuel Cell Expert System (FCE) consists of three main sections: (1) meta rule section (testFCE1.eql), (2) 12 ordinary rule classes (testFCE2.*.eql), and (3) output section (testFCE3.eql). The analysis results are as follows:

testFCE1.eql: rules are not compatible.

testFCE2.1.eql: 1 rule is in special form B.
testFCE2.2.eql: 3 rules are in special form B.
testFCE2.3.eql: rules 22-25 are in special form A.
rules 19-21 are in special form B.
rules 18 and 26 not compatible.
testFCE2.4.eql: 3 rules are in special form B.
testFCE2.5.eql: rules 44 and 45 not compatible.
testFCE2.6.eql: 5 rules are in special form B.
testFCE2.7.eql: 6 rules are in special form A.
testFCE2.8.eql: rules 57-60 are not compatible.
testFCE2.9.eql: rules 66 and 67 are in special form A.
rules 64, 65, 68, 69 are in special form B.
rules 61, 62, 63 are not compatible.
testFCE2.10.eql: 1 rule is in special form A.
testFCE2.11.eql: rules 78-89 are not compatible.
testFCE2.12.eql: 4 rules are in special form B.

testFCE3.eql: 30 rules are in special form A.

It should be noted that in the meta rule section (testFCE1.eql), each meta rule fires at most once and thus a fixed point will be reached in bounded time even though the rules are not compatible. In module testFCE2.3.eql, rules 18 and 26 may alternately fire infinitely often without reaching a fixed point. In module testFCE2.5.eql, rules 44 and 45 may alternately fire infinitely often without reaching a fixed point. In module testFCE2.8.eql, rules 57-60 may fire infinitely often without reaching a fixed point. In module testFCE2.11.eql, rules 78-89 may fire infinitely often without reaching a fixed point.

Chapter 6

Characterizing Classes of EQL Programs Analyzable by the General Analysis Algorithm

In this chapter, our goal is to characterize the classes of **EQL** programs, in terms of the type(s) of their corresponding dependency graphs, for which the general analysis algorithm with the recognition of special forms of rules can determine with certainty whether or not a program in these particular classes is always guaranteed to reach a fixed point in bounded time. In chapter 5, we have derived several sets of special form conditions which, if satisfied by a program p, guarantee that p will always reach a fixed point in a bounded number of rule firings. However, a program q which fails to satisfy all conditions specified by a special form does not imply that q will not always reach a fixed point in a bounded number of rule firings. It may be the case that q will always reach a fixed point in a bounded number of rule firings but does not satisfy all conditions of any one of the known special forms.

The power of the recognition procedure of special forms of rules is further augmented by the use of the *General Analysis Algorithm*, which does not require the entire program p to satisfy all conditions of a special form in a single step of the analysis algorithm and thus increases considerably the set of programs with bounded response time that can be analyzed and recognized. Again, a program r that cannot be recognized as a program with bounded response time by the general analysis algorithm does not mean that r definitely does not always have bounded response time. For a review of the notations and definitions used in this chapter, the reader is referred to chapter 5.

6.1. Representing an EQL Program as a Dependency Graph

We characterize an EQL program in terms of the type of its corresponding dependency graph.

Definition 6.1.1.

The **dependency graph** of a program p is a directed graph $G(V, E)$ where each node in V represents a distinct variable of p, and an edge in E connects node x_i to node x_j iff x_i appears in the right side of an assignment in p whose left side contains x_j. An edge is said to be of type TL iff x_i appears in the test. An edge is said to be of type RL iff x_i appears in the RHS.

In the remainder of this chapter, we use the terms *variable* and *node* when there is no ambiguity. All edges appearing in the dependency graphs corresponding to the programs in both class 1 and class 2 described below are of type TL and thus the type qualification is omitted.

6.2. Class 1: EQL Programs Whose Graphs Are Bipartite With Unidirectional Edges

EQL programs which assign constant expressions to left-side variables and whose set L and set T are disjoint always can be analyzed using the *General Analysis Algorithm* with only the recognition procedure for special form A. First, some definitions are in order.

Definition 6.2.1.

A graph $G(V, E)$ is said to be **bipartite** if its nodes can be partitioned into two sets M and N such that no two nodes in M or in N are adjacent, i.e., all edges extend between M and N. A bipartite graph $G(V, E)$ is commonly denoted as $G = (M, N, E)$ where $V = M \cup N$.

Definition 6.2.2.

A directed bipartite graph $G = (M, N, E)$ is said to have only **unidirectional** edges iff either all of its edges are directed from M to N or all of its edges are directed from N to M.

Theorem 6.2.1.

Any **EQL** program which assigns only constant expressions to LHS variables and whose dependency graph is bipartite with unidirectional edges only can be analyzed by the *General Analysis Algorithm* with only the recognition procedure for special form **A**. A program having this property is said to be in class 1.

Proof:

To prove that any class-1 program p can be analyzed by the general analysis algorithm with the recognition procedure of special form **A** only, we need to show that either

(1) p is in special form **A** or p can be recognized as a program which will always reach a fixed point in bounded time by the general analysis algorithm with the recognition procedure for special form **A** only, or

(2) p does not always reach a fixed point in bounded time and thus cannot be recognized by the general analysis algorithm with the recognition procedure for special form **A**.

That is, we need to prove that any class-1 program p is recognizable by the general analysis algorithm with only the recognition procedure for special form **A** if and only if p will always reach a fixed point in bounded time. The *only if* part is proved in chapter 5; the *if* part will be proved next.

Assume that there exist a class-1 program q which always reaches a fixed point in bounded time but cannot be recognized as such by the general analysis algorithm with the recognition procedure for special form **A**. Since q only assigns constant expressions to left-side variables, condition A1 of special form **A** is satisfied. Since the dependency graph $G = (M, N, E)$ of q is bipartite with unidirectional edges only, without loss of generality, let M be the set of variables appearing in the set L and let N be the set of variables appearing in the set T. This implies that $L \cap T = \emptyset$ and thus condition A3 of special form **A** is also satisfied. Hence, there exist at least two rules in q that are not compatible (condition A2 of special form **A** is violated) for q to be non-recognizable. These two rules are of the form

$$x := E_1 \text{ IF } C_1$$
$$x := E_2 \text{ IF } C_2$$

where E_1 and E_2 are constant expressions and C_1 and C_2 are logical expressions, such that $E_1 \neq E_2$, and C_1 and C_2 are not mutually exclusive. Obviously, these two rules will fire alternately without ever reaching a fixed point. Even if the rest of the rules in q are independent from these two rules and are in special form **A**, q is not always guaranteed to reach a fixed point in bounded time. This is a contradiction to our assumption and thus a class-1 program which cannot be recognized by the general analysis algorithm with special form **A** is not always guaranteed to reach a fixed point in bounded time. This completes the proof of the *if* part of the theorem.

□

The general analysis algorithm with the recognition procedure for special form **A** only is capable of analyzing a larger class of programs, as we shall see in the following section.

6.3. Class 2: EQL Programs Whose Graphs Are Acyclic and Layered With Unidirectional Edges

In this section, we show that a large class of **EQL** programs which assign constant expressions to left-side variables and whose set L and set T are not disjoint can be analyzed using the *General Analysis Algorithm* with the recognition procedure for special form **A** only. First, some definitions are in order.

Definition 6.3.1.

A directed graph $G(V, E)$ is said to be **layered** if its nodes can be partitioned into disjoint sets such that no edge connects two nodes in the same set, i.e., no two nodes in a set are adjacent, and if these sets can be drawn as a linear list of sets so that each edge connects only those nodes in adjacent sets. Each set is called a layer. A layered graph is a restricted type of n-partite graphs. A graph whose (disconnected) components are all layered graphs is a layered graph.

An example of an **EQL** program whose graph has the above property is the Space Shuttle Vehicle Cryogenic Hydrogen Pressure Monitoring Expert System (Appendix C).

Theorem 6.3.1.

Any **EQL** program which assigns only constant expressions to LHS variables and whose dependency graph is acyclic and layered with unidirectional edges only can be analyzed by the *General Analysis Algorithm* with the recognition procedure for special form A only. A program having this property is said to be in class 2.

Proof:

To prove that any class-2 program p can be analyzed by the general analysis algorithm with the recognition procedure of special form **A** only, we need to show that any class-2 program p is recognizable by the general analysis algorithm with only the recognition procedure for special form **A** if and only if p will always reach a fixed point in bounded time. The *only if* part is proved in chapter 5; the *if* part will be proved next.

Assume that there exist a class-2 program q which always reaches a fixed point in bounded time but cannot be recognized as such by the general analysis algorithm with the recognition procedure for special form **A**. Since q only assigns constant expressions to left-side variables, condition A1 of special form **A** is satisfied. If the dependency graph of q has more than two layers, then at least one variable appears in the set L and in the set T of q and thus the entire q cannot be in special form **A**. However, we shall prove that q can be analyzed by the general analysis algorithm with the recognition procedure for special from **A** only. If the dependency graph of q has only two layers, then q is in class 1 and was previously shown to be analyzable by the general analysis algorithm with the recognition procedure for special from **A** only. The case in which the dependency graph of q has more than two layers is considered next.

Each edge in the dependency graph actually corresponds to a rule in the program q. Note that more than one edge may correspond to the same rule since multiple variables may appear in the left side and/or in the right side test of an assignment. In the drawing of the graph corresponding to q as a linear list of layers, we label the layers by calling the layer with no in-edges as layer 1, the layer with in-edges directed from the nodes in layer 1 as layer 2, the layer with no out-edges as layer n if the number of layers is n, and in general, the layer with in-edges directed from the nodes in layer $i - 1$ as layer i, $i = 2, ..., n$.

With this representation, the rules corresponding to the edges connecting the nodes in layer i and in layer $i + 1$ (claim 1) must be recognized by the general analysis algorithm as either in special form A (if those rules are always guaranteed to reach a fixed point in bounded time) or not in special form A (if those rules are not always guaranteed to reach a fixed point in bounded time), and (claim 2) must be independent from all rules corresponding to those edges between layer $i+1$ and layer n in iteration i of the analysis algorithm, for $i = 1, \ldots, n-1$.

The first claim follows from the proof of theorem 6.2.1. The rules corresponding to the edges connecting the nodes in layer i and in layer $i+1$ must be in special form A by our assumption that q will always reach a fixed point in bounded time. If not, then at least two of these rules are not compatible -- the only way for these rules to be not in special form A, but this would imply that the entire program may not always reach a fixed point in bounded time.

We prove the second claim by contradiction. Assume that the rules corresponding to the edges connecting the nodes in layer j and those in layer $j+1$ are not independent from at least one of the rules corresponding to an edge between layer $j+1$ and layer n, for some $j = 1, \ldots, n-2$. For rules which assign only constant expressions to LHS variables, two conditions for establishing independence of a set of rules from another are used by the general analyzer. We shall show that the violation of each of these conditions will lead to a contradiction in the proof, thus establishing claim 2.

(1) Suppose that the firing of a rule b corresponding to an edge between layer $j+1$ and layer n changes the content of a variable in L_a, and a corresponds to an edge between layer j and layer $j+1$. Then rule b assigns the value m_2 to variable x appearing in the left side of rule a, and rule a assigns the value m_1 to x such that $m_1 \neq m_2$. This implies that variable x appears in layer $j+1$ and in a layer between $j+2$ and n, inclusive. This is possible only if more than one node may represent the same variable. However, from the definition of the dependency graph of a program, each node corresponds to a distinct variable in the program.

(2) Suppose that the firing of a rule c corresponding to an edge between layer $j+1$ and layer n enables a rule a which corresponds to an edge between layer j and layer $j+1$. This is possible only if there is an edge from layer k to layer j such that $k > j$. However, this violates the property that the acyclic layered graph of q has unidirectional edges only.

This completes the proof of the *if* part of the theorem.

$\square$

As part of our ongoing research effort, we are identifying more classes of **EQL** programs analyzable by the general analysis algorithm when special forms **B**, **C**, and **D** are also included in the analyzer's knowledge base.

Chapter 7

Estella: A Facility for Specifying Behavioral Constraint Assertions in Real-Time Rule-Based Systems

In chapter 5, we have developed a powerful and efficient analysis methodology for a large class of rule-based **EQL** programs to determine whether a program in this class has bounded response time. In particular, we have identified several sets of general behavioral constraint assertions: an **EQL** program which satisfies all constraints in one of these sets of assertions is guaranteed to have bounded response time. We now enhance the applicability of our analysis technique by introducing a facility with which the rule-based programmer can specify application-specific knowledge in the language **Estella** in order to validate the performance of an even wider range of programs. This facility can be viewed as a computer-aided software engineering (CASE) tool for aiding in the rapid prototyping and development of expert systems with guaranteed response time. In this chapter, we shall also describe efficient algorithms for implementing the General Analysis Tool.

7.1. Background and Motivation

Since the verification of the constraint assertions in the special forms of rules is based on static analysis of the **EQL** rules and does not require checking the state space graph corresponding to these rules, our analysis methodology makes the analysis of programs with a large number of rules and variables feasible. A suite of computer-aided software engineering tools based on this analysis approach have been implemented and have been used successfully to analyze several real-time expert systems developed by Mitre and NASA for the Space

Shuttle and the planned Space Station.

We now enhance the applicability of our analysis technique by introducing a facility with which the rule-based programmer can specify application-specific knowledge in order to validate the performance of an even wider range of programs. The idea is to provide the rule-based programmer a language called **Estella** that has been designed for specifying behavioral constraint assertions about rule-based **EQL** programs. These application-specific assertions capture the requirements for achieving certain performance levels for a rule-based program in a particular application, and are used by the general analyzer to determine whether an **EQL** program has bounded response time. This facility can be viewed as a computer-aided software engineering (CASE) tool for aiding in the rapid prototyping and development of expert systems with guaranteed response time.

In the remainder of this section, we explain the motivation for using **Estella** to specify behavioral constraint assertions that guarantee any program satisfying these constraints will have bounded response time. In section 7.2, we show how **Estella** can be used in conjunction with the *General Analysis Tool* **GAT** developed in chapter 5 to analyze **EQL** programs and to facilitate the development of **EQL** systems with guaranteed response time. In section 7.3, we demonstrate the practicality of the **GAT-Estella** facility as a computer-aided software engineering tool by analyzing two real rule-based systems developed by Mitre and NASA for the planned Space Station. In section 7.4, we discuss efficient algorithms for implementing the **GAT-Estella** facility. Section 7.5 is the conclusion.

7.1.1. Specifying Behavioral Constraint Assertions in Estella

Estella is a language for specifying behavioral constraint assertions (BCAs) about **EQL** programs. Both general BCAs and application-specific BCAs can be specified in **Estella**. General BCAs are stored in a permanent knowledge base whereas user-defined BCAs are placed in a temporary knowledge base. The rule-based programmer can specify different application-specific BCAs as input to the **Estella** front-end for the purpose of analyzing **EQL** programs in different application domains.

Once these application-specific BCAs are entered as input to the **Estella** front-end, and an **EQL** program is read into the **GAT** analyzer, the analyzer will make use of the general analysis algorithm, the general BCAs, and the programmer-defined application-specific BCAs to determine whether the rules in the **EQL** program will always reach a fixed point in bounded time. The details of **GAT** are discussed in chapter 5. Specific algorithms used in the **GAT-Estella** System will be discussed in a later section. For ease of discussion, we sketch the main steps of the general analysis algorithm here:

(1) Identify some subset of the rules which are of a special form of BCAs (determined by looking up a catalog of special forms of BCAs) and which can be treated independently. We call a subset of rules **independent** iff its fixed-point convergence can be determined without considering the behavior of the rest of the rules in the program. Rewrite the program to take advantage of the fact that some variables can be treated as constants because of the special form.

(2) If none of the special forms of BCAs applies, identify an independent subset of the rules and check the state space for that subset to determine if a fixed point can always be reached. Rewrite the program as in (1) to yield simpler ones if possible.

(3) Perform an analysis on each of the programs resulting from (1) or (2).

If the analyzer detects that the **EQL** program may not always have bounded response time, it will report a list of rules which may cause a cycle in the state-space graph corresponding to the program. If after examining the rules in question the programmer concludes that those rules which are involved in the apparent timing violations are actually acceptable (for instance, the scheduler knows about these cycles), he/she can interactively specify additional BCAs as input to the analyzer and then re-analyze the **EQL** program. The **EQL** general analyzer can be thought of as a compiler module for **EQL** programs since it checks whether a program is correct with respect to the timing requirement constraints. Figure 7.1 shows the **GAT-Estella** facility for analyzing **EQL** programs. We shall describe in detail the components of the **Estella** facility in section 7.2.

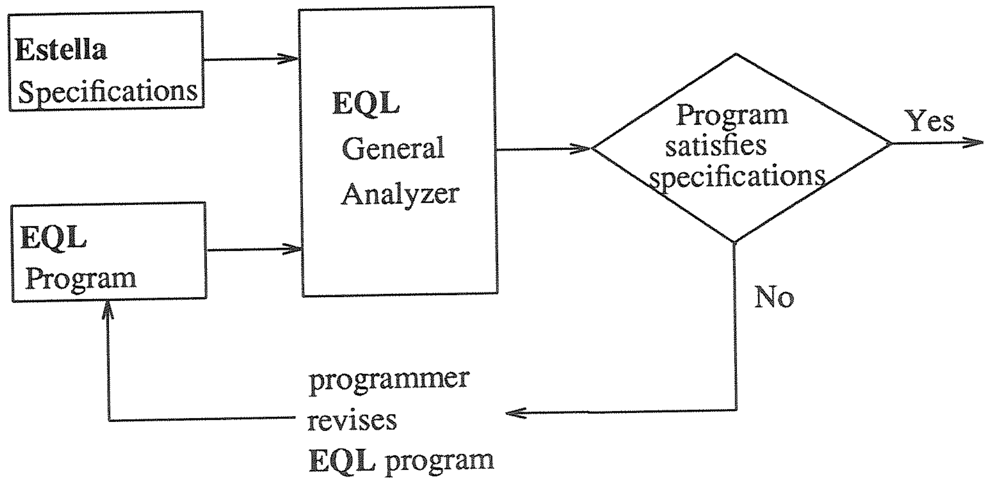

Estella Specifications
EQL Program
EQL General Analyzer
Program satisfies specifications
Yes
No
programmer revises EQL program

Figure 7.1. Checking Behavioral Constraint Assertions Specified in Estella.

The following example demonstrates the utility of the **Estella** facility for specifying behavioral constraint assertions on rule-based systems.

Example 7.1.1.1. Specifying Compatibility Criteria for Rules in the Simple

Object-Detection Program.

The program in example 3.1 is not in any of the special forms of BCAs described in chapter 5. As explained earlier, two pairs of rules are not compatible: rule 1 and rule 4, rule 2 and rule 3. The variables sensor_a and sensor_b contain respectively the values as detected by radar sensor a and by radar sensor b, both of which are directed to the same region of the sky.

Now suppose the rule-based programmer knows that the device for checking the statuses of both sensors is fail-safe; that is, it returns the value 'good' for sensor_x_status if and only if sensor_x is good, where x = a or b, and returns 'bad' if and only if sensor_x is bad or the checking device has malfunctioned. This fact implies that for rule 1 and rule 4 (as well as for rule 2 and rule 3), if the sensor values disagree, then one of the sensor statuses must be bad. Thus only one rule can be enabled at one time and no infinite firings can occur. Hence, these two pairs of rules need not be compatible in the sense defined in chapter 5 for this program to reach a fixed point in a bounded number of firings. The **Estella** statement for specifying this condition is:

```
COMPATIBLE_SET = ({1,4}, {2,3})
```

This statement states that the pair of rules 1 and 4, and the pair of rules 2 and 3 are specified to be compatible even though they do not satisfy the predefined conditions of compatibility. This **Estella** compatibility condition can be used in conjunction with special form A to identify this program as one that is guaranteed to reach a fixed point in bounded time. Next, we describe an example involving a real-time expert system developed by Mitre for the planned NASA Space Station.

Example 7.1.1.2. Specifying Cycles and Break Conditions for the ISA Expert System

Two of the cycles with rules which assign conflicting expressions to a LHS variable in the *enable-rule* (**ER**) graph corresponding to the ISA Expert System identified by the general analyzer are: (10→18→34→10) and (10→18→35→10). An **ER** graph shows the potentially enable relationships among rules in a program. These **ER** cycles indicate that the program may not reach a fixed point given a particular initial state. The rules involved in these two cycles are reproduced below.

```
(* 10 *)
[] state3 := failed IF find_bad_things = true AND state3 =
     suspect AND NOT (rel1_state = suspect AND rel1_mode = on
     AND rel1_type = direct) AND NOT (rel2_state = suspect AND
     rel2_mode = on AND rel2_type = direct)
(* 18 *)
[] state3 := nominal ! reconfig3 := true IF state3 = failed AND
     mode3 <> off AND config3 = bad
(* 34 *)
[] sensor3 := bad ! state3 := suspect IF state1 = suspect AND
     rel1_mode = on AND rel1_type = direct AND state3 = nominal
     AND rel3_mode = on AND rel3_type = direct AND state4 =
     suspect AND find_bad_things = true
(* 35 *)
[] sensor3 := bad ! state3 := suspect IF state2 = suspect AND
     rel2_mode = on AND rel2_type = direct AND state3 = nominal
     AND rel3_mode = on AND rel3_type = direct AND state4 =
     suspect AND find_bad_things = true
```

Now suppose the programmer knows that these rules will never fire infinitely because of certain application-specific scheduling policy. The following **Estella** statement, which specifies the general break condition for cycles in the **ER** graph in special form **B** needs to be modified for the four rules above.

```
BREAK_CYCLE(ENABLE_RULE) =
    (NOT(EXIST i,j
            ( ( NOT(EQUAL(i,j)) AND
                ( IN_CYCLE(EDGE(ENABLE_RULE,i,j)) AND
                  EXIST i.p, j.q
                    ( (EQUAL(LEXP[i.p],LEXP[j.q]) AND
                       NOT(EQUAL(REXP[i.p],REXP[j.q]))) ) )
              )
            )
        )
    )
```

A thorough treatment of the syntax of **Estella** shall be given in section 7.2. For this example, it suffices to know that the variables LEXP[i.p] and REXP[i.p] contain respectively the LHS variable appearing in subrule p of rule i and the expression appearing in RHS of subrule p of rule i. Similar definitions apply to LEXP[j.q] and REXP[[j.q]. To indicate that the above four rules will not fire infinitely often, the following **Estella** statement is used to specify that the break condition for the cycles (10,18,34) and (10,18,35) in the **ER** graph of the program is `TRUE`, and thus these two **ER** cycles will not cause infinite rule firings. The general analyzer would then ignore these two cycles when checking to see whether the set of rules containing these four rules satisfy the behavioral constraint assertions of a special form.

```
BREAK_CYCLE(ENABLE_RULE, {(10,18,34),(10,18,35)}) = TRUE
```

We shall show later in more detail how this specification is actually used by the analysis tool to produce the desired results. A complete treatment of the syntax of **Estella** appears in the following section.

7.2. Facility for Specifying Behavioral Constraint Assertions

The facility for specifying behavioral constraint assertions on **EQL** rules and analyzing **EQL** programs consists of the following major components:

(1) the BCA recognition procedure generator,

(2) the **EQL** program information extractor, and

(3) the general analyzer.

These three components are depicted in Figure 7.2. The BCA recognition procedure generator serves as an interpreter to the BCA specification written in **Estella** and generates the corresponding procedure for recognizing the specified BCAs of rules with bounded response times.

The **EQL** program information extractor is a collection of procedures that extract relevant information from the **EQL** program to be analyzed. These procedures provide information in the form of objects that can be used in the **Estella** specification. For example, the **ER** graph constructor builds the **ER** graph from the **EQL** program and provides objects such as **ER** cycles or **ER** edges which can be named by **Estella** primitives. The information extractor is designed to be expandable so that it can easily accommodate new procedures for extracting new information as the need arises and for providing new objects previously not available.

The general analyzer takes as input the information provided by the information extractor, and the recognition procedures generated by the BCA recognition procedure generator and stores the procedures in its knowledge bases. The general analysis algorithm can then be extended to accommodate this extra level of user-directed analysis assistance.

Estella is primarily designed for specifying behavioral constraint assertions about rules written in the **EQL** language. It is expressive enough to allow for the specification of a wide range of BCAs about **EQL** rules; and it does not require the rule-based programmer to possess knowledge of the implementation details of the **Estella** interpreter or the recognition procedures.

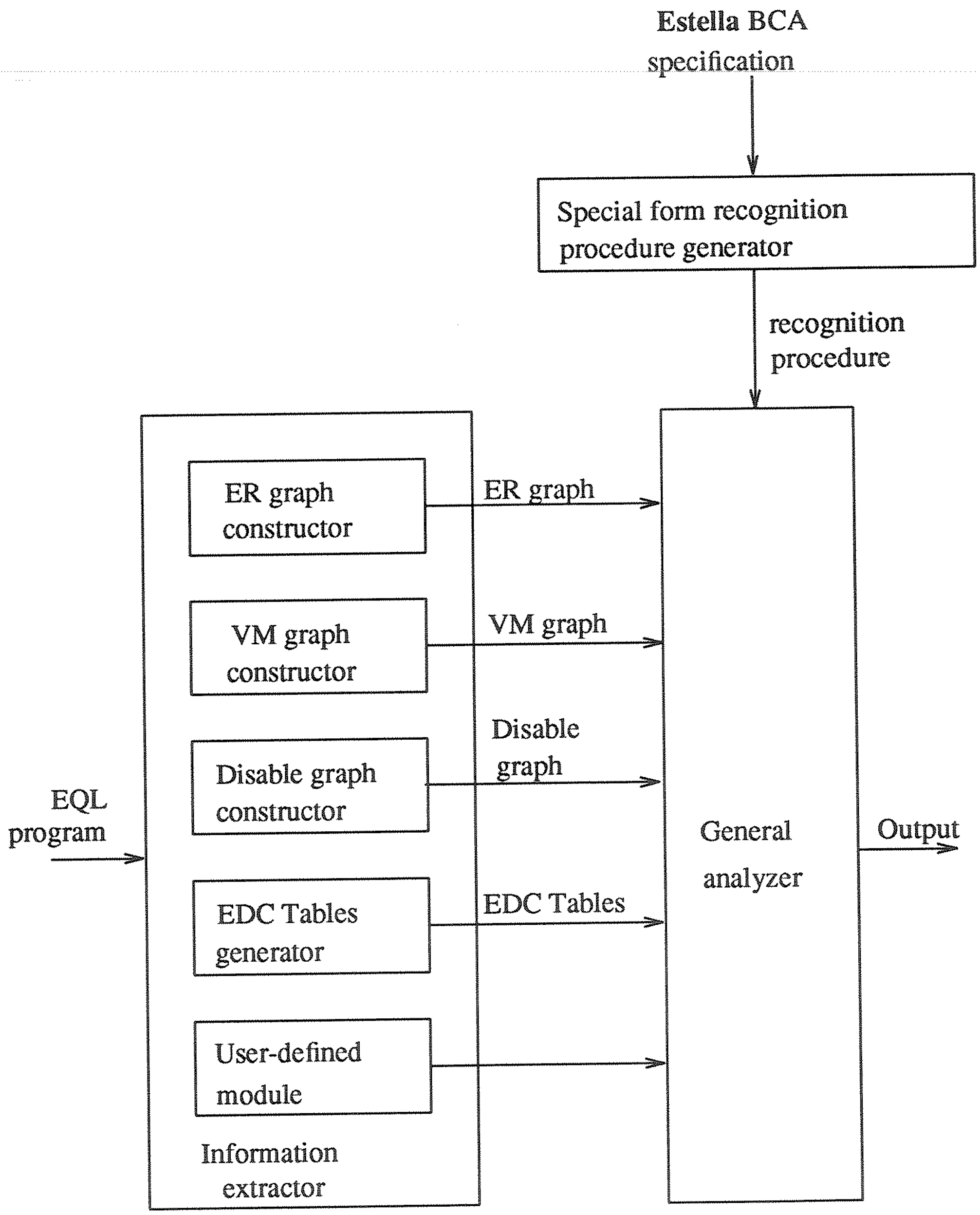


Figure 7.2. The General Analysis Tool - Estella Specification Facility.

7.2.1. Syntax and Semantics of Estella

Estella allows the specification of constraints on the syntactic and semantic structure of **EQL** programs. It provides a set of constructs that are designed to simplify the specification of conditions relevant to rule-based programs. A set of rules satisfying all BCAs of a special form specified by the programmer is guaranteed to have bounded response time. It is the responsibility of the programmer to ensure that the specified special form is correct since the analysis facility does not verify the correctness of user-defined special forms. The programmer may specify different special forms of rules with bounded response times as input to the analyzer.

There are two types of BCAs, *predefined general* BCAs and *user-defined* BCAs, both of which are specified in essentially the same way with **Estella**. The programmer has the option of interactively entering a BCA into the general analyzer or loading a previously specified special form stored in a library file into the general analyzer. The programmer can also specify the order in which the special forms should be checked against during each step of the analysis so that the analysis process can be optimized.

An important **Estella** statement deserves special mention:

> The definition of the break condition *break_cycle* for breaking cycles consisting of specific edges in a specific graph.

This statement constitutes a major special form condition since each of the general special forms of rules developed so far has at least one condition that stipulates

> (1) the absence of certain type(s) of cycles in some kind of graph representing information about the rules, or
>
> (2) if there exist cycles in a specific graph representing information about the rules, then there must be *break_cycle* condition which is

guaranteed to break this kind of cycles so that no cycles can appear in the state-space graph corresponding to the rules. Cycles in the state-space graph corresponding to a program indicate that the program may not have bounded response time because there may be an unbounded number of rule firings.

Note that the *break* condition for a particular type of cycles or for a specific cycle may specify `TRUE` (break condition is not needed) if the cycle(s) in question do not cause a cycle in the state-space graph corresponding to the program.

We first provide an informal description of **Estella** in order to highlight its features. A formal specification of **Estella** in the YACC grammar shall be given in the Appendix B.

Estella provides the following predefined constants:

{ } = empty set.

Estella provides the following predefined set variables:

L = { v | v is a variable appearing in LHS }

R = { v | v is a variable appearing in RHS }

T = { v | v is a variable appearing in EC }

L[i] = { v | v is a variable appearing in LHS of rule i }

L[i.j] = { v | v is a variable appearing in LHS of rule i, subrule j }

R[i] = { v | v is a variable appearing in RHS of rule i }

R[i.j] = { v | v is a variable appearing in RHS of rule i, subrule j }

T[i] = { v | v is a variable appearing in EC of rule i }

Estella provides the following predefined expression variables which are used to refer to the actual expression or the **EQL** variable in the location indicated by the indices of the variables:

LEXP[i.j] = LHS variable of rule i, subrule j.

REXP[i.j] = RHS expression of rule i, subrule j.

TEXP[i] = EC of rule i.

Estella provides the following predefined functions:

INTERSECT(A,B): intersection of set A and set B, where A and B are set variables or elements enclosed by '{' and '}'.

UNION(A,B): union of set A and set B, where A and B are set variables or elements enclosed by '{' and '}'.

RELATIVE_COMPLEMENT(A,B): relative complement = set A – set B, where A and B are set variables or elements enclosed by '{' and '}'.

Estella provides the following predefined predicates:

MEMBER(a,B): a is a member of set B, where a is an **EQL** variable, and B is a set variable or elements enclosed by '{' and '}'.

IN_CYCLE(EDGE(edge_type, a, b)): edge (a,b) of type edge_type is in the cycle found.

EQUAL(a,b): a is equal to b, where a and b can be set variables, expression variables, or values.

COMPATIBLE(a,b): rule a and rule b are compatible (defined in section 2.2).

MUTEX(a,b): test of rule a and test of rule b are mutually exclusive.

COMPATIBLE_SET = (compatible_sets)

This predicate specifies set of compatible sets of rules. All pairs of rules in each of these compatible sets are considered compatible by the rule-based programmer even though they do not satisfy the predefined requirements of compatibility. This predicate allows the programmer to relax the compatibility condition for some pairs of rules when he/she knows that this would not cause the rules to fire infinitely often without reaching a fixed point.

BREAK_CYCLE(graph_type, cycles_list) = break_condition,

where *graph_type* is the type of graph, *cycles_list* is an optional argument listing specific cycles in the graph of *graph_type*, and *break_condition* is a condition for breaking a cycle in the graph of *graph_type*. This predicate specifies a condition for breaking all cycles or specific cycles (when the second argument *cycles_list* is specified). Thus programs with cycles (as specified in conditions of the general special forms of rules) which satisfy the *break_condition* so specified will not cause unbounded rule firings.

Estella provides the following primitive objects:

VERTEX: vertex in a graph(**ER, VM** or **disable**).

EDGE(ENABLE_RULE, a, b): edge from vertex a to vertex b in the enable rule graph.

EDGE(DISABLE, a, b): edge (a,b) in the disable graph.

EDGE(VARIABLE_MODIFICATION, a, b): edge (a,b) in the variable modification graph.

CYCLE(ENABLE_RULE): a cycle in the enable rule graph.

CYCLE(VARIABLE_MODIFICATION): a cycle in the variable modification graph.

Estella provides the following primitive constructs:

specification:

a well-formed formula as defined below.

Definition 7.1.

Terms are defined recursively as follows.

(1) A constant is a term.
(2) A variable is a term.
(3) If f is a function symbol with n parameters, and $x_1, \ldots, t_n$ are terms, then $f(x_1, \ldots, x_n)$ is a term.
(4) All terms are generated by applying rules (1), (2), and (3).

Definition 7.2.

If p is a predicate symbol with n parameters, and $x_1, \ldots, t_n$ are terms, then $p(x_1, \ldots, x_n)$ is an atomic formula.

Definition 7.3.

A well-formed specification is defined recursively as follows.

(1) An atomic formula is a specification.
(2) If F and G are specifications, then not(F), (F OR G), (F AND G), ($F \rightarrow G$), and ($F \leftrightarrow G$) are specifications.
(3) If F is a specification and *x_list* is a list of free variables in F, then FORALL *x_list* (F) and EXIST *x_list* (F) are specifications.
(4) Specifications are generated only by a finite number of applications of rule (1), rule (2), and rule (3).

$\{P\}\ R\ \{Q\}$

Assertion: P and Q are predicates on the variables of an **EQL** program, and R is a sequence of one or more rules. This construct states that if the program is at a state in which the predicate P is true, then following the execution of the rules in R, the program reaches a state in which the predicate Q is true. Note that an assertion in **Estella**, like other constructs, is a constraint specified by the programmer, and this condition must be satisfied by a program in order to guarantee that the program has bounded response time. The analyzer does not attempt to prove the correctness of the specified assertion. It simply performs the following: given the initial state in which the predicate P is satisfied, execute the rules in R (note that it is possible that none of the rules are enabled and thus no rule is fired), determine whether the program reaches a state in which the predicate Q is satisfied. Since the current version of the analyzer does not make use of the assertion construct, this construct will be implemented in a future version of the **Estella** specification facility.

7.2.2. Specifying Special Forms of Rules with Estella

In this section, we show how **Estella** is used to define general BCAs (called special forms of rules) for ensuring that **EQL** programs satisfying these BCAs will always have bounded response time.

Definition of the predicate compatible in English and mathematical notations:

Let L_x denote the set of variables appearing in LHS of rule x. Two rules a and b are said to be **compatible** iff at least one of the following conditions holds:

(CR1) Test a and test b are mutually exclusive,

(CR2) $L_a \cap L_b = \emptyset$.

(CR3) Suppose $L_a \cap L_b \neq \emptyset$. Then for every variable v in $L_a \cap L_b$, the same expression must be assigned to v in both rule a and b.

Estella Specification of `COMPATIBLE(a,b):`

```
   MUTEX(a,b)
or EQUAL(INTERSECT(L[a],L[b]), { })
or FORALL v (FORALL a.p, b.q ((((MEMBER(v,intersect(L[a],L[b]))
     AND EQUAL(v,LEXP[a.p]))
     AND EQUAL(v,LEXP[b.q]))
     -> EQUAL(REXP[a.p],REXP[b.q])))
```

`compatible` is a predefined predicate in **Estella**. It should be noted that compatibility condition can be considered as a break condition since it is used to break a possibly cyclic sequence of rule firings.

Definition of special form A in English and mathematical notations:

(A1) Constant terms are assigned to all the variables in L.

(A2) All of the rules are compatible pairwise.

(A3) $L \cap T = \emptyset$.

Estella Special Form Specification of Special Form A:

```
SPECIAL_FORM a:
    EQUAL(R,{ });
    FORALL i,j (COMPATIBLE(i,j));
    EQUAL(INTERSECT(L,T), { })
END.
```

Definition of special form B in English and mathematical notations:

A set of rules are said to be in special form **B** if all of the following conditions hold.

(B1) Constant terms are assigned to all the variables in L, i.e., $R = \varnothing$.
(B2) All of the rules are compatible pairwise.
(B3) $L \cap T \neq \varnothing$.
(B4) For each cycle in the **ER** graph corresponding to this set of rules,
no two rules in the cycle assign different expressions to the same variable.
(B5) Rules in disjoint simple cycles (with at least two vertices) in the **ER** graph do not assign different expressions to a common variable appearing in their LHS.

Estella Special Form Specification of Conditions B1-B4:

```
SPECIAL_FORM b:
    EQUAL(R,{ });
    FORALL i,j (COMPATIBLE(i,j));
    NOT(EQUAL(INTERSECT(L,T),{ }));
    BREAK_CYCLE(ENABLE_RULE) =
         (NOT(EXIST i,j
                  ( ( NOT(EQUAL(i,j)) AND
                      ( IN_CYCLE(EDGE(ENABLE_RULE,i,j)) AND
                        EXIST i.p, j.q
```

```
                          ( (EQUAL(LEXP[i.p],LEXP[j.q]) AND
                            NOT(EQUAL(REXP[i.p],REXP[j.q]))) ) )
                   )
                  )
              )
          )
END.
```

Definition of special form C in English and mathematical notations:

A set of rules are said to be in special form C if all of the following four conditions hold.

(C1) Variable terms are assigned to the variables in L, i.e., $R \neq \varnothing$.
(C2) All of the rules are pairwise compatible.
(C3) $L \cap T = \varnothing$.
(C4) For each cycle in the variable-modification graph corresponding to this set of rules, there is at least a pair of rules (subrules) in the cycle that are compatible by condition CR1 (mutual exclusivity).

Estella Special Form Specification of Special Form C:

```
SPECIAL_FORM c:
    NOT(EQUAL(R,{ }));
    FORALL i,j (COMPATIBLE(i,j));
    EQUAL(INTERSECT(L,T), { });
    BREAK_CYCLE(VARIABLE_MODIFICATION) =
        (EXIST i, j (EXIST i.k, j.l
            ((IN_CYCLE(EDGE(VARIABLE_MODIFICATION,i.k,j.l)) AND
                MUTEX(i,j))))
            )
END.
```

Definition of special form D in English and mathematical notations:

A set of rules are said to be in special form **D** if all of the following conditions hold.

(D1) Variable terms are assigned to the variables in L, i.e., $R \neq \varnothing$.
(D2) All of the rules are compatible pairwise.
(D3) $L \cap T \neq \varnothing$.
(D4) For each cycle consisting of **LR** edges only in the variable-modification graph corresponding to this set of rules, there is at least a pair of rules (subrules) in the cycle that are compatible by condition CR1 (mutual exclusivity), or there is at least one **disable** edge in the cycle. (If a **disable** edge connects one *rule* vertex a to another *rule* vertex b, then there is one **disable** edge connecting every *subrule* vertex of rule a to every *subrule* vertex of rule b.)
(D5) For each cycle in the **ER** graph corresponding to this set of rules, no two rules in the cycle assign different expressions to the same variable.
(D6) Rules in disjoint simple cycles (with at least two vertices) in the **ER** graph do not assign different expressions to a common variable appearing in their LHS.

Estella Special Form Specification of Conditions D1-D5:

```
SPECIAL_FORM d:
   NOT(EQUAL(R,{ }));
   FORALL i,j (COMPATIBLE(i,j));
   NOT(EQUAL(INTERSECT(L,T),{ }));
   BREAK_CYCLE(VARIABLE_MODIFICATION) =
       (EXIST i, j (EXIST i.k, j.l
           ((IN_CYCLE(EDGE(VARIABLE_MODIFICATION,i.k,j.l)) AND
            MUTEX(i,j))))
        OR
        EXIST a,b
           (IN_CYCLE(EDGE(DISABLE,a,b)))
       );
   BREAK_CYCLE(ENABLE_RULE) =
       (NOT(EXIST i,j
               ( ( NOT(EQUAL(i,j)) AND
                   ( IN_CYCLE(EDGE(ENABLE_RULE,i,j)) AND
                     EXIST i.p, j.q
                       ( (EQUAL(LEXP[i.p],LEXP[j.q]) AND
                          NOT(EQUAL(REXP[i.p],REXP[j.q]))) ) )
                 )
               )
           )
       )
END.
```

7.3. Applications of the General Analyzer-Estella Facility

To demonstrate the applicability of the **Estella** facility, we use it in conjunction with the *General Analysis Tool* to analyze two expert systems from Mitre and NASA.

7.3.1. Specifying Cycles and Break Conditions for Analyzing the ISA Expert System

The purpose of the Integrated Status Assessment Expert System (ISA) program is to determine the faulty components in a network. A component can be either an *entity* (node) or a *relationship* (link). A relationship is a directed edge connecting two entities. Components are in one of three states: nominal, suspect, or failed. A failed entity can be replaced by an available backup entity. This expert system makes use of simple strategies to trace failed components in a network. The ISA Expert System consists of: 35 **EQL** rules, 46 variables (29 of which are input variables), and 12 constants.

After reading in the ISA program, the analyzer checks the rules to determine if every rule pair satisfy the compatibility condition. Two pairs of rules have been identified as being not compatible: rule 8 and rule 32, rule 9 and rule 33. At this point, the rule-based programmer can take one of the following actions:

(1) revise the ISA program to make the above rule pairs compatible,

(2) employ a special form which does not require the above rules to be compatible to perform further analysis,

(3) specify the above rules as compatible by using the COMPATIBLE_SET predicate if he/she considers these rules to be compatible in his/her application domain.

Suppose the programmer selects the third action, then the command 'cs' (compatible set) can be used to specify compatible sets of rules.

```
command > cs

compatible set specification >
     COMPATIBLE_SET = ({8,32}, {9,33})
```

```
compatible sets entered
```

Now we load two predefined special forms (special form **A** and special form **B** stored in files sfA and sfB respectively) into the analyzer using the 'ls' (load special form) command:

```
command > ls

special form file name > sfA
special form file sfA entered

command > l

special form file name > sfB
special form file sfB entered
```

Special forms of rules can also be specified interactively using the 'sf' (new special form) command. During the first iteration of the general analysis, a bad cycle in the **ER** graph corresponding to the ISA Expert System has been identified: (10→18→34→10).

```
Step 1:
  9 strongly connected components in dependency graph.
  Bad cycle: 34->10->18

Independent special form subset is empty.

Analysis stops.
```

This indicates that the program may not reach a fixed point given a particular initial state. The rules involved in this **ER** cycle are presented in example 7.1.1.2 earlier.

Now suppose the programmer knows that these rules will never fire infinitely because of the use of a scheduler which prevents these rules to fire forever. The general break condition for cycles in the enable rule graph in special form **B** needs to be relaxed for these rules. We do not actually modify this general break condition, we make use of the exception command 'bc' (break cycle condition) to define an application-specific assertion. To indicate that the above three rules will not fire infinitely often, the following **Estella** statement is used to specify that the break condition for the cycle (10,18,34) in the enable rule graph of the program is `TRUE`, and thus this **ER** cycle will not cause unbounded rule firings. The general analyzer would then ignore this cycle when checking to see whether the set of rules containing these three rules are in a special form. We specify this break condition in the **Estella** facility as follows:

```
command > bc

with respect to special form > b

break condition specification >
     BREAK_CYCLE(ENABLE_RULE, {(10,18,34)}) = TRUE
```

7.3.2. Specifying Assertions for Analyzing the FCE Expert System

The purpose of the Fuel Cell Expert System (FCE) program is to determine the statuses of the different components of the fuel cell system based on current sensor readings and previous system state values. Then it displays the corresponding diagnostics according to the evaluation of the statuses of the different components of the system. This expert system contains: 101 EQL rules, 56 program variables, 130 input variables, and 78 constants.

The FCE program is organized into three main sections: (1) the meta rule section (testFCE1.eql), (2) 12 ordinary rule classes (testFCE2.*.eql), and (3) the output section (testFCE3.eql). After reading in the FCE program, the analyzer checks the rules to determine if every rule pair satisfy the compatibility condition. The following pairs of rules have been identified as being not compatible:

```
Incompatible rule pairs: (R1,R2) (R1,R3) (R1,R4) (R1,R5) (R1,R6) (R1,R7) (R1,R8)
 (R1,R9) (R1,R10) (R1,R11) (R1,R12) (R1,R13) (R2,R3) (R2,R4) (R2,R5) (R2,R6)
 (R2,R7) (R2,R8) (R2,R9) (R2,R10) (R2,R11) (R2,R12) (R2,R13) (R3,R4) (R3,R5)
 (R3,R6) (R3,R7) (R3,R8) (R3,R9) (R3,R10) (R3,R11) (R3,R12) (R3,R13) (R4,R5)
 (R4,R6) (R4,R7) (R4,R8) (R4,R9) (R4,R10) (R4,R11) (R4,R12) (R4,R13) (R5,R6)
 (R5,R7) (R5,R8) (R5,R9) (R5,R10) (R5,R11) (R5,R12) (R5,R13) (R6,R7) (R6,R8)
 (R6,R9) (R6,R10) (R6,R11) (R6,R12) (R6,R13) (R7,R8) (R7,R9) (R7,R10) (R7,R11)
 (R7,R12) (R7,R13) (R8,R9) (R8,R10) (R8,R11) (R8,R12) (R8,R13) (R9,R10) (R9,R11)
 (R9,R12) (R9,R13) (R10,R11) (R10,R12) (R10,R13) (R11,R12) (R11,R13) (R12,R13)
 (R18,R26) (R18,R47) (R18,R48) (R18,R49) (R18,R50) (R18,R51) (R30,R31) (R43,R44)
 (R43,R45) (R43,R46) (R44,R45) (R44,R46) (R45,R46) (R47,R49) (R48,R49) (R57,R63)
 (R63,R66) (R63,R67)
```

The meta rule section contains high-level control rules used to determine which ordinary rule classes should be enabled. The meta rules are shown in Appendix E. From the documentation accompanying the FCE Expert System, it is evident that in the meta rule section (testFCE1.eql), each meta rule fires at most once and thus a fixed point will be reached in bounded time even though the rules are not compatible. The following **Estella** statement captures this application-specific assertion:

```
        COMPATIBLE_SET = ({1,2,3,4,5,6,7,8,9,10,11,12,13})
```

This statement states that all 13 rules in the meta rule section are compatible pairwise even though they do not satisfy the predefined requirements of compatibility. We specify this compatibility condition in the **Estella** facility as follows:

```
command > cs

compatible set specification >
     COMPATIBLE_SET = ({1,2,3,4,5,6,7,8,9,10,11,12,13})
compatible sets entered
```

Now we load two predefined special forms (special form **A** and special form **B**) into the analyzer using the 'ls' (load special form) command as explained earlier. We then analyze the rules in the meta rule section:

```
command > an

Select the rules to be analyzed:
  1. the whole program
  2. a continuous segment of program
  3. separated rules
Enter your choice: 2

Enter the first rule number:1
Enter the last rule number:13

Step 1:
  S.C.C.: R13 R12 R11 R10 R9 R8 R7 R6 R5 R4 R3 R2 R1
  1 strongly connected components in dependency graph.
  independent subset: 1 2 3 4 5 6 7 8 9 10 11 12 13
  13 rules in special form a.

0 rules remaining to be analyzed:

Textual analysis is completed.
The program always reaches a fixed point in bounded time.
```

After revising the FCE program to make the incompatible rules compatible, the analyzer reports that the entire program is always guaranteed to reach a fixed point in bounded time.

7.4. Implementation of the General Analysis Tool and Estella

In this section, we describe efficient algorithms for implementing the general analyzer and the **Estella** system.

7.4.1. General Analysis Algorithm

The general analysis tool allows the rule-based programmer to select a subset of the rules in a program for analysis. The subset may contain either a contiguous list of rules or separated rules. This provision reduces analysis time by directing the analyzer to focus the checking on ‘trouble spots’ which the rule-based programmer considers as possible sources of timing violations. In the following section, we shall show how the analyzer uses decomposition techniques to break the program into independent sets so that it does not have to analyze the entire program at once.

Algorithm 7.4.1. General Analyzer.

Input: A complete **EQL** program or a set of **EQL** rules; a list of special forms and exceptions, if any, specified in **Estella**.

Output: If the program will always reach a fixed point in bounded time, output 'yes'. If the program may not always reach a fixed point in bounded time according to the analysis, output 'no' and the rules involved in the possible timing violations.

(1) Parse the special form specifications and exceptions; then generate the corresponding BCA recognition procedures.

(2) Construct the high-level dependency graph corresponding to the program by using algorithm 7.4.2.1 (section 7.4.2.1).

(3) If there are more rules for analysis, identify forward-independent sets of rules which are in special forms using algorithm 7.4.2.2 (section 7.4.2.2). If at least one rule set in special form is found and there are more rules to be analyzed, go to step 5. If there are no more rules for analysis, output 'yes' (the **EQL** rules have bounded response time).

(4a) If no independent set of rules is in a special form catalogued but the variables in some rule set have finite domains, check whether this rule set can always reach a fixed point in bounded time by using a state-based model checking algorithm ([Clarke, Emerson & Sistla 86]). If the rule set is determined to be able to always reach a fixed point in bounded time, go to step 5. If the rule set is determined to be unable to always reach a fixed point in bounded time, report the rules involved in the timing violations; go to step (4b).

(4b) Prompt the user for new special forms or exceptions. If new special forms or exceptions are entered, go to step 3. If no new special forms or exceptions are entered, stop; output 'no' (the **EQL** rules do not have bounded response time).

(5) Mark those forward-independent sets identified to be in special forms as checked (which effectively removes those rules from further analysis). Rewrite remaining rules to take advantage of the fact that some variables can be treated as constants because of the special form; go to step (3).

7.4.2. Selecting Independent Special Form Set

To determine whether a set of rules is independent from another set of rules, the selection algorithm makes use of the following theorem.

Theorem. (Sufficient conditions for independence)

Let S and Q be two disjoint sets of rules. S is **independent** from Q if the following conditions hold:

(I1) $L_S \cap L_Q = \varnothing$.
(I2) the rules in Q do not potentially enable rules in S.
(I3) $R_S \cap L_Q = \varnothing$.

Proof.

(1) Condition I1 guarantees that the firing of any rule in the set Q will not change the content of any variable in L_S which has already settled down to stable value. This is so because the set of variables L_S and the set of variables L_Q are disjoint.
(2) Condition I2 guarantees that the firing of any rule in the set Q will not

enable any rule in the set S.

(3) Condition I3 guarantees that the firing of any rule in the set Q will not change the value of any expression containing variable(s) that are assigned to variables in the set L_S. Condition 1 guarantees that once the rules in S have reached a fixed point, the contents of the variables in L_S will not change. At this point, a rule in S would not fire again unless at least one expression in R of that rule has changed its value since the last firing of that rule. However, $R_S \cap L_Q = \varnothing$, so the values of the expression in R_S will not change despite the presence of Q.

$\square$

To determine whether rule a potentially enables rule b, the implementation makes use of the approximately enable checking function. This function returns true if rule a potentially enables rule b or if there is insufficient information (some expressions in the test part cannot be evaluated unless the whole state space of the program is checked). It returns false otherwise.

7.4.2.1. Constructing and Checking the Dependency Graph

Before checking to determine if a subset of the rules is in a special form, the analysis algorithm first constructs a high-level dependency graph based on the above conditions for establishing independence of one set of rules from another set.

Algorithm 7.4.2.1. High-level dependency graph construction.

Input: An **EQL** program.

Output: A high-level dependency graph corresponding to the input **EQL** program.

(1) For each rule i in the **EQL** program, create a vertex labeled i.

(2) Let S contain rule i and let Q contain rule j. If one of the conditions I1, I2 or I3 is not satisfied, create a directed edge from vertex i to vertex j.

(3) Find every strongly connected component in the dependency graph $G(V, E)$ constructed by step 1 and step 2.

(4) Let $C_1, C_2, ..., C_m$ be the strongly connected components of this graph $G(V, E)$. Define $\overline{G}(\overline{V}, \overline{E})$ as follows:

$$\overline{V} = \{C_1, C_2, ..., C_m\}$$
$$\overline{E} = \{ (C_i, C_j) \mid i \neq j, (x, y) \in E, x \in C_i \text{ and } y \in C_j \}$$

We call $\overline{G}$ the *high-level dependency graph* of the input **EQL** program. Each of the vertices C_i in this high-level dependency graph is called a *forward-independent set.*

Theorem. The *high-level dependency graph* of any **EQL** program is a directed acyclic graph.

Proof.

Assume that $\overline{G}$ is not an acyclic graph. Then $\overline{G}$ has a directed circuit. However, all strongly connected components on it should have been one strongly connected component.

□

Timing and Space Requirements.

All graphs used in the algorithms presented in this paper are represented as adjacency lists. Let n be the number of vertices (or rules) and let e be the number of edges in G as constructed by step 1 and step 2. Step 1 can be performed in O(n)-time. Step 2 can be done in O(n^2)-time since every pair of rules must be checked. The checking of each of the three conditions can be done efficiently. Step 3 can be achieved in O($MAX(n,e)$)-time using the depth-first search strongly connected components algorithm of Tarjan ([Tarjan 72]). The creation of edges for $\overline{G}$ in step 4 can be done in O(e)-time since in the worst case, all edges in G may have to be examined. In the next section, we shall describe an algorithm for identifying special form sets in the high-level dependency graph.

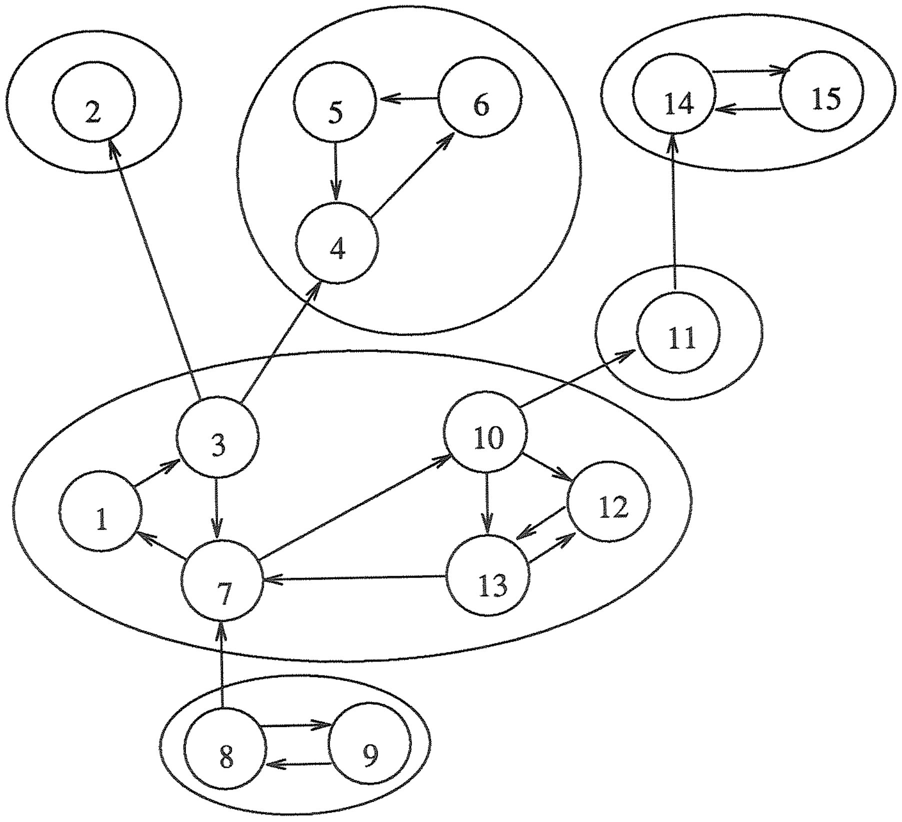


Figure 7.3. A High-level Dependency Graph and its Strongly Connected

Components.

7.4.2.2. Identifying Special Form Sets

The brute-force approach to identify sets of rules in a special form would be to generate all combinations of the rules in a program and then check the rules in each combination to see if they are in one of the special form catalogued. However, this approach does not take into account the syntactic and semantic structure of an **EQL** program and it has exponential time complexity. We shall present an algorithm for identifying special form sets by checking the high-level dependency graph constructed by algorithm 7.4.2.1.

Algorithm 7.4.2.2. Identification of Special Form Set.

Input: A high-level dependency graph $\overline{G}$ corresponding to an **EQL** program.

Output: Sets of rules, if any, identified to be in some predefined special forms or in some user-defined special forms.

(1) Sort the vertices in the high-level dependency graph $\overline{G}$ to obtain a reverse topological ordering of the vertices. Starting with the first vertex with no out-edge, label the vertices I_i, $i = 1, 2, ..., m$, where m is the total number of vertices in $\overline{G}$. This can be achieved by using a recursive depth-first search algorithm which labels the vertex just visited as describe above just before exiting from each call.

(2) For the set of rules contained in each vertex I_i such that I_i does not have any outgoing edge, determine if the rule set is in one of the special forms catalogued by using the algorithms which will be presented in section 7.4.3 and section 7.4.4. Report rules which are involved in the violation

of specific special form conditions.

Timing Requirements.

Step 1 can clearly be achieved in O($MAX(n,e)$)-time using a standard recursive depth-first search algorithm ([Aho, Hopcroft & Ullman 74]). The timing and space requirements for step 2 depends on the order in which the special forms are checked as well as on the complexity of the recognition algorithms used for each individual special forms.

Theorem.

The rules in a forward-independent set, I_i, are always guaranteed to reach a fixed point in bounded time if

(a) the rules in I_i by themselves can always reach a fixed point in bounded time and

(b) for every forward-independent set I_j such that there is an edge (I_i, I_j) in $\overline{G}$, the rules in I_j are always guaranteed to reach a fixed point in bounded time.

Proof.

There are two cases to consider:

(1) If I_i does not have any out-edge, then I_i is independent of any I_j, $i \neq j$. I_i is certainly guaranteed to always reach a fixed point in bounded time if the rules in it are always guaranteed to reach a fixed point in bounded time.

(2) If I_i has outgoing edges, then I_i is not independent from other vertices. Let I_j, $j = 1, 2, ..., p$ be these vertices such that for each I_j, there is an edge (I_i, I_j) in $\overline{G}$. Assume that rules in I_i may not reach a fixed point

in bounded time. Then there must be at least one variable v_1 in L of I_i whose value is changed infinitely often. However, rules in I_i are assumed to be able to always reach a fixed point in bounded time. Thus there must be another vertex I_k such that I_i is not independent from I_k and rules in I_k cannot reach a fixed point in bounded time. However, this violates our assumption that every I_j is always guaranteed to reach a fixed point in bounded time. Hence, rules in I_i are always guaranteed to reach a fixed point if both conditions (a) and (b) are satisfied.

$\square$

Theorem.

Let $I_j, j = 1, 2, ..., p$, be a list of mutually independent sets of rules. Suppose I_i is not independent from $I_j, j = 1, 2, ..., p$. If I_i is always guaranteed to reach a fixed point in bounded time, and each of the I_js is always guaranteed to reach a fixed point in bounded time, then the rules in $I_i \cup I_1 \cup I_2 \cup \cdots \cup I_p$ are always guaranteed to reach a fixed point in bounded time.

Proof.

Consider two different rule sets, I_a and I_b, taken from the list of I_js. Since I_a and I_b are mutually independent, the firing of rules in I_a does not enable rules in I_b. Thus the rules in I_b will reach a fixed point in bounded time despite the presence of I_a. The same argument applies to I_a. Extending this reasoning to more than two sets in the list of I_js, we can conclude that rules in $I_1 \cup \cdots \cup I_p$ are guaranteed to reach a fixed point in bounded time.

Let $K = I_1 \cup \cdots \cup I_p$. Since I_i is not independent from each set in the list of I_js, it is not independent of rules in K. However, rules in K are independent from rules in I_i. Therefore, the firing of rules in I_i does not enable rules in K. Consequently, rules in K will reach a fixed point in bounded time despite the presence of I_i. After the rules in K have reach a fixed point, the variables in K will no longer change their contents and thus these variables will not enable or disable rules in I_i. At this point, only rules in I_i may fire. Since I_i is guaranteed to reach a fixed point in bounded time, it can be concluded that the rules in $I_i \cup I_1 \cup \cdots \cup I_p$ are also always guaranteed to reach a fixed point in bounded time.

□

7.4.3. Checking Compatibility Conditions

Checking whether the second condition or the third condition for compatibility are satisfied by a pair of rules is a straightforward exercise in comparing sets and expressions. We shall therefore discuss the checking of the first condition: mutual exclusion.

The boolean function $mutex(e_1, e_2)$ checks whether two boolean expressions, e_1 and e_2, are mutually exclusive. The goal of *mutex* is not to conquer the general mutual exclusion problem. Instead, it tries to efficiently detect as much mutual exclusion as possible. When it returns true, it means that the two expressions are mutually exclusive. However, when it returns false, either it means that they are not mutually exclusive, or it means that there is insufficient information to determine the mutual exclusion. Therefore, the two expressions may still be mutual exclusive even when *mutex* returns false.

mutex makes use of a divide-and-conquer strategy inspired by the following observations. Let e_1, e_2, e_3 be arbitrary boolean expressions. For (e_1 AND e_2) and e_3 to be mutually exclusive, either e_1 or e_2 (or both) must be mutually exclusive with e_3. For (e_1 OR e_2) and e_3 to be mutually exclusive, both e_1 and e_2 must be mutually exclusive with e_3. Therefore we can continuously divide the problem into subproblems until both boolean expressions are simple expressions.

A subtle problem arises when NOT operators appear in expressions. For example, to decide whether (NOT e_1) and e_2 are mutually exclusive, we cannot simply invert the answer returned from *mutex*(e_1, e_2), otherwise errors may occur when *mutex*(e_1, e_2) returns false, meaning e_1 and e_2 may or may not be mutually exclusive. This problem can be solved by applying DeMorgan's laws to the expressions to reduce the occurrence of NOT operators to the lowest level. In the implementation, we do not explicitly rewrite the expressions before calling *mutex*. Instead, one flag is kept for each expression in *mutex* to keep track of the net effect of NOT operators down to this level. If a NOT operator is encountered at this level, the flag is inverted. This flag is then passed down to the subroutines in charge of the subexpressions.

The detection of mutual exclusion between two simple expressions is handled by function *smutex*(). A simple expression can be

(1) a boolean variable, e.g., sensor_a_good;

(2) a constant, e.g., FALSE; or

(3) a relational test, whose operator can be one of =, <>, >, <, >=, <=; and whose operands can be either variables or arithmetic expressions.

Although tedious, *smutex* handles the examination case by case. It also invokes function *eval*() which, given an expression, determines whether it can be evaluated or not. If it can be evaluated, *eval*() also returns the result of evaluation.

It should be noted that compatibility is independent of any special form. When the analyzer determines that rule *a* and rule *b* are compatible or when the user specifies this fact, then these two rules are considered compatible under any context (when checked by any special form or BCA recognition procedures).

Timing Requirements.

The complexity of *eval* () is linearly proportional to the length, in terms of the number of operators, of the expression to be evaluated. Although the general problem of mutual exclusion detection is of exponential complexity, *mutex* has been implemented to give answers in quadratic time.

7.4.4. Checking Cycle-Breaking Conditions

Cycle-breaking conditions are kept in a list, with the latest specified break condition placed at the head and the earliest specified break condition, which is the one in the special form part, placed at the tail. Later specified break conditions are checked first, and therefore are treated as exceptions. Since conditions in a special form may specify different types of graphs, there is one list for each type of graphs. We define any cycle found in a graph representing some information about a program to be **acceptable** if it does not cause a cycle in the state-space graph of the program. A cycle is **not acceptable** if it may cause a cycle in the state-space graph of the program. An arbitrary cycle-breaking condition is evaluated as follows:

Algorithm 7.4.4. Evaluating General Cycle-Breaking Conditions.

```
When a cycle is found in the graph of type graph_type:
   initially, assume the cycle is acceptable;
   if ( cycles_list is not specified or
       the cycle is found in the specified cycles_list )
      then if ( break_condition is evaluated to FALSE )
         then the cycle is not acceptable
         else the cycle is acceptable
   else if ( cycles_list is specified but the cycle is not found in it )
      then check the next break_condition in the list of break conditions for this
```

graph_type, if any

To allow for the checking of arbitrary break conditions for each cycle in an arbitrary graph representing syntactic and/or semantic structure of a program, the checking algorithm may be required to perform an exhaustive search on the arbitrary graph. The brute-force approach would have to determine every cycle in the graph and then check whether the break condition is satisfied for each cycle found. If $|E| \ll |V|^2$ where $|E|$ and $|V|$ are respectively the number of edges and the number of vertices in a graph (e.g., **ER** graph, **VM** graph) corresponding to a program, then this approach is still practical. The selection algorithm 7.4.2.1 described earlier decomposes the program into forward-independent modules which can be analyzed independently. This effectively breaks a large graph (corresponding to the entire program) into a set of smaller graphs (corresponding to independent modules of rules) each of which is more amenable to efficient analysis. It should be noted that an arbitrary graph corresponding to the rules in a forward-independent set may be different from the dependency graph (a strongly connected component) corresponding to these rules. Thus the arbitrary graph may be further decomposed into smaller components for analysis.

Furthermore, it should be noted that the types of graphs used for static analysis are much smaller in size than the state-space graph corresponding to a program. For instance, each vertex in an **ER** graph corresponds to a distinct rule in a program. If a program has n rules, then there are n vertices and at most $n(n-1)/2$ edges in the corresponding **ER** graph. In contrast, each of the vertices in a state-space graph represents a distinct vector of all variables in a program. For instance, for a finite-domain program with m variables, the total number of states in the corresponding state space graph in the worst case (i.e., all combinations of the values of the variables are possible) is $(\prod_{i=1}^{i=m} |X_i|$ where

$|X_i|$ is the size of the domain of the i^{th} variable. If all variables are binary, then this number is 2^m. Hence, our static analysis technique is a significant improvement over those based primarily on conventional finite-state-based checking.

To further improve the efficiency of checking cycle-breaking conditions, we have developed the following strategies that are applicable to some classes of break conditions and cycles. As in most strategies for attacking NP-complete graph problems, our strategies take advantage of special characteristics of a graph type and of the particular form of break conditions. Any attempt to find checking algorithms which are efficient for arbitrary graphs and arbitrary break conditions would not seem promising.

(1) For a class of break conditions which express relationships between two rules in a cycle, we can first determine those pairs of rules which violate the break condition being checked. Then for each pair of these rules, we check to see whether the pair of vertices corresponding to this pair of rules lie in a cycle. If the answer is yes, we can immediately conclude that the cycle detected is not broken by the break condition.

(2) For certain class of graphs and a class of break conditions, it is possible to first prune the acceptable cycles in the graph without actually finding all vertices in a cycle, and then apply a general cycle-checking algorithm. This would greatly reduce the number of cycles that have to be checked.

7.5. Concluding Remarks

We have described a specification facility called **Estella** for specifying behavioral constraint assertions about real-time **EQL** programs. This facility allows customization of the analysis tools so that they can be applied to the analysis of application-specific rule-based systems. In particular, the **Estella** facility allows the programmer to specify information about the rule-based program that is too hard to be detected by the general analyzer. The development of this facility as a computer-aided software engineering (CASE) tool is a significant step for aiding in the rapid prototyping and development of expert systems with guaranteed response time.

Chapter 8

The Synthesis Problem

The following definition is needed to formalize the synthesis problem. We say that an equational rule-based program P_2 is an **extension** of a program P_1 iff the following conditions hold. (1) The variables of P_1 are a subset of those of P_2. (2) P_1 has the same launch states as the projection of the state space of P_2 onto that of P_1, i.e., if P_2 has more variables than P_1, then we consider only those variables in P_2 that are also in P_1. (3) The launch states of P_1 and the corresponding ones in P_2 have the same end-points. Notice that we do not require the state space graph of P_1 to be the same as the corresponding graph of P_2, e.g., the paths from launch states to their end-points may be shorter in P_2. The synthesis problem is: Given an equational rule-based program P which always reaches a safe fixed point in finite time but is not fast enough to meet the timing constraints under a fair scheduler, determine whether there exists an extension of P that meets both the timing and integrity constraints under some scheduler.

For programs where all variables have finite domains, we can in principle compute all the end-points for each launch state, since the state space graph is finite. We can create a new program from the given program as follows. The new program has the same variables as the given program. Suppose s is a launch state and s' is an end-point of s. We create a rule r which is enabled iff the program is in s and firing r will result in the program being in s', i.e., the enabling condition of r is to match the the values of the variables in s and the multiple assignment statement of r assigns to the variables the corresponding values of those in s'. By this construction, the new program will always reach a fixed point in one iteration. Thus in theory, a solution always exists for the synthesis problem in the case of finite variables. This solution is very expensive in terms of memory since there are at least as many rules as there are launch states, even though some optimization can be performed to minimize the number of rules by exploiting techniques similar to those used in programmable logic array optimization. However, we still have to compute the end-points for each launch state and this can be computationally expensive.

We would like to find solutions to the synthesis problem without having to check the entire state space of a program. Two general approaches have been identified.

(1) Transforming the given equational rule-based program by adding, deleting, and/or modifying rules.

(2) Optimizing the scheduler to select the rules to fire such that a fixed point is always reached within the response time constraint. This assumes that there is at least one sufficiently short path from a launch state to every one of its end-points.

We shall illustrate both approaches with the program in example 3.1 which is reproduced below.

Example:

init: *object_detected* = false, *sensor_a_status*, *sensor_b_status* = good

input: read(*sensor_a*, *sensor_b*, *sensor_c*)

(* 1. *) *object_detected* := true IF *sensor_a* = 1 AND *sensor_a_status* = good
(* 2. *) [] *object_detected* := true IF *sensor_b* = 1 AND *sensor_b_status* = good
(* 3. *) [] *object_detected* := false IF *sensor_a* = 0 AND *sensor_a_status* = good
(* 4. *) [] *object_detected* := false IF *sensor_b* = 0 AND *sensor_b_status* = good

```
(* 5. *) [] sensor_a_status := bad IF sensor_a <> sensor_c AND sensor_b_status = good
(* 6. *) [] sensor_b_status := bad IF sensor_b <> sensor_c AND sensor_a_status = good
```

In this program, the variables *sensor_a_status* and *sensor_b_status* are initially set to *good*, and the variable *object_detected* is initially set to *false*. At the beginning of each invocation, the sensor values are read into the variables *sensor_a*, *sensor_b*, *sensor_c*. Note that if *sensor_a* and *sensor_b* read in values '1' and '0', respectively, then rule 1 and rule 4 may fire alternatively for an unbounded number of times before either rule 5 or rule 6 is fired. Similarly, if *sensor_a* and *sensor_b* read in values '0' and '1', respectively, then rule 2 and rule 3 may fire alternatively for an unbounded number of times before either rule 5 or rule 6 is fired. In this case, *sensor_c* can be used to arbitrate between the rules 1 and 4, or 2 and 3 by firing rule 5 or 6. (However, notice that only one of rule 5 or 6 may fire in each invocation; we do not want *sensor_c* to override both *sensor_a* and *sensor_b*.) This program will reach a fixed point in finite time since fairness guarantees that rule 5 or 6 will eventually be fired.

In approach (1), we ensure that this program will reach a fixed point in a bounded number of iterations starting from any launch state by performing the appropriate program transformation. First, we detect those rules that may constitute a cyclic firing sequence. In this program, the alternate firings of rule 1 and rule 4, or rule 2 and rule 3, may constitute a cyclic firing sequence. Noting that the firing of either rule 5 or rule 6 may disable two of rules 1-4, we add a rule (rule 7) and some additional conditions to enforce the firing of either rule 5 or rule 6 first if there is a conflict between the values read into *sensor_a* and *sensor_b*, thus breaking the cycle. The transformed program which is always guaranteed to reach a fixed point in a bounded number of iterations is shown below.

```
init: object_detected = false, sensor_a_status, sensor_b_status = good
```

```
invoke: $conflict$ := true
input: read($sensor\_a$ , $sensor\_b$ , $sensor\_c$ )

(* 1. *)    $object\_detected$ := true
                IF $sensor\_a = 1$ AND $sensor\_a\_status$ = good AND $conflict$ = false
(* 2. *) [] $object\_detected$ := true
                IF $sensor\_b = 1$ AND $sensor\_b\_status$ = good AND $conflict$ = false
(* 3. *) [] $object\_detected$ := false
                IF $sensor\_a = 0$ AND $sensor\_a\_status$ = good AND $conflict$ = false
(* 4. *) [] $object\_detected$ := false
                IF $sensor\_b = 0$ AND $sensor\_b\_status$ = good AND $conflict$ = false
(* 5. *) [] $sensor\_a\_status$ := bad
                IF $sensor\_a$ <> $sensor\_c$ AND $sensor\_b\_status$ = good
(* 6. *) [] $sensor\_b\_status$ := bad
                IF $sensor\_b$ <> $sensor\_c$ AND $sensor\_a\_status$ = good
(* 7. *) [] $conflict$ := false
                IF $sensor\_a = sensor\_b$ OR $sensor\_a\_status$ = bad OR $sensor\_b\_status$ = bad
```

In this program, the variable *conflict* is always set to *true* by the *invoke* command of **EQL** which is executed at the beginning of each invocation.

In approach (2), we customize an optimal scheduler which always selects the shortest path to any end-point from a launch state. For the program of example 3.1, this can be achieved by a fixed priority scheduler which assigns a higher priority to rules 5 and 6, i.e., if rule 5 or rule 6 is enabled, it is always fired before rules 1-4.

It should be emphasized that the two approaches for solving the synthesis problem are not in general polynomial time. Determining whether a scheduler exists which meets a response time constraint is NP-hard, as we shall show in the next section.

8.1. Time Complexity of Scheduling Rule-Based Programs

Consider the following equational rule-based program:

```
init: $R = 0, t_1 = t_2 = \cdots = t_n = 0$

input: read $(C)$

(* 1. *)   $R := R + q_1(\bar{t})$ ! $t_1 := t_1 + 1$ IF $R < C$
(* 2. *) [] $R := R + q_2(\bar{t})$ ! $t_2 := t_2 + 1$ IF $R < C$
            .
            .
            .
(* n. *) [] $R := R + q_n(\bar{t})$ ! $t_n := t_n + 1$ IF $R < C$
```

In the above program, $\bar{t}$ is the vector $<t_1, t_2, ..., t_n>$. We can think of the variable R which is initially 0, as the accumulated reward for firing the rules, and t_i as the number of times rule i has been fired, which is initially 0 for all n rules. The function $q_i(\bar{t})$ gives the additional reward that can be obtained by firing rule i one more time. All the $q_i(\bar{t})$s are monotonically non-decreasing functions of $\bar{t}$, and so the program may reach a fixed point in finite time assuming that some q_is return positive values.

The **time-budgeting problem** is to decide whether the above program can increase R from 0 to $\geq C$ in T iterations, for some given T. The time-budgeting problem arises when an output must be computed within a response time constraint of T by a real-time decision system which is composed of n subsystems. To compute the output, the decision system must invoke a number of subsystems S_i, $i = 1, \ldots, n$, each of which computes a partial output. The quality q_i of each partial output is dependent on the time t_i allocated to the subsystem S_i, and the overall quality of the output depends on some function of the quality of the partial outputs. Given a fixed time period T, the objective of the time-budgeting problem is to maximize the overall quality $R = q_1 + \cdots + q_n$ by determining an optimal partition of T into n time slices, where each time slice t_i, $i = 1, \cdots, n$ corresponds to the time allocated to subsystem S_i.

Referring to the **EQL** program above, it is obvious that the time-budgeting problem is in NP since a nondeterministic algorithm can guess the number of times each of the n rules should be fired, and check in polynomial time whether $t_1 + t_2 + \cdots + t_n \leq T$ and $R \geq C$. This time-budgeting problem can be shown to be NP-complete by an easy reduction from the NP-complete **knapsack problem**. The knapsack problem consists of a finite set U, a size $s(u)$, and a value $v(u)$ for each $u \in U$, a size constraint T, and a value objective C. All values $s(u)$, $v(u)$, T, and C are positive integers. The question is to determine whether there is a subset $U_1 \in U$ such that the sum of the sizes $s(u) \in U_1 \leq T$ and the sum of the values $v(u) \in U_1 \geq C$. To transform the knapsack problem into the time-budgeting problem, let each item $u_i \in U$ correspond to a unique rule i such that

$$q_i(\overline{t}) = \begin{cases} 0 & \text{if } t_i < s(u_i) \\ v(u_i) & \text{if } t_i \geq s(u_i) \end{cases}$$

Obviously, the knapsack problem has a solution iff it is possible to schedule a subset of the rules to fire a total of T times so that $R \geq C$.

The time-budgeting problem captures the property of an important class of real-time applications where the precision and/or certainty of a computational result can be traded for computation time. Solution methods to this problem are therefore of practical interest. For the case where the total reward is the sum of the value functions of the subsystems, the problem can be solved by a well-known pseudo-polynomial time algorithm based on the dynamic programming solution to the knapsack problem. Since this computation is done off-line, computation time is usually not critical here. However, if the total reward is a more complex function than the sum, then the dynamic programming approach may not apply. We shall propose another approach which is suboptimal but can handle complex total reward functions. The idea is to use a continuous function to interpolate and bound each reward function and then apply the method of Lagrange multipliers to maximize the total reward, subject to the given timing constraint. This approach is explored in the next section.

8.2. The Method of Lagrange Multipliers for Solving the Time-Budgeting Problem

Given that the reward for firing the i^{th} rule t_i times is $q_i(t_i)$, and T is the maximum number of iterations allowed, the time-budgeting problem can be formulated as a combinatorial optimization problem whose objective is to maximize R subject to the constraint: $t_1 + \cdots + t_n - T = 0$. For the above program, $R(\overline{t}) = q_1(t_1) + \cdots + q_n(t_n)$. Other than the requirement that the t_is must be integral, this problem is in a form that can be solved by the method of Lagrange multipliers. To maximize (or minimize) a reward function $f(\overline{t})$ subject to the side condition: $g(\overline{t}) = 0$ (i.e., response time constraint in our case), we solve for $\overline{t}$ in $\nabla H(\overline{t}, \lambda) = 0$ where λ is the Lagrange multiplier and

$$H(\overline{t}, \lambda) = f(\overline{t}) - \lambda * g(\overline{t}).$$

Example 8.2.1.

Consider the following **EQL** program which is an instance of the time-budgeting problem with two rules.

init: $R = 0, t_1 = t_2 = 0$

input: read(C)

(* 1. *) $R := R + q_1(\overline{t}) \,!\, t_1 := t_1 + 1$ IF $R < C$
(* 2. *) [] $R := R + q_2(\overline{t}) \,!\, t_2 := t_2 + 1$ IF $R < C$

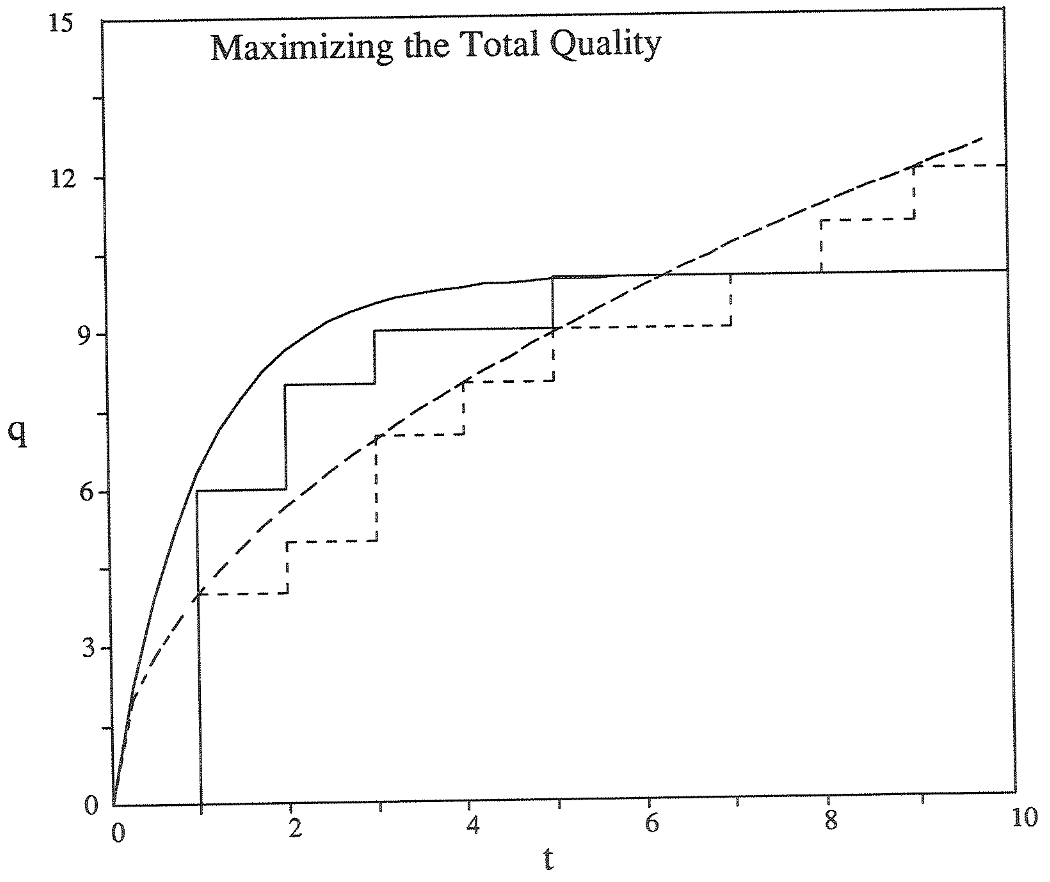
Let $T = 10$. The reward functions for these two rules are given below.

Reward functions q_1 and q_2:

Discrete reward function q_1										
t_1	1	2	3	4	5	6	7	8	9	10
q_1	4	5	7	8	9	9	10	11	12	12

Discrete reward function q_2										
t_2	1	2	3	4	5	6	7	8	9	10
q_2	6	8	9	9	10	10	10	10	10	10

The Lagrange multipliers method can be applied as follows. First, we interpolate and bound the two sets of data points with two continuous and differentiable functions f_1 and f_2, obtaining $f_1(t_1) = 4t_1^{1/2}$, $f_2(t_2) = 10(1 - e^{-t_2})$. The graph below shows the plots of the two discrete reward functions and their respective approximate continuous functions. The discrete reward function q_1 and its corresponding approximate function f_1 are plotted in dotted lines. The discrete reward function q_2 and its corresponding approximate function f_2 are plotted in solid lines.



----------- $q_1(t_1),\ f_1(t_1) = 4\sqrt{t_1}$

__________ $q_2(t_2),\ f_2(t_2) = 10(1 - e^{-t_2})$

Figure 8.1. Continuous Functions f_1 and f_2 Approximating the Discrete Functions q_1 and q_2.

The side constraint of this problem is: $t_1 + t_2 = T = 10$. Both t_1 and t_2 must be non-negative because a rule cannot fire a negative number of times. We have:

$$H(t_1, t_2, \lambda) = f(\overline{t}) - \lambda * g(\overline{t})$$
$$= f_1(t_1) + f_2(t_2) - \lambda * (t_1 + t_2 - T)$$
$$= 4t_1^{1/2} + 10(1 - e^{-t_2}) - \lambda(t_1 + t_2 - 10).$$

Differentiating $H(t_1, t_2, \lambda)$ with respect to t_1, t_2, and λ and then setting each derivative to 0, we obtain the following three equations:

$$\frac{\partial H}{\partial t_1} = 2t_1^{-1/2} - \lambda = 0, \ \textit{(eq. 1)}$$
$$\frac{\partial H}{\partial t_2} = 10e^{-t_2} - \lambda = 0, \ \textit{(eq. 2)}$$
$$\frac{\partial H}{\partial \lambda} = -(t_1 + t_2) + 10 = 0. \ \textit{(eq. 3)}$$

Combining the first two equations, we obtain two equations with two unknowns. Solving for t_1 and t_2:

$$2t_1^{-1/2} - 10e^{-t_2} = 0$$
$$t_1 + t_2 = 10.$$

The values for t_1 and t_2 are 7.391 and 2.609 respectively. Because these optimal values are not integral, we first truncate t_1 and t_2 to obtain $t_1 = 7$ and $t_2 = 2$. We are then left with one extra time unit which can be used to fire a rule once. We allocate this extra time unit to the rule that will add the largest marginal reward to R. Ties are broken arbitrarily. In our example, the marginal reward for firing rule 1 or rule 2 is 1 in either case. We select rule 2 to fire for another time to obtain a total reward = 19, with $t_1 = 7$ and $t_2 = 3$. For programs with more rules, an integral solution is obtained by truncating the Lagrange multiplier solution and using a greedy algorithm to select rules to fire to maximize the marginal reward. In this example, this also turns out to be the optimal solution to the integer optimization problem.

It should be noted that it is unclear whether the quality of the solutions obtained by the Lagrange multiplier approach is in general better than that of a greedy algorithm for solving the knapsack problem. However, this approach can handle more general reward functions, and more importantly, it lends itself to parameterizing the solution with respect to the response time constraint T and the reward objective C. For example, we may use a quadratic B-spline interpolation algorithm to interpolate and bound each set of discrete reward values to obtain n quadratic functions. After taking the partial derivatives as required by the Lagrange multiplier method, we have $n + 1$ linear equations. Given the values of T and C at run time, these equations can be efficiently solved, e.g., by the Gaussian elimination algorithm. The use of a continuous function to bound the rewards also gives us a better handle on guaranteeing that an equational rule-based program can meet some minimum performance index in bounded time than *ad hoc* greedy algorithms which must be analyzed for individual reward functions. Such guarantees are of great importance for safety-critical applications.

8.3. The Scheduling Approach

In this approach, we optimize the scheduler to select the rules to fire such that a fixed point is always reached within the response time constraint. This assumes, of course, that there is at least one sufficiently short path from a launch state to every one of its end-points. To implement this approach, the scheduler must have prior knowledge about the interaction among rules in the program. In particular, it must know the general order of firing the enabled rules such that the number of rule firing to reach a fixed point is minimized. One way to acquire this scheduling knowledge is to decompose the original program into independent modules of rules and determine an optimum order to fire these module of rules.

To decompose the original program into independent modules, we make use of the analysis results derived from the general analysis algorithm. Recall that the general analyzer iteratively selects **forward-independent** subsets of rules that are guaranteed to reach a fixed point in bounded time. A set of rules is said to be **forward-independent** in the context of the analysis process iff its fixed-point convergence depends only on the rules contained in this set or in those sets of rules that have been selected (these sets of rules always reach a fixed point in bounded time) during the previous iterations of the analysis process, but not on the fixed-point convergence of the remaining rules in the simpler rewritten program(s).

Each set of rules selected by the general analysis procedure is called a **module**. Let A_1 and A_2 be two modules selected in the same first iteration by the analysis procedure. Let B be the module selected in the second iteration by the analysis procedure. Then modules A_1 and A_2 are said to be forward-independent relative to module B. Modules A_1 and A_2 are said to be **parallel**.

We are now ready to show how the knowledge about these modules and the order in which they are selected by the analysis procedure can be used by the scheduler to optimally select rules for execution at run time. Let A and B be respectively the first and the second forward-independent modules selected by the analysis procedure. Let t_A and t_B be respectively the maximum number of rule firings required for module A and module B to converge to a fixed point. Let $t_{A \cup B}$ be the time it takes for the rules contained in both modules to reach a fixed point.

A key observation is the following. Since module A is forward-independent relative to module B, once the rules contained in module A have reached a fixed point, firing any rules contained in module B will not affect the fixed point convergence of module A. Rules contained in module B may not reach a fixed point before rules contained in module A have reached a fixed point. Thus firing rules contained in module B before those rules contained in module A have converged to a fixed point will not decrease $t_{A \cup B}$, but may increase $t_{A \cup B}$. Consequently, firing the rules contained in module A until a fixed point is reached in module A and then firing those rules contained in module B until a fixed point is reached in module B will minimize the total number of rule firings. Thus the optimum worst-case time needed for the rules contained in both modules to reach a fixed point using this schedule is $t_A + t_B$. With the rules decomposed in this fashion, this computation time is the best one can guarantee without further knowledge about the interaction of the rules within each module. This result can be extended to optimally schedule all modules of rules selected by the general analysis procedures. We now state a theorem about the relation **forward-independence** (**FI**) defined on the set of modules selected by the general analysis procedure.

Theorem 8.3.1.

The relation **forward-independence** (**FI**) induces a partial ordering on the set of modules selected by the general analysis procedure.

Proof:

A relation is a partial order iff it is transitive, irreflexive, and antisymmetric ([Horowitz and Sahni 82]). Irreflexivity implies that there are no directed cycles. Let A be the module selected at iteration i, and let B be the set of rules which are not selected and remain to be analyzed at iteration i. The analysis algorithm guarantees that A is forward-independent from B, thus $(A, B) \in$ **FI**. Let $B = B_1 \cup B_2$. Let B_1 and B_2 be the modules selected at iteration $i + 1$ and at iteration $i + 2$ respectively. Again, the analysis algorithm guarantees that B_1 is forward-independent from B_2, thus $(B_1, B_2) \in$ **FI**. Note that A must be forward-independent from B_1 and B_2 since $B_1 \subset B$ and $B_2 \subset B$. Since $(A, B_1) \in$ **FI** and $(B_1, B_2) \in$ **FI** imply $(A, B_2) \in$ **FI**, **FI** is transitive. Since a module cannot be forward-independent from itself, **FI** is irreflexive. From the discussion of the scheduling approach, it is obvious that **FI** is antisymmetric. If **FI** has a nontrivial directed cycle $(A_1, \ldots, A_n)$, where $n \geq 2$, then $(A_n, A_1) \in$ **FI** and also, by transitivity, $(A_1, A_n) \in$ **FI**. Since $A_1 \neq A_n$, antisymmetry is violated and thus **FI** cannot be a partial order. There cannot be any directed cycles. Since **FI** is transitive, irreflexive, and antisymmetric, it is a partial order.

□

We can graphically represent this set of modules by applying the following two rules:

(1) Each vertex in the graph represents each module selected.
(2) There is an edge from vertex A to vertex B iff module A is forward-independent relative to module B.

Note that the graph constructed using rules (1) and (2) is acyclic and layered. The set of modules selected at each iteration is called a layer. If a multiprocessor is available for executing a rule-based program with a graph as constructed using the above rules, then the number of processors needed to achieve maximum parallelism corresponds to the number of modules contained in the largest layer.

To further investigate the scheduling approach for minimizing the response time of a rule-based program, we have identified two important research problems:

(1) How to determine the size of each module such that the number of rule firings needed to reach a fixed point is always minimized?

(2) Given a fixed number of processors, determine the optimum way to allocate the parallel modules into these n processors such that the time needed to compute the respective fixed points of all parallel modules is minimized.

We have just described one simple but very effective way to solve the synthesis problem by using the analysis procedure to compute the optimum schedule for firing the rules in a program prior to its execution. It is our view that there are other methods for scheduling rules that can be exploited in this fashion. We expect that our ongoing research will discover alternative novel methods for scheduling rules or specifying the scheduler that are conducive to guarantee a given program satisfying certain properties will meet all timing constraints.

Chapter 9

Computer-Aided Design Tools

To put theory into practice, we have implemented a suite of computer-aided design tools for the analysis and synthesis of real-time rule-based decision systems. As in the design of most complex software systems, we envision the development of real-time rule-based systems to be an iterative process. Our goal is to speed up this iterative process by automating it as much as possible. Figure 9.1 shows the interaction of a designer with these software engineering tools. In each design cycle, the equational rule-based program is analyzed by the analysis tools for compliance with the timing and integrity constraints. Violations are passed to the designer who can then modify the program with the help of the synthesis tools until the program meets all the constraints. It should be emphasized, however, that the purpose of the design tools is not to encourage people to write sloppy programs and rely on the tools to fix them. Design tools are particularly necessary for rule-based programs because the addition and/or deletion of just one rule may change the behavior of a program drastically, and it is essential to guard against unintended interference between the work of individual members of a design team.

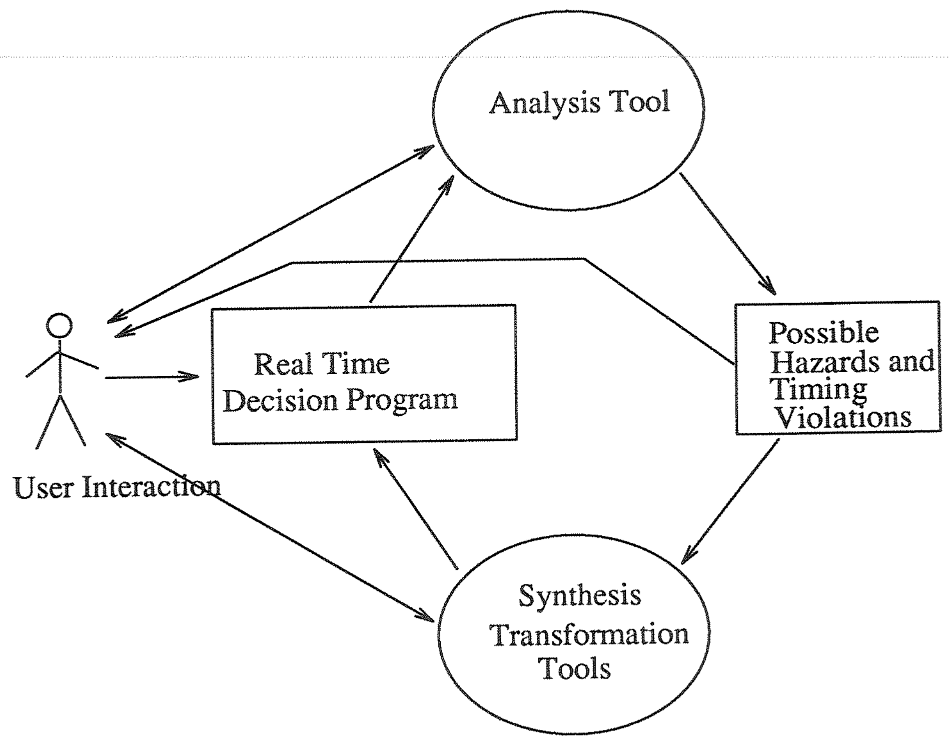

Analysis Tool
Real Time Decision Program
Possible Hazards and Timing Violations
User Interaction
Synthesis Transformation Tools

Figure 9.1. Development of Real-Time Decision Systems.

In previous chapters, we have described the *General Analysis Tool* and the **Estella** BCA specification facility. In this chapter, we shall describe the other modules of our computer-aided software engineering tools which have been implemented for the equational rule-based language **EQL**. For efficient execution, there is a translator of **EQL** programs into C code. We selected C as the target language since it is widely available and has efficient compilers. Since **EQL** has nondeterministic constructs, our translator generates the appropriate C code to simulate nondeterminism and parallelism on a sequential machine.

Our CAD tools include:

(1) A translator for transforming an **EQL** program into its corresponding state space graph as described in chapter 4. The state space graph serves as an intermediate form of the program under development and is used for mechanical analysis and synthesis.

(2) A temporal logic verifier for checking integrity assertions about **EQL** programs. (In the current implementation, these assertions can be expressed in a temporal logic called Computation Tree Logic (CTL) [Clarke, Emerson & Sistla 86].) Specifically, the verifier determines, given any launch state, whether some fixed points are always reachable in a finite number of iterations and whether the reachable fixed points are safe, i.e., satisfy the specified integrity constraints. If the given **EQL** program may not reach a fixed point from a particular launch state, then the temporal logic verifier will warn the designer of the existence of a cycle which does not have any path exiting from it.

(3) A timing analyzer for determining the maximum number of iterations to reach a safe fixed point, and the sequence of states traversed (along with the sequence of of rules fired) from the launch state to the fixed point. This helps the designer pinpoint the sets of rules that may constitute a performance bottleneck so that optimization efforts can be concentrated on them. The timing analyzer can also be used to investigate the performance of customized schedulers which are used to deterministically select a rule to fire when two or more rules are enabled.

A suite of practical prototyping tools has been implemented on a Sun Microsystems 3® work station running under BSD UNIX® to perform timing and safety analysis of real-time equational rule-based programs. These tools are

® UNIX is a registered trademark of AT&T Bell Laboratories. SUN is a registered trademark of SUN Microsystems Inc.

sufficiently practical for analyzing realistic real-time decision systems. To demonstrate their usefulness, we have used these tools to analyze:

(1) the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure Control System ([Helly 84]),

(2) the Integrated Status Assessment Expert System (ISA) ([Marsh 88]), and

(3) the Fuel Cell Monitoring Expert System (FCE) ([Marsh 88]).

Figure 9.2 illustrates the dependency among the modules in our tool system. The names of the modules and descriptions of their functions follow.

(1) **eqtc** - **EQL** to **C** translator
(2) **ptf** - **EQL** to finite state-space graph translator for a launch state
(3) **ptaf** - **EQL** to finite state-space graphs translator for all launch states
(4) **mcf** - CTL model checker (extended to cover fairness)
(5) **fptime** - Timing analyzer on state space graphs

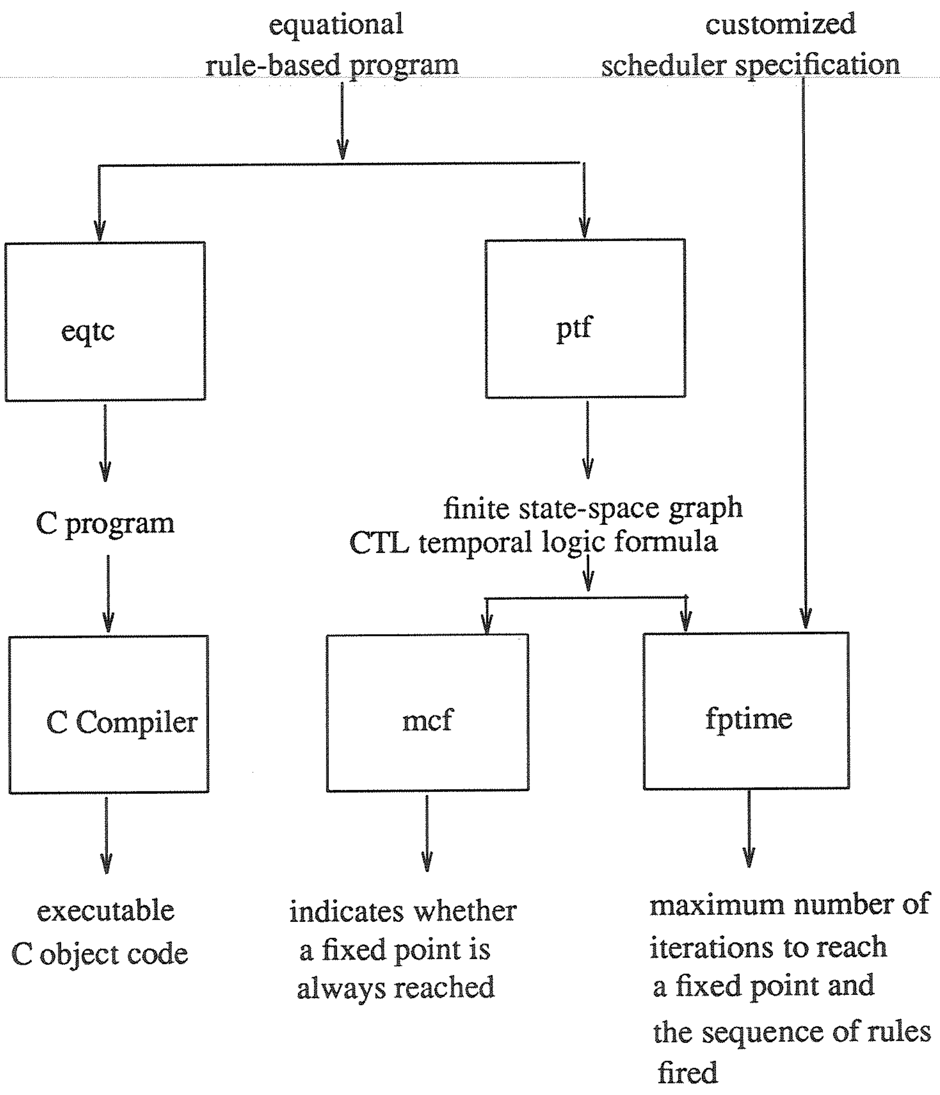


Figure 9.2. Computer-Aided Design Tools for Real-Time Decision Systems.

The module **eqtc** translates a program written in the language **EQL** into a C program suitable for compilation using the **cc** compiler under UNIX. As described earlier, **EQL** is a Unity-based language with nondeterministicallyscheduled rules and parallel assignment statements.

The module **ptf** translates an **EQL** program with finite domains (all variables have finite ranges) into a finite state-space graph which contains all the states reachable from the launch state corresponding to the initial values of the variables given in the program. It also generates the appropriate temporal logic formula for checking whether the program will reach a fixed point. **ptf** produces a file named *mc.in* which will be read by the fairness-extended model checker **mcf** and the timing analyzer module **fptime**. The file *mc.in* contains the internal representation of the state space graph of the corresponding **EQL** program.

The module **ptaf** is similar to **ptf**, but it automatically generates the complete state space graph (i.e., it generates all the states reachable from every launch state). **ptaf** invokes the model checker and the timing analyzer to determine whether the program will reach a fixed point in a finite number of iterations, starting from any launch state. If the **EQL** program indeed reaches a fixed point in a finite number of iterations starting from any launch state, **ptaf** informs the designer accordingly. Otherwise, **ptaf** stops at the first launch state for which the program may not reach a fixed point in a finite number of iterations and informs the designer about the unstable launch state.

The module **mcf** is a temporal logic model checker based on the Clarke-Emerson-Sistla algorithm for checking the satisfiability of temporal logic formulas written in CTL ([Clarke, Emerson & Sistla 86]). Our model checker assumes that strong fairness is observed by the scheduler, i.e., rules that are enabled infinitely often will eventually fire. Under this assumption, a cycle in the state space graph which has at least one edge exiting from it is not sufficient to cause the program not to reach a fixed point in a finite number iterations. (The program will leave the states in the cycle because the rule associated with the exit edge must eventually fire.) However, the model checker will warn the designer that the program may require a finite but unbounded number of iterations to reach a fixed point. The default scheduler used to schedule the next enabled rule to fire is fair and is based on a linear-congruential pseudo-random number generator.

The module **fptime** is a timing analyzer that computes the maximum number of iterations for a given program to reach a fixed point if at least one reachable fixed point exists. In addition, it provides the sequence of rule firings leading from the launch state to this fixed point. It can also compute the number of iterations to any other fixed point and the corresponding sequence of rule firings if the designer so desires. **fptime** has been designed so that a designer will be able to specify restrictions on the scheduler if it is desired to determine how the specified scheduling restrictions may affect the number of rule firings. This is useful for investigating the performance of customized schedulers. For a complete analysis of the distributed program of example 3.2 using the CAD tools, the reader is referred to chapter 5.

One of the principal goals of our computer-aided design tools is to provide maximum flexibility in the user-toolset interface so as to allow, for instance, the analyzer to be tailored to particular application domains so that the analysis can be performed with upmost efficiency. While we would like to mechanize the analysis and synthesis procedures as much as possible, it is our belief that allowing user interaction which would speed up and refine the analysis and synthesis processes does not defeat our goal. This is reflected in one of our research directions which is the development of a standard language **Estella** for specifying behavioral constraint assertions and special forms of rules for which the analysis problem can be solved efficiently.

Another goal is to make this suite of tools as general as possible so that it is capable to analyze a wide range of realistic real-time rule-based programs. A third goal is to allow for future extensions to these tools as the need arises with minimal efforts. Both of these goals are accomplished by modularizing the different phases and components of the tools and by utilizing standardized data structures shared by the different modules. To put theory into practice, this suite of tools is constantly updated to incorporate new analysis and synthesis techniques as they are being discovered.

The current version of our programming tools consists of over 21,000 lines of C, YACC, and LEX codes. We have given demonstration of this preliminary version of our programming tools for real-time rule-based decision systems to visitors from International Business Machines (IBM), National Science Foundation (NSF), National Aeronautics and Space Agency (NASA), Office of Naval Research (ONR), Mitre, Texas Instrument (TI), Software Research Department of Hitachi (Japan), and a visitor from UK. We have recently distributed the version 1.0 of the programming tools to Texas Instrument Inc, Dallas.

Chapter 10

Future Research Directions

Having formalized two fundamental problems, namely, the analysis problem and the synthesis problem, with respect to our real-time decision system model and having developed concrete techniques for tackling both of them, we are now in a position to explore possible avenues for further research in this increasingly important area. In the following sections, we identify several research directions within our model and worthwhile problems associated with each direction, and provide feasible approaches for attacking some of these problems.

10.1. Structured Design of Real-Time Rule-Based Systems

The analysis of rule-based programs often involves exhaustive state-space graph checks and thus is computationally-intensive. In the worst case the analysis may require exponential computation time as a function of the number of variables in the program and is in general undecidable for non-finite-domain programs. Although polynomial-time static analysis techniques which take advantage of special forms of rules have been developed for a sizable class of real-time rule-based programs implemented in **EQL** for which exhaustive state-space graph analysis is unnecessary, the verification of large and complex programs remains computationally intensive. To reduce the computational complexity of the analysis procedure, a discipline of programming real-time rule-based systems must be developed and imposed to expedite the analysis procedure. We would like to develop a structured design approach aimed at facilitating the verification of fixed-point convergence and managing the complexity of the expert system by ensuring modularity and providing structure to the specification and design of these time-critical decision systems.

The objective of structuring **EQL** programs is thus twofold. First, it facilitates verification (by a combination of static analysis and exhaustive state graph analysis) of fixed point convergence. Second, it enforces modularity and guides the specification and design of real-time rule-based programs. The decomposition of the program into suitable modules may even help identify sets of rules which can fire in parallel without conflicts. Different criteria may be used to decompose a program into modules with desirable inter-module and intra-module properties.

10.2. Analysis of Self-Stabilizing EQL Programs

A technique for transforming any non-self-stabilizing acyclic **EQL** program into one that is self-stabilizing has been proposed recently by Gouda and Rosier ([Gouda & Rosier 88]). We show that any self-stabilizing program obtained from a non-self-stabilizing acyclic program using the Gouda-Rosier method described in [Gouda & Rosier 88] can be analyzed using the techniques described in chapter 5. In particular, we show that a self-stabilizing program is in special form **A** if its non-self-stabilizing counterpart is in special form **A**. We conjecture that other classes of self-stabilizing programs obtained from their non-self-stabilizing counterparts which are in other special forms using a more general Gouda-Rosier transformation method are also in special forms. If this conjecture is verified, then this result is significant because it allows us to analyze self-stabilizing programs whose non-self-stabilizing counterparts are in special form with the facility provided by static analysis techniques.

Any non-self-stabilizing program p in special form **A** can be transformed into a self-stabilizing program q as follows. For every set of rules (each of which is called a *non-self-stabilizing rule set*) in program p with exactly the same variable on the left side:

```
(* 1. *)    $v_i := B_i(v)$ IF $C_i(v)$
    :            :
    :            :
(* m. *) [] $v_i := D_i(v)$ IF $E_i(v)$
```

We add the following rule:

```
(* m'. *) [] $v_i := \overline{v_i}$  IF $\sim C_i(v) \wedge \cdots \wedge \sim E_i(v)$
```

where $\overline{v_i}$ is the initial value of v_i. We call this rule m' a *new* rule for the non-self-stabilizing rule set consisting of rules 1–m. The resulting program q is guaranteed to be self-stabilizing ([Gouda & Rosier 88]).

10.2.1. The Preservation of Special Form A

In this section, we show that the self-stabilizing program obtained using the method described in the preceding section is also in special form A. For ease of discussion, we re-state the conditions for special form A.

A set of rules are said to be in special form A if all of the following three conditions hold.

(A1) Constant terms are assigned to all the variables in L, i.e., $R = \varnothing$.
(A2) All of the rules are compatible pairwise.
(A3) $L \cap T = \varnothing$.

Theorem 10.2.1.

If a non-self-stabilizing program p is in special form A, then the self-stabilizing version q obtained by the Gouda-Rosier method is also in special form A.

Proof:

We need to show that program q satisfy all three conditions of special form A. Let L^s, R^s and T^s denote the sets of variables appearing in the LHS, RHS and tests of the new rules.

Satisfaction of condition A1: The only new values assigned are the initial values $\overline{v_i}$ of the program variables. Since for all i, $\overline{v_i}$ is the initial value of v_i, every $\overline{v_i}$ can be treated as a constant throughout the execution of the program. Since $R = \emptyset$, hence $R^s \cup R = \emptyset$ and thus condition (A1) is satisfied.

Satisfaction of condition A2: Test m' and test i, $i = 1,...,m$, are mutually exclusive (condition CR1 is satisfied) because a conjunct and its negation cannot be true at the same time and thus the new rule is compatible with every rule in the corresponding non-self-stabilizing rule set consisting of rules $1-m$. Next, we need to show that this new rule is compatible with every other new not in this non-self-stabilizing rule set since the variables on the left sides are also different and thus condition CR2 is also satisfied. The above reasoning applies to every new rule added to program p. Hence, we can conclude that all rules in program q are compatible pairwise and thus condition A2 is satisfied.

Satisfaction of condition A3: The variables appearing in the test part of each new rule are exactly those appearing in the test parts of its corresponding non-self-stabilizing rule set. The variable appearing in the left side of each new rule is also the same as those appearing in the left parts of its corresponding non-self-stabilizing rule set. Therefore, $L^s = L$ and $T^s = T$. Hence $(L^s \cup L) \cap (T^s \cup T) = \emptyset$ and thus condition (A3) is satisfied.

Since the self-stabilizing program q satisfies conditions A1, A2, and A3, we can conclude that it is in special form A.

□

One of our goals is to extend the applicability of our general analysis strategy and our programming tools to the analysis of self-stabilizing rule-based programs derived using the Gouda-Rosier method. The result described above is a step in this direction.

10.3. Synthesis of Real-Time Rule-Based Systems with Bounded Response Time

In chapter 8, we have formalized the general synthesis problem and have identified two general approaches for solving the synthesis problem. In particular, we have shown how information derived from the general analysis algorithm can be used to provide an optimal schedule for firing the rules in a rule-based program. We have barely scratched the surface of possible solutions to the synthesis problem. We now explore in more details the transformation approach.

The scheduling approach is sufficient to guarantee that the specified timing constraints are satisfied provided that there is at least one sufficiently short path from a launch state to every one of its end-points. For programs that do not meet this criterion, the transformation approach must be taken to obtain extended† programs that meet both the timing and integrity constraints. What is needed is a systematic procedure for transforming the given equational rule-based program by adding, deleting, and/or modifying rules. We give a sketch of the proposed synthesis algorithm.

Outline of the Synthesis Algorithm:

† The reader might want to consult chapter 8 for definitions of terms.

[1] Given an **EQL** program that does not meet the specified timing constraints, construct the corresponding **rule transition program graph.**

[2] Perform computations on the rule transition program graph to obtain required values for synthesis such as shortest paths.

[3] Modify the rule transition program graph using a synthesis procedure and the above values to obtain a *new* rule transition program graph.

[4] Generate a new **EQL** program from the modified rule transition program graph; this **EQL** program is guaranteed to always reach a fixed point in bounded time.

Each of the first three steps of this proposed synthesis algorithm constitutes a nontrivial research problem which must be resolved before we can apply this algorithm to synthesize programs that meet both the timing constraints as well as the integrity constraints. We first explore techniques for implementing the first step of the algorithm.

Performing an exhaustive analysis on the global state-space graph representing a program in order to generate an optimized version is not practical. This motivates us to develop a more succinct graph upon which the analysis and optimization procedures can be applied. One space-inefficiency problem with the conventional global state transition graph representation of rule-based programs is that each possible state may have to be represented by this graph. Furthermore, the firing of the same rule is represented by more than one edge in the graph to signify that that same rule may be fired from different states. Although temporal logic model-checking algorithms exist which have time complexity quadratic and even linear in the size of the global state transition graph (number of nodes and edges), the size of this graph is in the worst-case exponential in the number of variables in the program, or more precisely, $(\prod_{i=1}^{i=n} |X_i| \cdot \prod_{j=1}^{j=m} |S_j|)$ where $|X_i|$, $|S_j|$ are respectively the size of the domains of the i^{th} input and j^{th} program variable (please refer to chapter 5). Therefore, the analysis problem in the case of finite-domain programs is still PSPACE-complete in the worst case.

To alleviate this problem, we propose a new and potentially more compact representation of rule-based programs. We call this representation a **rule transition graph** because each of its nodes represents a distinct instantiation of a rule and transitions correspond to the order in which rules may fire. For instance, an edge from node A to node B signifies that rule B may fire following the firing of rule A.

We have just begun to study the feasibility of this representation and thus have not fully accessed its expressive power and its storage requirements as compared to the state transition graph representation. Hence, the space savings stated herein remains a conjecture that has to be rigorously checked. If this conjecture is true, we may develop faster state-space based or rule-space based analysis procedures, or use existing model-checking algorithms to perform more efficient analysis on this smaller graphical representation of the program. It is our belief that this development will also allow more tractable efficient synthesis procedures which operate on graphs.

10.4. Extending the EQL Language

EQL does not have provisions for declaring and using pattern (structured) variables as those found in popular rule-based languages such as OPS5. A new language called **MRL** has been built on the core language **EQL** ([Wang, Mok & Cheng 90]). As in OPS5-like languages, the working memory for an **MRL** production system may vary in size and content. However, **MRL** is different from traditional rule-based languages in that the semantics of **MRL** is a simple extension of that of **EQL**. By maintaining the same semantics of fixed point convergence as a termination condition for both **EQL** and **MRL** programs, it is possible for any given working memory, an **MRL** program can be instantiated into an equivalent **EQL** program by macro expansion of its rules.

By parameterizing the response time of the corresponding **EQL** programs by the size of the working memory of the **MRL** program, we can use the techniques developed in this dissertation to analyze the response time of the **MRL** program. To analyze an **MRL** program with a variable-size working memory, new techniques may need to be developed. However, owing to the **EQL**-like semantics of **MRL**, we believe that it is possible to extend techniques presented in this dissertation to analyze **MRL** programs directly. This is certainly one research direction worth pursuing.

Chapter 11

Concluding Remarks

This dissertation has been roughly divided into five parts. The first part (chapter 1 through chapter 4) formalizes the research problems associated with real-time rule-based decision systems and provides examples illustrating these problems. The rule-based language **EQL** is developed for programming real-time decision systems. The second part (chapter 5 through chapter 7) formalizes the analysis problem and describes a powerful and efficient analysis methodology for solving this problem. The applicability of this analysis technique is further enhanced by the introduction of a facility with which the rule-based programmer can specify domain-specific knowledge in the language **Estella** in order to validate the performance of an even wider range of programs. The third part (chapter 8) formalizes the synthesis problem and describes concrete methods for solving this problem. The fourth part (chapter 9) describes a suite of computer-aided software engineering (CASE) tools based on the solution strategies presented in chapter 5 through chapter 8 and demonstrates how they can be used to perform analysis on realistic real-time rule-based systems developed by industry. The fifth part (chapter 10) explores avenues for future research.

Our research opens up a number of interesting problems in the analysis and synthesis of real-time decision systems and has provided concrete approaches for tackling them so that we can rely on computers to perform *intelligent* tasks in bounded time. It is expected that further research along the directions outlined in chapter 10 will provide even greater insight into the development of better solutions for solving these problems. This dissertation has contributed to the advancement of the state-of-the-art in the analysis and synthesis of real-time rule-based decision systems. Implemented using our novel analysis and synthesis techniques, the computer-aided design tools are capable of analyzing realistic real-time rule-based software whose complexity exceeds the computational capability of previously available analysis technology.

Appendix A

EQL: A Rule-Based Language for Programming Real-Time Decision Systems

Real-time rule-based decision programs must react to events in the external environment by performing decision-intensive computation sufficiently fast to meet some specified timing constraints. In this appendix, we describe the syntax and the semantics of **EQL**, a rule-based language designed for programming real-time decision systems and used for the investigation of whether and how performance objectives can be met when rule-based programs are used to perform safety-critical functions in real time.

A.1. Introduction

An **EQL** program has a set of rules for updating variables which denote the state of the physical system under control. The firing of a rule computes a new value for one or more state variables to reflect changes in the external environment as detected by sensors. Sensor readings are sampled periodically. Every time sensor readings are taken, the state variables are recomputed iteratively by a number of rule firings until no further change in the variables can result from the firing of a rule. The equational rule-based program is then said to have reached a *fixed point*. Intuitively, rules in an **EQL** program are used to express the constraints on a system and also the goals of the controller. If a fixed point is reached, then the state variables have settled down to a set of values that are consistent with the constraints and goals as expressed by the rules.

EQL differs from the popular *expert system* languages such as OPS5 in some important ways. These differences reflect the goal of our research, which is not to invent yet another *expert system shell* but to investigate whether and how performance objectives can be met when rule-based programs are used to perform safety-critical functions in real time. Whereas the interpretation of a language like OPS5 is defined by the *recognize-act cycle* ([Forgy 81]), the basic interpretation cycle of **EQL** is defined by fixed point convergence†. It is our belief that the time it takes to converge to a fixed point is a more pertinent measure of the response time of a rule-based program than the length of the *recognize-act cycle*. More importantly, we do not require the firing of rules that lead to a fixed point to be implemented sequentially; rules can be fired in parallel if they do not interfere with one another. The definition of response time in terms of fixed point convergence is architecture independent and is therefore more robust. The rest of this appendix is organized as follows. Section A.2 describes the syntax of the **EQL** language. Section A.3 explains the semantics of the **EQL** language. Section A.4 gives the syntax diagrams of the **EQL** language for ready reference.

A.2. The EQL Language

An **EQL** program is organized like the model in Figure A.1. It is composed of four major distinct sections: the declaration section, the initialization section, the rule section, and the output section. The syntax of **EQL** follows closely that of the language **Pascal**. **EQL** programs are entirely free format, with no restrictions on columns or spacing. A comment can be indicated by enclosing it within the character pairs (* *).

† The fixed point semantics of **EQL** follows closely that of the language **Unity** ([Chandy & Misra 88]). Both **Unity** and the work described herein are part of a coordinated research effort to explore the foundation of programming concurrent systems at the University of Texas at Austin.

```
PROGRAM name;
CONST declaration;
VAR declaration;
INPUTVAR declaration;
INIT
        statement,
        statement,
            :
            :
        statement
RULES
       rule
    [] rule
        :
        :
    [] rule
END.
```

Figure A.1. The Organization of an EQL Program.

Identifiers are used in the program to name variables, constants, and the program name. The rules for forming an identifier are as follows. All alphabetic characters used are lowercase. The first character must be a lowercase alphabetic character: 'a'..'z'. All succeeding characters must be alphabetic, numeric or the underscore character '_'. No special characters or punctuation marks are allowed and no embedded blanks are allowed. Identifiers may be as long as desired subject to the restrictions of the system in which **EQL** is implemented.

A.2.1. The Declaration Section

The declaration of an **EQL** program consists of three different types of declaration: CONST, VAR and INPUTVAR. Each type of declaration must appear only once and in the order indicated in Figure A.1.

A.2.1.1. The CONST Declaration

The CONST declaration assigns a name to a scalar constant. There are no predefined constants in **EQL** and thus all constants used in the program must be declared, including the values of the boolean constants *true* and *false*. For example, the following declaration declares four constants.

```
CONST
      false  = 0 ;
      true   = 1 ;
      bad    = 2 ;
      good   = 3 ;
```

A.2.1.2. The VAR declaration

All variables used in the program except input variables must be declared in the VAR section. Input variables are those that do not appear on the left-hand-side of any assignment statement. They are used to store the values read from sensors attached to the external environment. For example, the following VAR declaration declares three variables of type BOOLEAN.

```
VAR
      sensor_a_status, sensor_b_status, object_detected : BOOLEAN ;
```

A.2.1.3. The INPUTVAR declaration

All input variables used in the program must be declared in the INPUTVAR section. For example, the following INPUTVAR declaration declares three input variables of type INTEGER.

```
INPUTVAR
        sensor_a, sensor_b, sensor_c : INTEGER ;
```

A.2.2. The Initialization Section INIT

All non-input variables are assigned initial or default values in the initialization section INIT before the start of the firing of the rules in the RULES section of the program. For example, the following statements initialize the variables: *sensor_a_status, sensor_b_status*, and *object_detected.*

```
INIT
        sensor_a_status := good,
        sensor_b_status := good,
        object_detected := false
```

A.2.3. The RULES Section

The RULES section is composed of a finite set of rules each of which is of the form:

$$a_1 := b_1 \; ! \; a_2 := b_2 \; ! \; \cdots \; a_m := b_m \;\; \text{IF} \;\; \mathit{test}$$

where b_i is an arbitrary expression whose value is to be assigned to the variable a_i when the rule fires, $m \geq 1$, and *test* is an arbitrary logical expression. A rule has three parts:

(1) LHS: the left-hand-side of the multiple assignment statement,
(2) RHS: the right-hand-side of the multiple assignment statement, and
(3) EC: the enabling condition (also referred to as the test).

We define three sets of variables for an equational rule-based program.

$L = \{ v \mid v$ is a variable appearing in LHS $\}$
$R = \{ v \mid v$ is a variable appearing in RHS $\}$
$T = \{ v \mid v$ is a variable appearing in EC $\}$

Rules are separated by the delimiter characters '[]'. For example, six rules are shown below.

```
(* 1 *)    object_detected := true  IF sensor_a = 1  AND  sensor_a_status = good
(* 2 *) [] object_detected := true  IF sensor_b = 1  AND  sensor_b_status = good
(* 3 *) [] object_detected := false IF sensor_a = 0  AND  sensor_a_status = good
(* 4 *) [] object_detected := false IF sensor_b = 0  AND  sensor_b_status = good
(* 5 *) [] sensor_a_status := bad   IF  sensor_a <> sensor_c  AND  sensor_b_status = good
(* 6 *) [] sensor_b_status := bad   IF  sensor_b <> sensor_c  AND  sensor_a_status = good
```

For this set of rules, the three sets of variables L, R, T are:

$L = \{$ object_detected, sensor_a_status, sensor_b_status $\}$,
$R = \varnothing$, and
$T = \{$ sensor_a, sensor_b, sensor_a_status, sensor_b_status, sensor_c $\}$.

A.2.4. The Output Section

The TRACE statement prints the values of the specified variables following the firing of a rule in each cycle. For example, the following TRACE statement prints the values of the variables *sensor_a_status, sensor_b_status,* and *object_detected* following the firing of any rule.

```
TRACE sensor_a_status, sensor_b_status, object_detected
```

The PRINT statement prints the values of the specified variables only after the entire program has reached a fixed point. For example, the following PRINT statement prints the values of the same variables after the program has reached a fixed point.

```
PRINT sensor_a_status, sensor_b_status, object_detected
```

A.2.5. Implementation of EQL

EQL is currently implemented to run under BSD UNIX.† Two translators (**eqtc** and **eqc**) have been implemented for translating **EQL** programs into **C** programs for compilation using the **cc C** compiler available on the UNIX system. This allows the execution of **EQL** program on sequential UNIX machines.

A.3. Semantics of the EQL Language

An enabling condition (test) is a predicate on the variables in the program. A rule is enabled if its test becomes true. A rule firing is the execution of the multiple assignment statement of an enabled rule. A multiple assignment statement assigns values to one or more variables in parallel. The expressions must be side-effect free. The execution of a multiple assignment statement consists of the evaluation of all the RHS expressions, followed by updating the LHS variables with the values of the corresponding expressions. An invocation of an

† UNIX is a trademark of Bell Laboratories.

equational rule-based program is a sequence of rule firings (execution of multiple assignment statements whose tests are true). When two or more rules are enabled, the selection of which rule to fire is up to the implementation, but any rule that stays enabled must eventually be fired.

An equational rule-based program is said to have reached a fixed point when either: (1) none of the rules is enabled, or (2) if firing of any enabled rule will not change the value of any variable in L. Intuitively, when a fixed point is reached, a rule-based program has arrived at a consistent evaluation of its environment.

Some variables appearing in an equational rule-based program are *input variables*, and their values are determined by sensor readings from the external environment at the beginning of each invocation of the program. Input variables do not appear on the left hand side of any assignment statement. The other variables in a program are called program variables. Although **EQL** does not distinguish program variables into different types; for most implementations, program variables are classified into either *decision variables* or *temporary variables*. Decision variables are used to control the physical system and to communicate with the outside world after a fixed point is reached, e.g., signaling system status, giving helpful advice to human operators, etc. Temporary variables are used for storing information about the environment and for communication between rules within the program.

In the code segment in section A.2.3, *sensor_a*, *sensor_b*, and *sensor_c* are the input variables; *object_detected*, *sensor_a_status*, and *system_b_status* are the program variables. The decision variable is *object_detected*.

We now show a complete sample **EQL** program.

Example A.3.1.

```
(* Example EQL Program *)
PROGRAM distributed;
CONST
        false  = 0;
        true   = 1;
        a      = 0;
        b      = 1;
VAR
        sync_a,
        sync_b,
        wake_up,
        object_detected    : BOOLEAN;
        arbiter            : INTEGER;
INPUTVAR
        sensor_a,
        sensor_b     : INTEGER;
INIT
        sync_a       := true,
        sync_b       := true,
        wake_up      := true,
        object_detected := false,
        arbiter      := a
RULES
(* process A *)
        object_detected := true ! sync_a := false
                IF (sensor_a = 1) AND (arbiter = a) AND (sync_a = true)
     [] object_detected := false ! sync_a := false
                IF (sensor_a = 0) AND (arbiter = a) AND (sync_a = true)
     [] arbiter := b ! sync_a := true ! wake_up := false
                IF (arbiter = a) AND (sync_a = false) AND (wake_up = true)
```

```
(* process B *)
    [] object_detected := true ! sync_b := false
            IF (sensor_b = 1) AND (arbiter = b) AND (sync_b = true)
                AND (wake_up = true)
    [] object_detected := false ! sync_b := false
            IF (sensor_b = 0) AND (arbiter = b) AND (sync_b = true)
                AND (wake_up = true)
    [] arbiter := a ! sync_b := true ! wake_up := false
            IF (arbiter = b) AND (sync_b = false) AND (wake_up = true)

      TRACE object_detected
      PRINT sync_a, sync_b, wake_up, object_detected, arbiter, sensor_a,
      sensor_b
END.
```

In this example, the input variables are: *sensor_a* and *sensor_b*, and the program variables are:

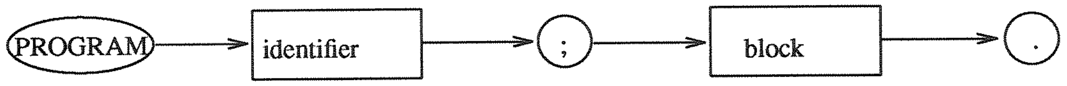
object_detected, sync_a, sync_b, arbiter, and *wake_up*. The three sets of variables L, R, T are:

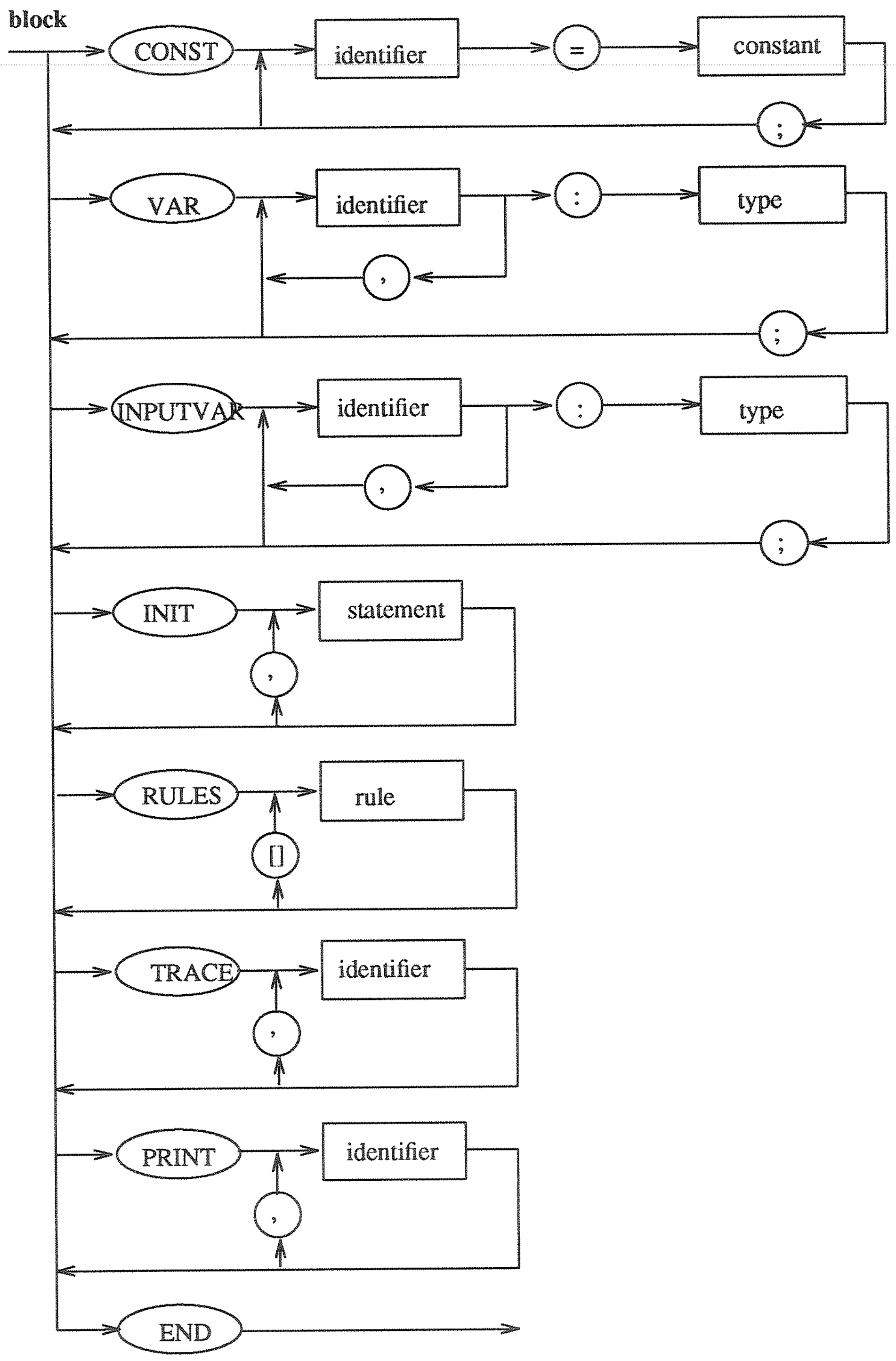
L = { object_detected, sync_a, sync_b, arbiter, wake_up },
R = $\emptyset$, and
T = { sensor_a, sensor_b, arbiter, sync_a, sync_b, wake_up }.

A.4. Syntax Diagrams of the EQL Language

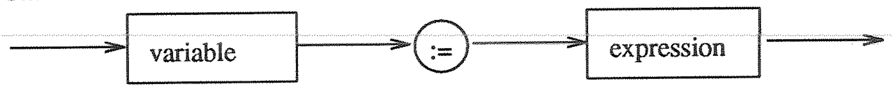
In this section, we formally define the syntax of the **EQL** language by means of syntax diagrams. An ellipse represents **EQL** reserved words or syntactic entities that are not defined further (e.g., a letter or a digit). A circle represents an **EQL** operator. A rectangle represents a syntactic entity that is defined by another syntax flow diagram.

program

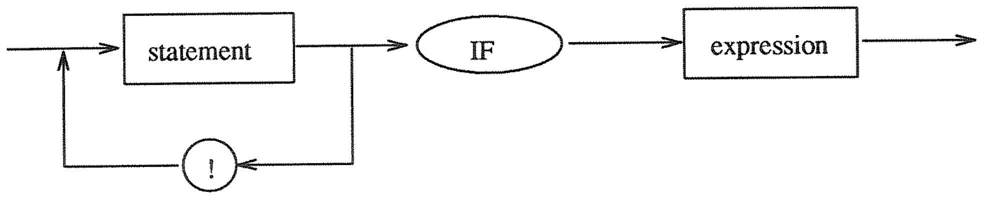



block
CONST
identifier
=
constant
;
VAR
identifier
,
:
type
;
INPUTVAR
identifier
,
:
type
;
INIT
statement
,
RULES
rule
[]
TRACE
identifier
,
PRINT
identifier
,
END

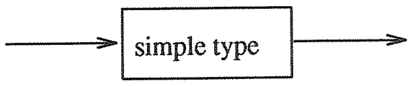
statement



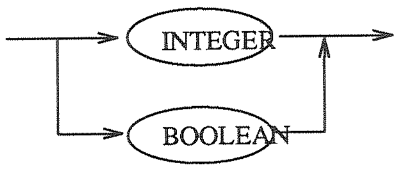
rule



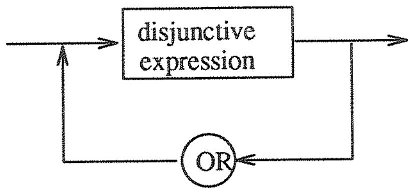
type



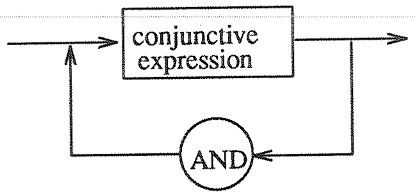
simple type



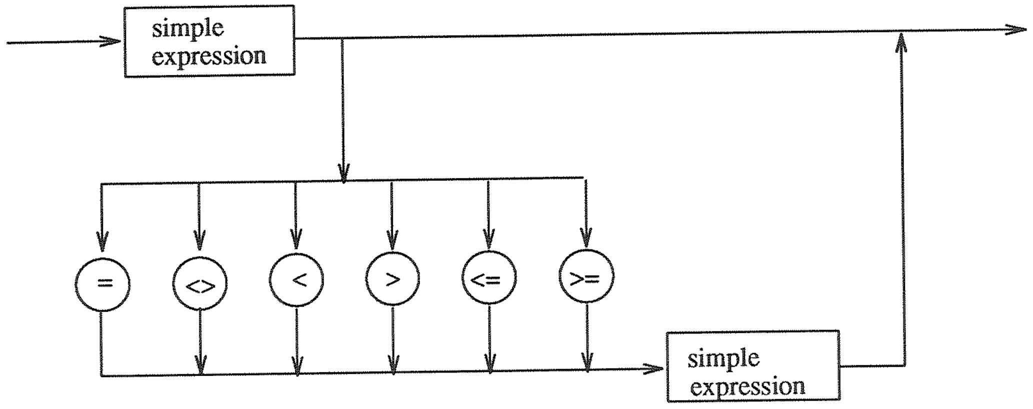
expression



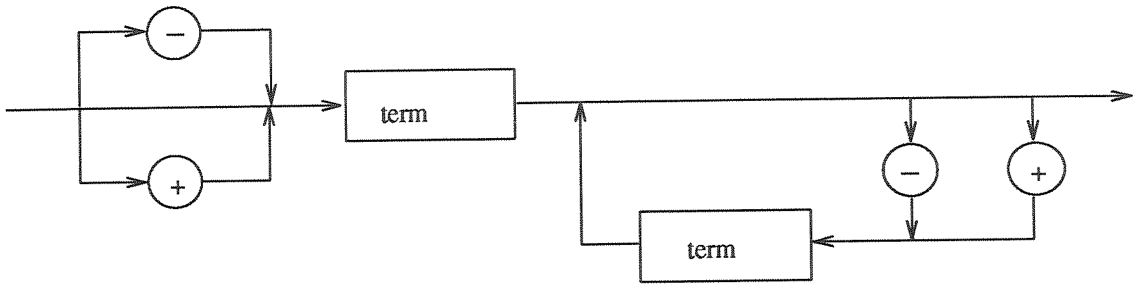
disjunctive expression



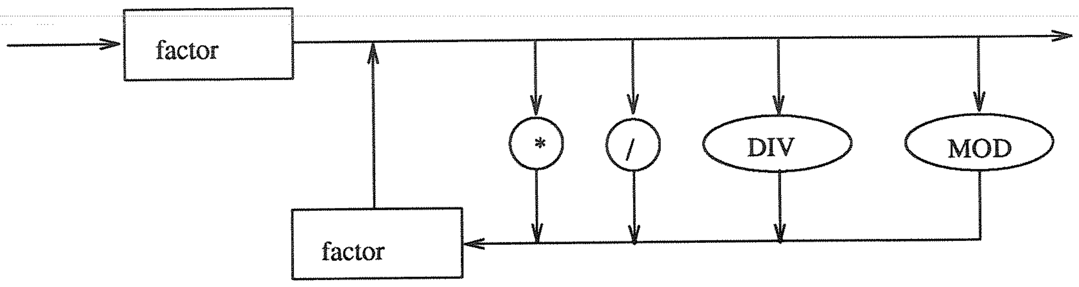
conjunctive expression



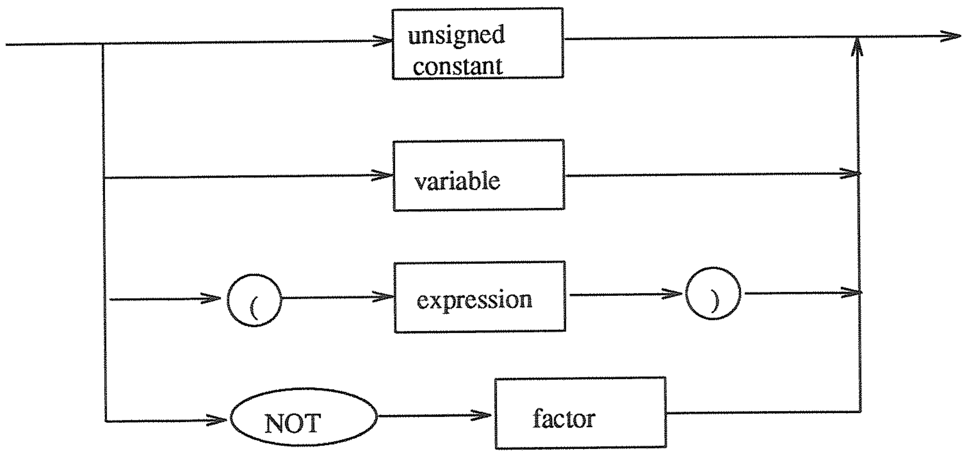
simple expression



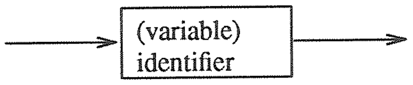
term



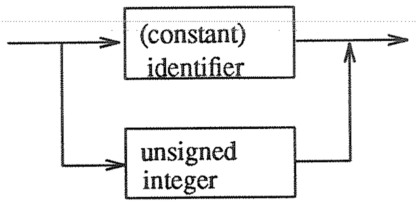
factor



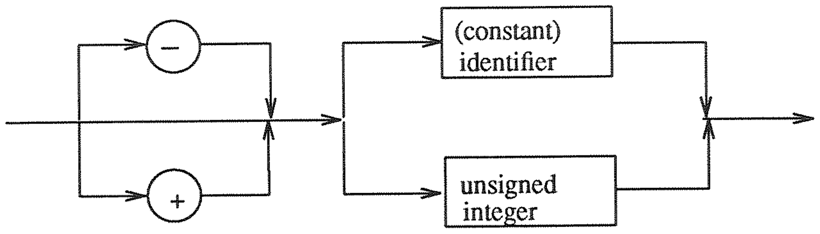
variable



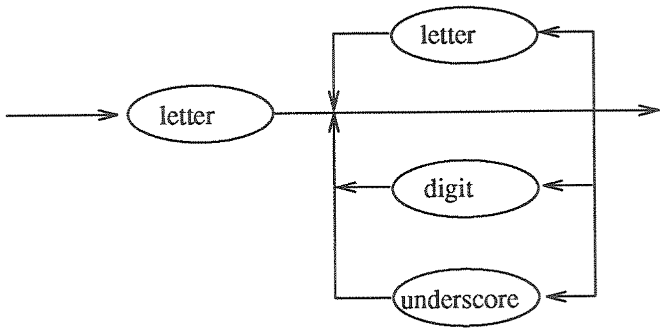
unsigned constant



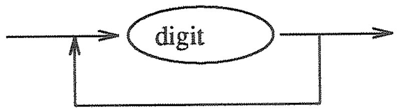
constant



identifier



unsigned integer



Appendix B

Context-Free Grammar for Estella

The top-level commands of the analyzer are:

```
check_command       | 'rp' /* read program */
                    | 'ls' /* load special form */
                    | 'sf' /* new special form */
                    | 'ps' /* print special forms */
                    | 'ds' /* delete special form */
                    | 'vm' /* verbose mode? */
                    | 'cs' /* compatible set */
                    | 'bc' /* break condition */
                    | 'an' /* analyze */
                    | 'ex' /* exit */
                    ;
```

The context-free grammar for **Estella** are specified in the YACC language.

```
estella_command     : special_form
                    | exception
                    ;

special_form        : special_form_name special_form_body end_mark
                    ;

special_form_name : 'special_form' IDENTIFIER ':'
```

```
                  ;

special_form_body : conditions
                  ;

end_mark      : 'end' '.'

conditions : condition
           | conditions ';' condition
           ;

condition     : specification
              | break_condition
              ;

exception     : compatible_set
              | break_condition
              ;

term          : var_set
              | variable
              | function_name '(' term arg2')'
              ;

elements_list: element
              | elements_list ',' element
              ;

element       : IDENTIFIER
              | var_set
              ;

arg2          :
              | ',' term
```

```
             ;

atom_formula : predicate_name '(' term arg2 ')'
             ;

specification : atom_formula
              | 'NOT' '(' specification ')'
              | '(' specification connective specification ')'
              | quantifier rule_list '(' specification ')'
              | quantifier subrule_list '(' specification ')'
              ;

connective   : 'OR'
             | 'AND'
             | '->'
             | '<->'
             ;

quantifier   : 'FORALL'
             | 'EXIST'
             ;

set_variable : 'L'
             | 'R'
             | 'T'
             | 'L' '[' irule_number ']'
             | 'L' '[' irule_number '.' isubrule_number ']'
             | 'R' '[' irule_number ']'
             | 'R' '[' irule_number '.' isubrule_number ']'
             | 'T' '[' irule_number ']'
             ;

exp_variable : 'l' '[' irule_number '.' isubrule_number ']'
```

```
               | 'r' '[' irule_number '.' isubrule_number ']'
               | 't' '[' irule_number ']'
               ;

irule_number : inumber
               ;

isubrule_number : inumber
                  ;

inumber        : NUMBER
               | IDENTIFIER
               ;

graph_variable : 'edge' '(' graph_type ',' pvertex ',' pvertex ')'
                 | 'cycle' '(' graph_type ')'
                 ;

pvertex          : vertex
                 | vmvertex
                 ;

graph_type     : 'ENABLE_RULE'
               | 'DISABLE'
               | 'VARIABLE_MODIFICATION'
               ;

sets           : set
               | sets ',' set
               ;

set            : '{' list '}'
               | '{' '}'
               ;
```

```
list           : set_rule_list
               ;

set_rule_list: rule_number
               | set_rule_list ',' rule_number
               ;

rule_list      : rule
               | rule_list ',' rule
               ;

rule           : rule_number
               ;

subrule_list      : subrule
                   | subrule_list ',' subrule
                  ;

subrule            : IDENTIFIER '.' subrule_number
                   | NUMBER '.' subrule_number
                   ;

compatible_set : 'COMPATIBLE_SET' '=' '(' sets ')'
                 ;

break_condition  : 'BREAK_CYCLE' '(' graph_type ')' '=' break_cond
                 | 'BREAK_CYCLE' '(' graph_type ',' cycles_list ')'
                   '=' break_cond
                 ;

break_cond       : specification
                 | simple_pascal_expression
                 ;
```

```
cycles_list      : '{' cycles '}'
                 ;

cycles           : cycle
                 | cycles ',' cycle
                 ;

cycle            : '(' vertex_list ')'
                 | '(' vmvertex_list ')'
                 ;

vertex_list    : vertex
               | vertex_list ',' vertex
               ;

vertex         : irule_number
               ;

vmvertex_list    : vmvertex
                 | vmvertex_list ',' vmvertex
                 ;

vmvertex         : irule_number '.' isubrule_number
                 ;

rule_number      : IDENTIFIER
                 ;

subrule_number   : IDENTIFIER
                 ;

variable         : IDENTIFIER
                 | graph_variable
                 | set_variable
```

```
                | exp_variable
                ;

function_name   : 'INTERSECT'
                | 'UNION'
                | 'RELATIVE_COMPLEMENT'
                | 'VALUE'
                ;

predicate_name  : 'MEMBER'
                | 'IN_CYCLE'
                | 'EQUAL'
                | 'COMPATIBLE'
                | 'MUTEX'
                ;

pascal_expression     : disjunctive_expr
                      | pascal_expression  'OR'  disjunctive_expr
                      ;

disjunctive_expr      : conjunctive_expr
                      | disjunctive_expr  'AND'  conjunctive_expr
                      ;

conjunctive_expr      : simple_expression
                      | conjunctive_expr  '='   simple_expression
                      | conjunctive_expr  '<>'  simple_expression
                      | conjunctive_expr  '<='  simple_expression
                      | conjunctive_expr  '>='  simple_expression
                      | conjunctive_expr  '<'   simple_expression
                      | conjunctive_expr  '>'   simple_expression
                      ;
```

```
simple_expression       : term1
                        | simple_expression '+' term1
                        | simple_expression '-' term1
                        ;

term1           : factor
                | term1 '*' factor
                | term1 'DIV' factor
                | term1 'MOD' factor
                ;

factor          : variable
                | NUMBER
                | '(' pascal_expression ')'
                | 'NOT' factor
                | '+' term1
                | '-' term1
                ;

simple_pascal_expression : '(' simple_exp ')'
                         ;

simple_exp : exp
           | simple_exp 'AND' exp
           ;

exp        : IDENTIFIER '=' value
           ;

value      : IDENTIFIER
           | NUMBER
           ;
```

Appendix C

The Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure control System

This real-time decision system is called the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure Control System. It is invoked in every monitor-decide cycle to diagnose the condition of the Cryogenic Hydrogen Pressure Control System and to give advice for correcting the diagnosed malfunctions. The complete **EQL** program for this malfunction procedure consists of 36 rules, 31 sensor input variables, and 32 program variables. The meaning of most of these sensor input and program variables requires specialized knowledge of the pressure control system.

Define the following sensor input variables:

v63a1a	sensor H2 P Normal.
v63a1b	sensor H2 P High.
v63a1c	sensor H2 P Low.
v63a3	sensor Press in all tks < 153 psia.
v63a5	sensor Both P and TK P of affected tk low.
v63a8	sensor Received O2 PRESS Alarm and/or S68 CRYO H2 PRES and S68 CRYO 2 PRES msg lines.
v63a11	sensor TK3 and/or TK4 the affected tk.
v63a12	sensor TK3 and TK4 depleted, QTY < 10 %.
v63a13	sensor a13.
v63a16	sensor CNTLR cb of affected tk on Pnl ML868 open.

v63a17 sensor TK3 and TK4 Htrs cycle on when press in both tks = 217-223 psia.

v63a22 sensor a22.

v63a23 sensor TK3 and/or TK4 the affected tk.

v63a26 sensor TK3 and TK4 htrs were deactivated (all htrs switches in OFF when the problem occurred).

v63a29 sensor Press in both TK3 and TK4 > 293.8 psia.

v63a31 sensor Both P and TK P of affected tk high.

v63a32 sensor MANF Ps agree with P and TK P of affected TK.

v63a34a sensor P high.

v63a34b sensor TK P high.

v63b7 sensor b7.

Define the following program variables:

v63a2 diagnosis: C/W failure.

v63a4 diagnosis: System leak. Execute ECLS SSR-1(7).

v63a6 diagnosis: Leak between affected TK and check valve. Leak cannot be isolated.

v63a7 action: Deactivate htrs in affected tk.

v63a9 recovery: Reconfigure htrs per BUS LOSS SSR.

v63a10 temporary variable.

v63a14 if true, then CNTLR cb of affected tk (TK1 and/or TKS) on Pnl 013 is open.
if false, then CNTLR cb of affected tk (TK1 and/or TK2) on Pnl 013 is closed.

v63a15 diagnosis: Possible electrical problem. Do not attempt to reset circuit breaker.

v63a18 diagnosis: P 63axduce failed low. Continue to operate TK3 and TK4 in AUTO.

v63a19 diagnosis: Possible electrical problem.

Do not attempt to reset circuit breaker.

v63a20	diagnosis: PWR failure in affected HTR CNTLR.
v63a21	action: deactivate htrs in affected tk(s).
v63a24	diagnosis: P Xducer failed low. Continue to operate TK1 and TK2 in AUTO.
v63a25	diagnosis: PWR failure in affected htr cntlr.
v63a27	diagnosis: Instrumentation failure. No action required.
v63a28	action: Operate TK1 and TK2 htrs in manual mode.
v63a30	diagnosis: Auto pressure control failure.
v63a33	diagnosis: Line blockage in tk reading high.
v63a35	diagnosis: Auto pressure control or RPC failure.
v63a36	diagnosis: Instrumentation failure.
v63a37	action: Leave affected htrs deactivated until MCC develops consumables management plan.
v63a38	diagnosis: Instrumentation failure.
v63a39	action: Activate htrs.

```
PROGRAM cryov63a;
CONST
      true   = 1;
      false  = 0;
VAR
    v63a2, v63a4, v63a6, v63a7, v63a9, v63a10, v63a14, v63a15, v63a18, v63a19, v63a20,
    v63a21, v63a24, v63a25, v63a27, v63a28, v63a30, v63a33, v63a35, v63a36, v63a37,
      v63a38, v63a39, v63a42, v63a43, v63a44, v63a47, v63a48, v63a50, v63a52, v63a53
            : BOOLEAN;

INPUTVAR
    v63a1a, v63a1b, v63a1c, v63a1, v63a3, v63a5, v63a8, v63a11, v63a12, v63a13, v63a16,
    v63a17, v63a22, v63a23, v63a26, v63a29, v63a31, v63a32, v63a34a, v63a34b, v63a34bn,
        v63a40, v63a41, v63a45, v63a46, v63a49, v63a49a, v63a49b, v63a51, v63b7, v63b8
            : BOOLEAN;

INIT
    v63a2  := false, v63a4  := false, v63a6  := false, v63a7  := false, v63a9  := false,
    v63a14 := false, v63a15 := false, v63a18 := false, v63a19 := false, v63a20 := false,
    v63a21 := false, v63a10 := false, v63a24 := false, v63a25 := false, v63a27 := false,
    v63a28 := false, v63a30 := false, v63a33 := false, v63a35 := false, v63a36 := false,
    v63a37 := false, v63a38 := false, v63a39 := false, v63a42 := false, v63a43 := false,
      v63a44 := false, v63a47 := false, v63a48 := false, v63a50 := false, v63a52 := false,
      v63a53 := false

RULES
 v63a2 := true IF (v63a1a = true)
[]v63a2 := false IF (v63a1a = false)

[]v63a4 := true IF (v63a1c = true) AND (v63a3 = true)
[]v63a4 := false IF (v63a1c = false) OR (v63a3 = false)

[]v63a6 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = true)
[]v63a6 := false IF (v63a1c = false) OR (v63a3 = true) OR (v63a5 = false)

[]v63a7 := true IF (v63a6 = true)
[]v63a7 := false IF (v63a6 = false)

[]v63a9 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
          (v63a8 = true)
[]v63a9 := false IF (v63a1c = false) OR (v63a3 = true) OR (v63a5 = true) OR
          (v63a8 = false)

[]v63a10 := true IF (v63a9 = true)
[]v63a10 := false IF (v63a9 = false)

[]v63a14 := true IF ((v63a12 = true) OR ((v63a12 = false) AND (v63a13 = true)))
```

```
[]v63a14 := false IF ((v63a12 = false) AND ((v63a12 = true) OR (v63a13 = false)))

[]v63a15 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
            (v63a8 = false) AND (v63a11 = false) AND
            (v63a12 = true) AND (v63a14 = true)
[]v63a15 := false IF (v63a1c = false) OR (v63a3 = true) OR (v63a5 = true) OR
            (v63a8 = true) OR (v63a11 = false) OR
            (v63a12 = false) OR (v63a14 = false)

[]v63a18 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
            (v63a8 = false) AND (v63a11 = true) AND
            (v63a16 = true) AND (v63a17 = true)
[]v63a18 := false IF (v63a1c = false) OR (v63a3 = true) OR (v63a5 = true) OR
            (v63a8 = true) OR (v63a11 = false) OR
            (v63a16 = false) OR (v63a17 = false)

[]v63a19 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
            (v63a8 = false) AND (v63a11 = true) AND (v63a16 = true)
[]v63a19 := false IF (v63a1c = false) OR (v63a3 = true) OR (v63a5 = true) OR
            (v63a8 = true) OR (v63a11 = false) OR (v63a16 = false)

[]v63a20 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
            (v63a8 = false) AND (v63a11 = true) AND
            (v63a16 = true) AND (v63a17 = false)
[]v63a20 := false IF (v63a1c = false) OR (v63a3 = true) OR (v63a5 = true) OR
            (v63a8 = true) OR (v63a11 = false) OR
            (v63a16 = false) OR (v63a17 = true)

[]v63a21 := true IF ((v63a19 = true) OR (v63a20 = true))
[]v63a21 := false IF ((v63a19 = false) AND (v63a20 = false))

[]v63a24 := true IF (v63a22 = true) AND (v63a14 = false) AND (v63a12 = true) AND
            (v63a11 = false) AND (v63a8 = false) AND (v63a5 = false) AND
            (v63a3 = false) AND (v63a1c = true)
[]v63a24 := false IF (v63a22 = false) OR (v63a14 = true) OR (v63a12 = false) OR
            (v63a11 = true) OR (v63a8 = true) OR (v63a5 = true) OR
            (v63a3 = true) OR (v63a1c = false)

[]v63a25 := true IF (v63a22 = false) AND (v63a14 = false) AND (v63a12 = true) AND
            (v63a11 = false) AND (v63a8 = false) AND (v63a5 = false) AND
            (v63a3 = false) AND (v63a1c = true)
[]v63a25 := false IF (v63a22 = true) OR (v63a14 = true) OR (v63a12 = false) OR
            (v63a11 = true) OR (v63a8 = true) OR (v63a5 = true) OR
            (v63a3 = true) OR (v63a1c = false)

[]v63a27 := true IF (v63a26 = true) AND (((v63a23 = true) AND (v63a1b = true)) OR
                        (v63b7 = true))
[]v63a27 := false IF (v63a26 = false) OR (((v63a23 = false) OR (v63a1b = false)) AND
                        (v63b7 = false))

[]v63a28 := true IF ((v63a25 = true) OR (v63a15 = true))
[]v63a28 := false IF ((v63a25 = false) AND (v63a15 = false))

[]v63a30 := true IF (((v63a1b = true) AND (v63a23 = true)) OR (v63b7 = true)) AND
            (v63a26 = false) AND (v63a29 = true)
[]v63a30 := false IF (((v63a1b = false) OR (v63a23 = false)) AND (v63b7 = false)) OR
            (v63a26 = true) OR (v63a29 = false)

[]v63a33 := true IF (v63a32 = false) AND (v63a31 = true) AND (v63a29 = false) AND
            (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a33 := false IF (v63a32 = true) OR (v63a31 = false) OR (v63a29 = true) OR
            (v63a26 = true) OR (v63a23 = false) OR (v63a1b = false)

[]v63a35 := true IF (v63a32 = true) AND (v63a31 = true) AND (v63a29 = true) AND
            (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a35 := false IF (v63a32 = false) OR (v63a31 = false) OR (v63a29 = false) OR
```

```
            (v63a26 = true) OR (v63a23 = false) OR (v63a1b = false)

[]v63a36 := true IF (v63a34b = true) AND (v63a31 = false) AND (v63a29 = false) AND
            (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a36 := false IF (v63a34b = false) OR (v63a31 = true) OR (v63a29 = true) OR
            (v63a26 = true) OR (v63a23 = false) OR (v63a1b = false)

[]v63a37 := true IF (v63a30 = true) AND (v63a33 = false) AND (v63a35 = false) AND
            (v63a38 = true)
[]v63a37 := false IF (v63a30 = false) OR (v63a33 = true) OR (v63a35 = true) OR
            (v63a38 = false)

[]v63a38 := true IF (v63a34a = true) AND (v63a31 = false) AND (v63a29 = false) AND
            (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a38 := false IF (v63a34a = false) OR (v63a31 = true) OR (v63a29 = true) OR
            (v63a26 = true) OR (v63a23 = false) OR (v63a1b = false)

[]v63a39 := true IF (v63a36 = true)
[]v63a39 := false IF (v63a36 = false)

[]v63a42 := true IF (v63a41 = true) AND (v63a40 = true) AND (v63a23 = false) AND
            (v63a1b = true)
[]v63a42 := false IF (v63a41 = false) OR (v63a40 = false) OR (v63a23 = true) OR
            (v63a1b = false)

[]v63a43 := true IF (v63a41 = false) AND (v63a40 = true) AND (v63a23 = false) AND
            (v63a1b = true)
[]v63a43 := false IF (v63a41 = true) OR (v63a40 = false) OR (v63a23 = true) OR
            (v63a1b = false)

[]v63a44 := true IF (v63a42 = true) OR (v63a47 = true)
[]v63a44 := false IF (v63a42 = false) AND (v63a47 = false)

[]v63a47 := true IF (v63a46 = true) AND (v63a45 = true) AND (v63a40 = false) AND
```

```
            (v63a23 = false) AND (v63a1b = true)

[]v63a47 := true IF (v63a46 = true) AND (v63a45 = true) AND (v63b8 = true)
[]v63a47 := false IF ((v63a46 = false) OR (v63a45 = false) OR (v63a40 = true) OR
            (v63a23 = true) OR (v63a1b = false))  AND
                ((v63a46 = false) OR (v63a45 = false) OR (v63b8 = false))

[]v63a48 := true IF (v63a46 = false) AND (v63a45 = true) AND (v63a40 = false) AND
            (v63a23 = false) AND (v63a1b = true)
[]v63a48 := true IF (v63a46 = false) AND (v63a45 = true) AND (v63b8 = true)
[]v63a48 := false IF ((v63a46 = true) OR (v63a45 = false) OR (v63a40 = true) OR
            (v63a23 = true) OR (v63a1b = false))  AND
                ((v63a46 = true) OR (v63a45 = false) OR (v63b8 = false))

[]v63a50 := true IF (v63a49b = true) AND (v63a45 = false) AND (v63a40 = false) AND
            (v63a23 = false) AND (v63a1b = true)
[]v63a50 := true IF (v63a49b = true) AND (v63a45 = false) AND (v63b8 = true)
[]v63a50 := false IF ((v63a49b = false) OR (v63a45 = true) OR (v63a40 = true) OR
            (v63a23 = true) OR (v63a1b = false))  AND
                ((v63a49b = false) OR (v63a45 = true) OR (v63b8 = false))

[]v63a51 := true IF (v63a50 = true)
[]v63a51 := false IF (v63a50 = false)

[]v63a52 := true IF (v63a49a = true) AND (v63a45 = false) AND (v63a40 = false) AND
            (v63a23 = false) AND (v63a1b = true)
[]v63a52 := true IF (v63a49a = true) AND (v63a45 = false) AND (v63b8 = true)
[]v63a52 := false IF ((v63a49a = false) OR (v63a45 = true) OR (v63a40 = true) OR
            (v63a23 = true) OR (v63a1b = false))  AND
                ((v63a49a = false) OR (v63a45 = true) OR (v63b8 = false))

[]v63a53 := true IF (v63a52 = true)
[]v63a53 := false IF (v63a52 = false)
```

```
TRACE
    v63a2, v63a4, v63a6, v63a7, v63a9, v63a10, v63a14, v63a15, v63a18, v63a19, v63a20,
    v63a21, v63a24, v63a25, v63a27, v63a28, v63a30, v63a33, v63a35, v63a36, v63a37,
      v63a38, v63a39, v63a42, v63a43, v63a44, v63a47, v63a48, v63a50, v63a52, v63a53

PRINT
    v63a2, v63a4, v63a6, v63a7, v63a9, v63a10, v63a14, v63a15, v63a18, v63a19, v63a20,
    v63a21, v63a24, v63a25, v63a27, v63a28, v63a30, v63a33, v63a35, v63a36, v63a37,
    v63a38, v63a39, v63a42, v63a43, v63a44, v63a47, v63a48, v63a50, v63a52, v63a53

END.
```

Appendix D

The MITRE Integrated Status Assessment (ISA) Expert System

```
PROGRAM isa;
CONST
        true    = 1;
        false   = 0;

        nominal      = 0;
        suspect= 1;
        failed  = 2;

        direct  = 1;

        off     = 0;
        on      = 1;

        bad     = 0;
        good    = 1;

        backup_failed = 2; (* try to use backup, but backup device failed *)
        no_backup     = 3; (* try to use backup, but no backup is found *)
VAR
        problem1, problem2, problem3, problem4,   (* true, false *)
        sensor1, sensor2, sensor3, sensor4,   (* good, bad *)
        reconfig1, reconfig2,
        reconfig3, reconfig4,                 (* true, false *)
        switch_backup1, switch_backup2,    switch_backup3,
        switch_backup4,              (* true, false, switch_failed, no_backup *)
        find_bad_things                    (* true, false *)
        : INTEGER;
INPUTVAR
        state1, state2, state3, state4,          (* nominal, suspect, failed *)
        mode1, mode2, mode3, mode4,              (* on, off *)
        config1, config2, config3, config4,      (* good, bad *)
        backup1, backup2, backup3, backup4,      (* true, false *)
        backup1_state, backup2_state,
        backup3_state, backup4_state,            (* nominal, suspect, failed *)
        rel1_state, rel2_state, rel3_state,      (* nominal, suspect, failed *)
        rel1_type, rel2_type, rel3_type,         (* direct *)
        rel1_mode, rel2_mode, rel3_mode          (* on, off *)
        : INTEGER;
INIT
        problem1 := false, problem2 := false,
        problem3 := false, problem4 := false,
        sensor1 := good, sensor2 := good,
        sensor3 := good, sensor4 := good,
        reconfig1 := false, reconfig2 := false,
        reconfig3 := false, reconfig4 := false,
        switch_backup1 := false, switch_backup2 := false,
        switch_backup3 := false, switch_backup4 := false,
        find_bad_things := false

RULES
(* 1 *)
        rel1_state := suspect IF (state1 = suspect OR state1 = failed) AND
                                 rel1_state <> suspect AND
                                 rel1_mode = on AND rel1_type = direct
(* 2 *)
[]      rel2_state := suspect IF (state2 = suspect OR state2 = failed) AND
```

```
                                rel2_state <> suspect AND
                                rel2_mode = on AND rel2_type = direct

(* 3 *)
[]      rel3_state := suspect IF (state3 = suspect OR state3 = failed) AND
                                rel3_state <> suspect AND
                                rel3_mode = on AND rel3_type = direct

(* 4 *)
[]      find_bad_things := true IF (state1 = suspect OR state1 = failed) AND
                                NOT (rel1_state = nominal AND
                                rel1_mode = on AND rel1_type = direct)
(* 5 *)
[]      find_bad_things := true IF (state2 = suspect OR state2 = failed) AND
                                NOT (rel2_state = nominal AND
                                rel2_mode = on AND rel2_type = direct)
(* 6 *)
[]      find_bad_things := true IF (state3 = suspect OR state3 = failed) AND
                                NOT (rel3_state = nominal AND
                                rel3_mode = on AND rel3_type = direct) AND
                                NOT (rel1_state = suspect AND
                                rel1_mode = on AND rel1_type = direct) AND
                                NOT (rel2_state = suspect AND
                                rel2_mode = on AND rel2_type = direct)
(* 7 *)
[]      find_bad_things := true IF (state4 = suspect OR state4 = failed) AND
                                NOT (rel2_state = suspect AND
                                rel2_mode = on AND rel2_type = direct)

(* 8 *)
[]      state1 := failed IF find_bad_things = true AND
                        state1 = suspect
```

```
(* 9 *)
[]      state2 := failed IF find_bad_things = true AND
                            state2 = suspect
(* 10 *)
[]      state3 := failed IF find_bad_things = true AND
                            state3 = suspect AND
                            NOT (rel1_state = suspect AND rel1_mode = on AND
                                rel1_type = direct) AND
                            NOT (rel2_state = suspect AND rel2_mode = on AND
                                rel2_type = direct)
(* 11 *)
[]      state4 := failed IF find_bad_things = true AND
                            state4 = suspect AND
                            NOT (rel3_state = suspect AND rel3_mode = on AND
                                rel3_type = direct)

(* 12 *)
[]      problem1 := true IF state1 = failed AND mode1 <> off AND config1 = good
(* 13 *)
[]      problem2 := true IF state2 = failed AND mode2 <> off AND config2 = good
(* 14 *)
[]      problem3 := true IF state3 = failed AND mode3 <> off AND config3 = good
(* 15 *)
[]      problem4 := true IF state4 = failed AND mode4 <> off AND config4 = good

(* 16 *)
[]      state1 := nominal ! reconfig1 := true
                          IF state1 = failed AND mode1 <> off AND config1 = bad
(* 17 *)
[]      state2 := nominal ! reconfig2 := true
                          IF state2 = failed AND mode2 <> off AND config2 = bad
```

```
(* 18 *)
[]      state3 := nominal ! reconfig3 := true
                        IF state3 = failed AND mode3 <> off AND config3 = bad
(* 19 *)
[]      state4 := nominal ! reconfig4 := true
                        IF state4 = failed AND mode4 <> off AND config4 = bad

(* 20 *)
[]      switch_backup1 := true IF problem1 = true
                        AND backup1 = true AND backup1_state = nominal
                        AND state1 = failed AND mode1 = on
(* 21 *)
[]      switch_backup2 := true IF problem2 = true
                        AND backup2 = true AND backup2_state = nominal
                        AND state2 = failed AND mode2 = on
(* 22 *)
[]      switch_backup3 := true IF problem3 = true
                        AND backup3 = true AND backup3_state = nominal
                        AND state3 = failed AND mode3 = on
(* 23 *)
[]      switch_backup4 := true IF problem4 = true
                        AND backup4 <> 0 AND backup4_state = nominal
                        AND state4 = failed AND mode4 = on

(* 24 *)
[]      switch_backup1 := backup_failed IF problem1 = true
                        AND backup1 = true AND backup1_state = failed
                        AND state1 = failed AND mode1 = on
(* 25 *)
[]      switch_backup2 := backup_failed IF problem2 = true
                        AND backup2 = true AND backup2_state = failed
```

```
                    AND state2 = failed AND mode2 = on
(* 26 *)
[]      switch_backup3 := backup_failed IF problem3 = true
                    AND backup3 = true AND backup3_state = failed
                    AND state3 = failed AND mode3 = on
(* 27 *)
[]      switch_backup4 := backup_failed IF problem4 = true
                    AND backup4 = true AND backup4_state = failed
                    AND state4 = failed AND mode4 = on

(* 28 *)
[]      switch_backup1 := no_backup IF problem1 = true AND state1 = failed
                    AND backup1 = false
(* 29 *)
[]      switch_backup2 := no_backup IF problem2 = true AND state2 = failed
                    AND backup2 = false
(* 30 *)
[]      switch_backup3 := no_backup IF problem3 = true AND state3 = failed
                    AND backup3 = false
(* 31 *)
[]      switch_backup4 := no_backup IF problem4 = true AND state4 = failed
                    AND backup4 = false

(* 32 *)
[]      sensor1 := bad ! state1 := nominal IF state1 = suspect AND
                    rel1_mode = on AND rel1_type = direct AND
                    state3 = nominal AND rel3_mode = on AND
                    rel3_type = direct AND state4 = nominal AND
                    find_bad_things = true
(* 33 *)
[]      sensor2 := bad ! state2 := nominal IF state2 = suspect AND
```

```
                        rel1_mode = on AND rel1_type = direct AND
                        state3 = nominal AND rel3_mode = on AND
                        rel3_type = direct AND state4 = nominal AND
                        find_bad_things = true

(* 34 *)
[]      sensor3 := bad ! state3 := suspect IF state1 = suspect AND
                        rel1_mode = on AND rel1_type = direct AND
                        state3 = nominal AND rel3_mode = on AND
                        rel3_type = direct AND state4 = suspect AND
                        find_bad_things = true
(* 35 *)
[]      sensor3 := bad ! state3 := suspect IF state2 = suspect AND
                        rel2_mode = on AND rel2_type = direct AND
                        state3 = nominal AND rel3_mode = on AND
                        rel3_type = direct AND state4 = suspect AND
                        find_bad_things = true
        TRACE
        PRINT state1,   state2,   state3,   state4,
            problem1, problem2, problem3, problem4,
            sensor1,  sensor2,  sensor3,  sensor4,
            reconfig1,  reconfig2,  reconfig3,  reconfig4,
            switch_backup1, switch_backup2,
            switch_backup3, switch_backup4,
            rel1_state, rel2_state, rel3_state
END.
```

Appendix E

The MITRE-NASA Fuel Cell Expert (FCE) System

```
PROGRAM fuel_cell_expert_system;
CONST

(*------------------------------------------------------------------------*)
(*      Names of modules of rules, initial digit is 1                    *)
(*------------------------------------------------------------------------*)
        start_work_rule_base = 100;
        fc_exit_t7_3d        = 101;
        fc_stack_t7_1b                 = 102;
        cool_pump7_1a                  = 103;
        fc_amps7_3c          = 104;
        fc_delta_v7_4_1_4    = 105;
        fc_ech7_3_1_2                  = 106;
        fc_purge7_2          = 107;
        fc_cool_p7_3e        = 108;
        fc_h20_vlvt7_3g                = 109;
        prd_h20_lnt7_3f                = 110;
        fc_reacs7_1_1_1                = 111;
        fc_general           = 112;
        fc_volts7_3b         = 113;
        bus_tie_rules        = 114;

(*------------------------------------------------------------------------*)
(*     Constants                                                          *)
(*------------------------------------------------------------------------*)
```

```
false           = 0;
true            = 1;
off             = 0;
on              = 1;
start_value     = 200;
clsd_confirmed          = 201;
clsd            = 202;
high            = 203;
high1           = 204;
high2           = 205;
unverified_high = 206;
high_confirmed          = 207;
high_start      = 208;
disconnected    = 209;
low             = 210;
low1            = 211;
failed_low      = 212;
ok              = 213;
ecu_pwr_loss    = 214;
restarted       = 215;
cool_pump_loss          = 216;
internal_ld     = 217;
disconn_v_low           = 218;
inh             = 219;
a_auto          = 220;
b_auto          = 221;
start           = 222;
sw_to_b                 = 223;
decr            = 224;
fc_failed_dv_hi         = 225;
```

```
tied            = 226;
untied          = 227;
tied_short      = 228;
untied_short    = 229;
safed           = 230;
shutdn          = 231;
cool_lp_prob    = 232;
fc_or_stk_t_fail= 233;
hi_snsr_fail    = 234;
snsr_failed     = 235;
ok_su_htr_inh   = 236;
ech_failed_on   = 237;
failed_on       = 238;
overloaded1     = 239;
overloaded2     = 240;
reconnected     = 241;
internal_short  = 242;
disconn_stop    = 243;
lightly_loaded  = 244;
shorted         = 245;
pripl_short_isolated = 246;
degraded        = 247;
degraded_bad    = 248;
perf_low        = 249;
orb_freon_lp_prob = 250;
ece_pwr_loss    = 251;
low2            = 252;
low3            = 253;
incr            = 254;
stop            = 255;
```

```
        open            = 256;
        gpc             = 257;
        low_confirmed          = 258;

VAR
        work_rule_base, o2_reac_vlv, status, status1, status2, status3,
        fc_mn_dv, load_status, stk_t_status, cool_loop_status, prd_h20_lnt,
        h20_rlf_timer, ss1_dv, ss2_dv, ss3_dv, ph,
        bus_tie_statusa, bus_tie_statusb, bus_tie_statusc,
        busa_tie_status, busb_tie_status, busc_tie_status,
        stack_t, stack_t2, stack_t3, fc_mn_conn,
        message1, message2, message3, message4, message5,
        message6, message7, message8, message9, message10,
        message11, message12, message13, message14, message15,
        message16, message17, message18, message19, message20,
        message21, message22, message23, message24, message25,
        message26, message27, message28, message29, message30 : INTEGER;

INPUTVAR
        tce, last_tce,
        cool_rtn_t, last_cool_rtn_t,
        koh_in_conc, last_koh_in_conc,
        koh_out_conc, last_koh_out_conc,
        last_prd_h20_lnt,

        cool_rtn_t1, last_cool_rtn_t1,
        cool_rtn_t2, last_cool_rtn_t2,
        cool_rtn_t3, last_cool_rtn_t3,
        last_fc_mn_dv,
        mn_bus, last_mn_bus,
        last_fc_mn_conn,
last_status,
su_atr, last_su_atr,
voltage, last_voltage,
voltage1, last_voltage1,
voltage2, last_voltage2,
voltage3, last_voltage3,
last_stk_t_status,
stk_t_status2, last_stk_t_status2,
stk_t_status3, last_stk_t_status3,
last_stack_t,
delta_v, last_delta_v,
stk_t_rate, last_stk_t_rate,
stk_t_disconn2, last_stk_t_disconn2,
stk_t_disconn3, last_stk_t_disconn3,
amps1, last_amps1,
amps2, last_amps2,
amps3, last_amps3,

cntlr_pwr, last_cntlr_pwr,
cool_pump_dp, last_cool_pump_dp,
fc_rdy_for_ld, last_fc_rdy_for_ld,

last_load_status,
last_amps,

last_ss1_dv,
last_ss2_dv,
last_ss3_dv,
ss1_dv_rate, last_ss1_dv_rate,
ss2_dv_rate, last_ss2_dv_rate,
ss3_dv_rate, last_ss3_dv_rate,
end_cell_htr1, last_end_cell_htr1,
end_cell_htr2, last_end_cell_htr2,
fc_ess_conn, last_fc_ess_conn,

purge_vlv_sw_pos, last_purge_vlv_sw_pos,
auto_purge_seq, last_auto_purge_seq,
h2_flow_rate, last_h2_flow_rate,
o2_flow_rate, last_o2_flow_rate,
purge_htr_sw_pos, last_purge_htr_sw_pos,
fc_purge_alarm, last_fc_purge_alarm,
fc_purge_seq_alarm, last_fc_purge_seq_alarm,
fc_purge_t_alarm, last_fc_purge_t_alarm,

cool_p, last_cool_p,
cool_p_rate, last_cool_p_rate,
last_o2_reac_vlv,
h2_reac_vlv, last_h2_reac_vlv,
bus_tie_status, last_bus_tie_status,
cool_ph20_p, last_cool_ph20_p,

h20_rlf_vlv_t, last_h20_rlf_vlv_t,
h20_rlf_t_msg, last_h20_rlf_t_msg,
h20_rlf_sw_pos, last_h20_rlf_sw_pos,
last_h20_rlf_timer,
h20_vlv_t_rate, last_h20_vlv_t_rate,

prt_h20_lnt, last_prt_h20_lnt,          (* 10 *)
prd_h20_sw_pos, last_prd_h20_sw_pos,

fc_pripl_conn, last_fc_pripl_conn,    (* 13 *)
mn_pripl_conn, last_mn_pripl_conn,
mna_voltage, last_mna_voltage,
          mnb_voltage, last_mnb_voltage,
          mnc_voltage, last_mnc_voltage,

          mn_id,                          (* 19 *)

          su_htr,                         (* 20 *)

          stk_t_status1,                  (* 26 *)
          stk_t_disconn1,

          amps,                           (* 27 *)

          busa_tie, busb_tie, busc_tie,

          mn_voltage, fc_id               (* 64 *)
          : INTEGER;

INIT
          work_rule_base        := start_work_rule_base,
        o2_reac_vlv       := start_value,
        status            := start_value,
        status1           := start_value,
        status2           := start_value,
        status3           := start_value,
        fc_mn_dv          := start_value,
        load_status       := start_value,
        stk_t_status      := start_value,
        cool_loop_status:= start_value,
        prd_h20_lnt       := start_value,
        h20_rlf_timer     := start_value,
        ss1_dv            := start_value,
        ss2_dv            := start_value,
        ss3_dv            := start_value,
     ph                   := start_value,
     bus_tie_statusa := start_value,
     bus_tie_statusb := start_value,
     bus_tie_statusc := start_value,
     busa_tie_status := start_value,
     busb_tie_status := start_value,
     busc_tie_status := start_value,
     stack_t              := start_value,
     stack_t2  := start_value,
     stack_t3  := start_value,
     fc_mn_conn        := start_value,
        message1 := false, message2 := false, message3 := false,
        message4 := false, message5 := false, message6 := false,
        message7 := false, message8 := false, message9 := false,
        message10 := false, message11 := false, message12 := false,
        message13 := false, message14 := false, message15 := false,
        message16 := false, message17 := false, message18 := false,
        message19 := false, message20 := false, message21 := false,
        message22 := false, message23 := false, message24 := false,
        message25 := false, message26 := false, message27 := false,
        message28 := false, message29 := false, message30 := false

RULES

(*------------------------------------------------------------------------*)
(*      Metarules                                                          *)
(*------------------------------------------------------------------------*)

(* 1 *)
          work_rule_base := fc_exit_t7_3d
          IF NOT (tce = last_tce AND cool_rtn_t = last_cool_rtn_t AND
                koh_in_conc = last_koh_in_conc AND
                koh_out_conc = last_koh_out_conc AND
                prd_h20_lnt = last_prd_h20_lnt)
(* 2 *)
        [] work_rule_base := fc_stack_t7_1b
          IF NOT (cool_rtn_t1 = last_cool_rtn_t1 AND
                cool_rtn_t2 = last_cool_rtn_t2 AND
                cool_rtn_t3 = last_cool_rtn_t3 AND
                fc_mn_dv = last_fc_mn_dv AND
                mn_bus = last_mn_bus AND
                fc_mn_conn = last_fc_mn_conn AND
                status = last_status AND
                su_atr = last_su_atr AND
                voltage = last_voltage AND
                voltage1 = last_voltage1 AND
                voltage2 = last_voltage2 AND
                voltage3 = last_voltage3 AND
                stk_t_status = last_stk_t_status AND
                stk_t_status2 = last_stk_t_status2 AND
                stk_t_status3 = last_stk_t_status3 AND
                stack_t = last_stack_t AND
                delta_v = last_delta_v AND
                stk_t_rate = last_stk_t_rate AND
                cool_rtn_t = last_cool_rtn_t AND
                stk_t_disconn2 = last_stk_t_disconn2 AND
                stk_t_disconn3 = last_stk_t_disconn3 AND
                amps1 = last_amps1 AND
                amps2 = last_amps2 AND
                amps3 = last_amps3)
(* 3 *)
```

```
        [] work_rule_base := cool_pump7_1a
          IF NOT (cntlr_pwr = last_cntlr_pwr AND
                  cool_pump_dp = last_cool_pump_dp AND
                  fc_rdy_for_ld = last_fc_rdy_for_ld AND
                  status = last_status AND
                  tce = last_tce)
(* 4 *)
        [] work_rule_base := fc_amps7_3c
          IF NOT (status = last_status AND
                  mn_bus = last_mn_bus AND
                  load_status = last_load_status AND
                  voltage = last_voltage AND
                  amps = last_amps AND
                  fc_mn_conn = last_fc_mn_conn)
(* 5 *)
        [] work_rule_base := fc_delta_v7_4_1_4
          IF NOT (ss1_dv = last_ss1_dv AND
                  ss2_dv = last_ss2_dv AND
                  ss3_dv = last_ss3_dv AND
                  ss1_dv_rate = last_ss1_dv_rate AND
                  ss2_dv_rate = last_ss2_dv_rate AND
                  ss3_dv_rate = last_ss3_dv_rate)
(* 6 *)
        [] work_rule_base := fc_ech7_3_1_2
          IF NOT (fc_rdy_for_ld = last_fc_rdy_for_ld AND
                  end_cell_htr1 = last_end_cell_htr1 AND
                  end_cell_htr2 = last_end_cell_htr2 AND
                  fc_mn_conn = last_fc_mn_conn AND
                  fc_ess_conn = last_fc_ess_conn)
(* 7 *)
```

```
        [] work_rule_base := fc_purge7_2
          IF NOT (purge_vlv_sw_pos = last_purge_vlv_sw_pos AND
                  auto_purge_seq = last_auto_purge_seq AND
                  h2_flow_rate = last_h2_flow_rate AND
                  o2_flow_rate = last_o2_flow_rate AND
                  purge_htr_sw_pos = last_purge_htr_sw_pos AND
                  fc_purge_alarm = last_fc_purge_alarm AND
                  fc_purge_seq_alarm = last_fc_purge_seq_alarm AND
                  fc_purge_t_alarm = last_fc_purge_t_alarm)
(* 8 *)
        [] work_rule_base := fc_cool_p7_3e
          IF NOT (cool_p = last_cool_p AND
                  cool_p_rate = last_cool_p_rate AND
                  h2_flow_rate = last_h2_flow_rate AND
                  o2_flow_rate = last_o2_flow_rate AND
                  cool_pump_dp = last_cool_pump_dp AND
                  purge_vlv_sw_pos = last_purge_vlv_sw_pos AND
                  auto_purge_seq = last_auto_purge_seq AND
                  o2_reac_vlv = last_o2_reac_vlv AND
                  h2_reac_vlv = last_h2_reac_vlv AND
                  fc_mn_conn = last_fc_mn_conn AND
                  bus_tie_status = last_bus_tie_status AND
                  delta_v = last_delta_v AND
                  status = last_status AND
                  cool_ph20_p = last_cool_ph20_p)
(* 9 *)
        [] work_rule_base := fc_h20_vlvt7_3g
          IF NOT (h20_rlf_vlv_t = last_h20_rlf_vlv_t AND
                  h20_rlf_t_msg = last_h20_rlf_t_msg AND
                  h20_rlf_sw_pos = last_h20_rlf_sw_pos AND
```

```
                h20_rlf_timer = last_h20_rlf_timer AND
                h20_vlv_t_rate = last_h20_vlv_t_rate)
(* 10 *)
        [] work_rule_base := prd_h20_lnt7_3f
          IF NOT (prt_h20_lnt = last_prt_h20_lnt AND
                prd_h20_sw_pos = last_prd_h20_sw_pos AND
                fc_mn_conn = last_fc_mn_conn AND
                amps = last_amps)
(* 11 *)
        [] work_rule_base := fc_reacs7_1_1_1
          IF NOT (fc_rdy_for_ld = last_fc_rdy_for_ld AND
                o2_reac_vlv = last_o2_reac_vlv AND
                h2_reac_vlv = last_h2_reac_vlv AND
                o2_flow_rate = last_o2_flow_rate AND
                h2_flow_rate = last_h2_flow_rate AND
                cool_p = last_cool_p)
(* 12 *)
        [] work_rule_base := fc_general
          IF NOT (cool_p = last_cool_p AND
                cool_pump_dp = last_cool_pump_dp AND
                h2_reac_vlv = last_h2_reac_vlv AND
                o2_reac_vlv = last_o2_reac_vlv AND
                fc_rdy_for_ld = last_fc_rdy_for_ld AND
                fc_mn_conn = last_fc_mn_conn AND
                fc_ess_conn = last_fc_ess_conn AND
                status = last_status AND
                cntlr_pwr = last_cntlr_pwr)
(* 13 *)
        [] work_rule_base := fc_volts7_3b
          IF NOT (status = last_status AND
                    delta_v = last_delta_v AND
                    fc_mn_conn = last_fc_mn_conn AND
                    fc_pripl_conn = last_fc_pripl_conn AND
                    mn_pripl_conn = last_mn_pripl_conn AND
                    bus_tie_status = last_bus_tie_status AND
                    amps = last_amps AND
                    voltage = last_voltage AND
                    load_status = last_load_status AND
                    cntlr_pwr = last_cntlr_pwr AND
                    mna_voltage = last_mna_voltage AND
                    mnb_voltage = last_mnb_voltage AND
                    mnc_voltage = last_mnc_voltage)

(*-----------------------------------------------------------------------*)
(*      rule class  fc_reacs7_1_1_1                                     *)
(*-----------------------------------------------------------------------*)

(* 14 *)
        [] o2_reac_vlv := clsd_confirmed (* *)
                IF fc_rdy_for_ld = on AND
                  (o2_reac_vlv = clsd OR h2_reac_vlv = clsd) AND
                  (o2_flow_rate = low1 OR o2_flow_rate = low1 OR cool_p = low1)
                  AND work_rule_base = fc_reacs7_1_1_1

(*-----------------------------------------------------------------------*)
(*      rule class  cool_pump7_1a                                        *)
(*-----------------------------------------------------------------------*)

(* 15 *)
        [] status := ecu_pwr_loss (* *)
                IF cntlr_pwr = off AND
```

```
                    cool_pump_dp = low AND
                    fc_rdy_for_ld = off AND
                    status = ok
                    AND work_rule_base = cool_pump7_1a
(* 16 *)
        [] status := cool_pump_loss
                  IF cntlr_pwr = on AND
                    cool_pump_dp = low AND
                    tce = high
                    AND work_rule_base = cool_pump7_1a
(* 17 *)
        [] status := restarted
                  IF (status = ece_pwr_loss OR status = cool_pump_loss) AND
                    cntlr_pwr = on AND
                    cool_pump_dp = ok AND
                    fc_rdy_for_ld = on AND
                    tce = ok
                    AND work_rule_base = cool_pump7_1a

(*------------------------------------------------------------------------*)
(*      rule class  fc_stack_t7_1b                                          *)
(*------------------------------------------------------------------------*)

(* 18 *)
        [] status1 := orb_freon_lp_prob !
          status2 := orb_freon_lp_prob !
          status3 := orb_freon_lp_prob
                  IF cool_rtn_t1 = high AND
                    cool_rtn_t2 = high AND
                    cool_rtn_t3 = high
                    AND work_rule_base = fc_stack_t7_1b
```

```
(* 19 *)
        [] fc_mn_dv := high
                IF fc_mn_dv = high AND
                  mn_bus = mn_id AND
                  fc_mn_conn = on
                  AND work_rule_base = fc_stack_t7_1b
(* 20 *)
        [] load_status := internal_ld
                IF status = disconn_v_low AND
                  su_htr = inh AND
                  (voltage = ok OR voltage = low1)
                  AND work_rule_base = fc_stack_t7_1b
(* 21 *)
        [] status := disconn_v_low
                IF status = disconnected AND
                  stk_t_status = high AND
                  voltage <> high1 AND
                  voltage <> high2
                  AND work_rule_base = fc_stack_t7_1b
(* 22 *)
        [] load_status := internal_ld
                IF stack_t = high AND
                  delta_v = low1
                  AND work_rule_base = fc_stack_t7_1b
(* 23 *)
        [] stk_t_status := high
                IF stack_t = high1 AND
                  cool_rtn_t = ok AND
                  stk_t_rate = ok
                  AND work_rule_base = fc_stack_t7_1b
```

```
(* 24 *)
        [] stk_t_status := low
                IF stack_t = low1 AND
                  cool_rtn_t = ok AND
                  stk_t_rate = ok
                  AND work_rule_base = fc_stack_t7_1b
(* 25 *)
        [] cool_loop_status := failed_low
                IF stack_t = low1 AND
                  (delta_v = low OR delta_v = low1)
                  AND work_rule_base = fc_stack_t7_1b
(* 26 *)
        [] status1 := cool_lp_prob
                IF stk_t_status1 = high_start AND
                  stk_t_disconn1 = ok
                  AND work_rule_base = fc_stack_t7_1b

(*-----------------------------------------------------------------------*)
(*      rule class  fc_cool_p7_3e                                          *)
(*-----------------------------------------------------------------------*)

(* 27 *)
        [] message1 := true
                IF cool_p = high1 AND
                  (purge_vlv_sw_pos = open OR auto_purge_seq = on)
                  AND work_rule_base = fc_cool_p7_3e
(* 28 *)  (* Missing condition *)
        [] message2 := true
                IF
                  (h2_flow_rate = high1 OR o2_flow_rate = high1)
                  AND work_rule_base = fc_cool_p7_3e
```

```
(* 29 *)
        [] message3 := true
                IF cool_p = low1 AND
                  cool_p_rate = decr AND
                  h2_reac_vlv = open AND
                  o2_reac_vlv = open AND
                  fc_mn_conn = on
                  AND work_rule_base = fc_cool_p7_3e
(* 30 *)
        [] message4 := true
                IF cool_p = low1 AND
                  h2_reac_vlv = open AND
                  o2_reac_vlv = open AND
                  fc_mn_conn = on AND
                  mn_bus = mn_id AND
                  bus_tie_status = untied
                  AND work_rule_base = fc_cool_p7_3e
(* 31 *)
        [] message5 := true
                IF cool_p_rate = ok AND
                  cool_p = low2 AND
                  delta_v = ok AND
                  fc_mn_conn = on AND
                  h2_reac_vlv = open AND
                  o2_reac_vlv = open
                  AND work_rule_base = fc_cool_p7_3e
(* 32 *)
        [] message6 := true
                IF cool_p = low1 AND
                  status = disconnected AND
```

```
                    cool_p_rate = decr
                    AND work_rule_base = fc_cool_p7_3e
(* 33 *)
        [] message7 := true
                  IF cool_p_rate = ok AND
                    cool_p = low1 AND
                    h2_reac_vlv = open AND
                    o2_reac_vlv = open AND
                    cool_ph20_p = low
                    AND work_rule_base = fc_cool_p7_3e
(* 34 *)
        [] message8 := true
                  IF cool_p = low1 AND
                    status = disconnected AND
                    cool_p_rate = incr
                    AND work_rule_base = fc_cool_p7_3e
(* 35 *)
        [] message9 := true
                  IF cool_p_rate = ok AND
                    delta_v = low AND
                    h2_reac_vlv = open AND
                    o2_reac_vlv = open
                    AND work_rule_base = fc_cool_p7_3e
(* 36 *)
        [] message10 := true
                  IF cool_p_rate = ok AND
                    cool_p = low1 AND
                    cool_ph20_p = ok AND
                    delta_v = ok AND
                    h2_reac_vlv = open AND
                    mn_bus = mn_id AND
                    bus_tie_status = tied AND
                    fc_mn_conn = on
                    AND work_rule_base = fc_cool_p7_3e
(* 37 *)
        [] message11 := true
                IF cool_p = low1 AND
                    status = disconnected AND
                    cool_p_rate = ok
                    AND work_rule_base = fc_cool_p7_3e
(* 38 *)
        [] message12 := true
                IF cool_p_rate = ok AND
                    cool_p = low1 AND
                    cool_ph20_p = ok AND
                    delta_v = low AND
                    h2_reac_vlv = open AND
                    o2_reac_vlv = open
                    AND work_rule_base = fc_cool_p7_3e

(*------------------------------------------------------------------------*)
(*      rule class  prd_h20_lnt7_3f                                        *)
(*------------------------------------------------------------------------*)

(* 39 *)
        [] message13 := true
                IF prd_h20_lnt = high1 AND
                    prd_h20_sw_pos = off
                    AND work_rule_base = prd_h20_lnt7_3f
(* 40 *)
```

```
        [] message14 := true
                IF prd_h20_lnt = low1 AND
                  fc_mn_conn = on AND
                  (amps = ok OR amps = high1)
                  AND work_rule_base = prd_h20_lnt7_3f
(* 41-27 *)
        [] prd_h20_lnt := low1
                IF prd_h20_lnt = low1 AND
                  (fc_mn_conn = off OR amps = low2 OR amps = low3) AND
                  (prd_h20_sw_pos = a_auto OR prd_h20_sw_pos = b_auto)
                  AND work_rule_base = prd_h20_lnt7_3f
(* 42-28 *)
        [] prd_h20_lnt := low1
                IF prd_h20_lnt = low1 AND
                  (fc_mn_conn = off OR amps = low2 OR amps = low3) AND
                  prd_h20_sw_pos = off
                  AND work_rule_base = prd_h20_lnt7_3f
(* 43-29 *)
        [] prd_h20_lnt := high1
                IF prd_h20_lnt = high1 AND
                  (prd_h20_sw_pos = a_auto OR prd_h20_sw_pos = b_auto)
                  AND work_rule_base = prd_h20_lnt7_3f

(*------------------------------------------------------------------------*)
(*      rule class  fc_h20_vlvt7_3g                              *)
(*------------------------------------------------------------------------*)

(* 44-30 *)
        [] h20_rlf_timer := start
                IF h20_rlf_vlv_t = high AND
                  h20_rlf_t_msg = sw_to_b AND
```

```
                    h20_rlf_sw_pos = b_auto
                    AND work_rule_base = fc_h20_vlvt7_3g
(* 45-31 *)
        [] h20_rlf_timer := stop
                IF h20_rlf_timer = start AND
                    h20_vlv_t_rate <> decr
                    AND work_rule_base = fc_h20_vlvt7_3g

(*----------------------------------------------------------------------*)
(*      rule class  fc_delta_v7_4_1_4                                  *)
(*----------------------------------------------------------------------*)

(* 46-32 *)
        [] ss1_dv := unverified_high
                IF ss1_dv = high AND
                    ss1_dv_rate = ok AND
                    ph = ok
                    AND work_rule_base = fc_delta_v7_4_1_4
(* 47-33 *)
        [] ss2_dv := unverified_high
                IF ss2_dv = high AND
                    ss2_dv_rate = ok AND
                    ph = ok
                    AND work_rule_base = fc_delta_v7_4_1_4
(* 48-34 *)
        [] ss3_dv := unverified_high
                IF ss3_dv = high AND
                    ss3_dv_rate = ok AND
                    ph = ok
                    AND work_rule_base = fc_delta_v7_4_1_4
(* 49-35 *)
```

```
        [] status := fc_failed_dv_hi
                IF ph = ok AND
                  ((ss1_dv = high AND ss1_dv_rate = incr) OR
                   (ss2_dv = high AND ss2_dv_rate = incr) OR
                   (ss3_dv = high AND ss3_dv_rate = incr))
                  AND work_rule_base = fc_delta_v7_4_1_4
(* 50-36 *)
        [] ph := high_confirmed
                IF ph = high AND
                  (ss1_dv_rate = incr OR
                   ss2_dv_rate = incr OR
                   ss3_dv_rate = incr OR
                   ss1_dv = high OR
                   ss2_dv = high OR
                   ss3_dv = high)
                  AND work_rule_base = fc_delta_v7_4_1_4

(*-----------------------------------------------------------------------*)
(*       rule class  bus_tie_rules                                      *)
(*-----------------------------------------------------------------------*)

(* 51-37 *)
        [] bus_tie_statusa := tied ! bus_tie_statusb := tied
                IF busa_tie = on AND
                  busb_tie = on
                  AND work_rule_base = bus_tie_rules
(* 52-38 *)
        [] bus_tie_statusa := tied ! bus_tie_statusc := tied
                IF busa_tie = on AND
                  busc_tie = on
                  AND work_rule_base = bus_tie_rules
```

```
(* 53-39 *)
          [] busa_tie_status := untied
                    IF (busa_tie = off OR
                       (busb_tie = off AND busc_tie = off))
                      AND work_rule_base = bus_tie_rules
(* 54-40 *)
          [] bus_tie_statusb := tied
                    IF busb_tie = on AND
                      busc_tie = on
                      AND work_rule_base = bus_tie_rules
(* 55-41 *)
          [] busb_tie_status := untied
                    IF (busb_tie = off OR
                       (busa_tie = off AND busc_tie = off))
                      AND work_rule_base = bus_tie_rules
(* 56-42 *)
          [] busc_tie_status := untied
                    IF busa_tie = off AND
                      busb_tie = off
                      AND work_rule_base = bus_tie_rules

(*------------------------------------------------------------------------*)
(*        rule class  fc_general                                  *)
(*------------------------------------------------------------------------*)

(* 57-43 *)
          [] status := safed
                    IF cool_p = low AND
                      cool_pump_dp = low AND
                      NOT (h2_reac_vlv = open AND o2_reac_vlv = open)
                      AND work_rule_base = fc_general
```

```
(* 58-44 *)
        [] status := shutdn
                IF fc_rdy_for_ld = off AND
                  cool_pump_dp = low AND
                  ( (o2_reac_vlv <> open) OR (h2_reac_vlv <> open))
                  AND work_rule_base = fc_general
(* 59-45 *)
        [] status := disconnected
                IF fc_mn_conn = off AND
                  fc_ess_conn = off
                  AND work_rule_base = fc_general
(* 60-46 *)
        [] status := disconn_stop
                IF status = disconnected AND
                  fc_rdy_for_ld = off AND
                  cool_pump_dp = low
                  AND work_rule_base = fc_general

(*-----------------------------------------------------------------*)
(*      rule class  fc_stack_t7_1b                                   *)
(*-----------------------------------------------------------------*)

(* 61-47 *)
        [] status1 := cool_lp_prob
                IF stk_t_status2 = high_start AND
                  stk_t_disconn2 = ok
                  AND work_rule_base = fc_stack_t7_1b
(* 62-48 *)
        [] status1 := cool_lp_prob
                IF stk_t_status3 = high_start AND
                  stk_t_disconn3 = ok
                AND work_rule_base = fc_stack_t7_1b
(* 63-49 *)
        [] status1 := fc_or_stk_t_fail
              IF status1 = disconnected AND
                stk_t_status1 = high AND
                (voltage1 = high1 OR voltage1 = high2) AND
                amps1 = low2
                AND work_rule_base = fc_stack_t7_1b
(* 64-50 *)
        [] status3 := fc_or_stk_t_fail
              IF status3 = disconnected AND
                stk_t_status3 = high AND
                (voltage3 = high1 OR voltage3 = high2) AND
                amps3 = low2
                AND work_rule_base = fc_stack_t7_1b
(* 65-51 *)
        [] status2 := fc_or_stk_t_fail
              IF status2 = disconnected AND
                stk_t_status2 = high AND
                (voltage2 = high1 OR voltage2 = high2) AND
                amps2 = low2
                AND work_rule_base = fc_stack_t7_1b
(* 66-52 *)
        [] stack_t2 := hi_snsr_fail
              IF stk_t_status2 = high_start AND
                stk_t_disconn2 <> ok
                AND work_rule_base = fc_stack_t7_1b
(* 67-53 *)
        [] stack_t3 := hi_snsr_fail
              IF stk_t_status3 = high_start AND
```

```
                stk_t_disconn3 <> ok
                AND work_rule_base = fc_stack_t7_1b
(* 68-54 *)
        [] stack_t := snsr_failed
              IF stk_t_rate = high AND
                stack_t = high2 AND
                stack_t = low2
                AND work_rule_base = fc_stack_t7_1b
(* 69-55 *)
        [] status := ok_su_htr_inh
              IF status = disconn_v_low AND
                su_htr = inh AND
                (voltage = high1 OR voltage = high2)
                AND work_rule_base = fc_stack_t7_1b

(*-----------------------------------------------------------------------*)
(*      rule class  fc_purge7_2                                   *)
(*-----------------------------------------------------------------------*)

(* 70 *)
        [] message15 := true
              IF purge_vlv_sw_pos = gpc AND
                auto_purge_seq = on AND
                (h2_flow_rate = high1 OR o2_flow_rate = high1)
                AND work_rule_base = fc_purge7_2
(* 71 *)
        [] message16 := true
              IF purge_htr_sw_pos = gpc AND
                auto_purge_seq = on
                AND work_rule_base = fc_purge7_2
(* 72 *)
```

```
          [] message17 := true
                    IF purge_vlv_sw_pos = open AND
                      (h2_flow_rate = high1 OR o2_flow_rate = high1)
                      AND work_rule_base = fc_purge7_2
(* 73 *)
          [] message18 := true
                    IF auto_purge_seq = on AND
                      fc_purge_alarm = on
                      AND work_rule_base = fc_purge7_2
(* 74 *)
          [] message19 := true
                    IF fc_purge_seq_alarm = on AND
                      auto_purge_seq = off
                      AND work_rule_base = fc_purge7_2
(* 75 *)
          [] message20 := true
                    IF fc_purge_t_alarm = on
                      AND work_rule_base = fc_purge7_2

(*-----------------------------------------------------------------------*)
(*        rule class  fc_ech7_3_1_2                                       *)
(*-----------------------------------------------------------------------*)

(* 76 *)
          [] message21 := true
                    IF fc_rdy_for_ld = on AND
                      (end_cell_htr1 = failed_on OR end_cell_htr2 = failed_on) AND
                      fc_mn_conn = off AND
                      fc_ess_conn = off
                      AND work_rule_base = fc_ech7_3_1_2
(* 77-56 *)
```

```
        [] status := ech_failed_on
                IF (end_cell_htr1 = failed_on OR end_cell_htr2 = failed_on) AND
                  mn_bus = mn_id
                  AND work_rule_base = fc_ech7_3_1_2

(*------------------------------------------------------------------------*)
(*      rule class  fc_volts7_3b                                          *)
(*------------------------------------------------------------------------*)

(* 78-57 *)
        [] load_status := overloaded1
                IF fc_mn_conn = on AND
                  mn_bus = mn_id AND
                  mn_pripl_conn = off AND
                  bus_tie_status = untied AND
                  amps = high1 AND
                  voltage = low1 AND
                  load_status <> tied_short
                  AND work_rule_base = fc_volts7_3b
(* 79-58 *)
        [] status := reconnected
                IF status = disconnected AND
                  cntlr_pwr = on AND
                  voltage = ok AND
                  load_status <> perf_low AND
                  fc_mn_conn = on
                  AND work_rule_base = fc_volts7_3b
(* 80-59 *)
        [] status := internal_short
                IF mn_bus = mn_id AND
                  load_status = overloaded1 AND
                status = disconn_stop AND
                (voltage = low1 OR voltage = ok)
                AND work_rule_base = fc_volts7_3b
(* 81-60 *)
        [] fc_mn_conn := off
              IF voltage = high1 AND
                status = ok AND
                (amps = low2 OR fc_mn_conn = off)
                AND work_rule_base = fc_volts7_3b
(* 82-61 *)
        [] load_status := lightly_loaded
              IF voltage = high1 AND
                amps = low3 AND
                fc_mn_conn = on
                AND work_rule_base = fc_volts7_3b
(* 83-62 *)
        [] status := shorted
              IF load_status = overloaded1 AND
                mn_bus = mn_id AND
                status = disconn_stop AND
                voltage = high1
                AND work_rule_base = fc_volts7_3b
(* 84-63 *)
        [] load_status := overloaded2
              IF fc_id = mn_id AND
                bus_tie_status = untied AND
                voltage = low1 AND
                amps = high1 AND
                (fc_pripl_conn = on OR mn_pripl_conn = on)
                AND work_rule_base = fc_volts7_3b
```

```
(* 85-64 *)
          [] load_status := pripl_short_isolated
                    IF mn_bus = mn_id AND
                      load_status = overloaded2 AND
                      mn_voltage = ok AND
                      amps = ok AND
                      fc_pripl_conn = off AND
                      mn_pripl_conn = off
                      AND work_rule_base = fc_volts7_3b
(* 86 *)
          [] message22 := true
                    IF mn_bus = mn_id AND
                      load_status = tied_short AND
                      bus_tie_status = untied AND
                      voltage = ok AND
                      amps = ok
                      AND work_rule_base = fc_volts7_3b
(* 87-65 *)
          [] load_status := tied_short
                    IF fc_id = mn_id AND
                      bus_tie_status = tied AND
                      amps = high1 AND
                      voltage = low1
                      AND work_rule_base = fc_volts7_3b
(* 88-66 *)
          [] load_status := untied_short
                    IF mn_bus = mn_id AND
                      load_status = tied_short AND
                      bus_tie_status = untied AND
                      voltage = low1 AND
```

```
                amps = high1 AND
                    (mna_voltage = low1 OR
                     mnb_voltage = low1 OR
                     mnc_voltage = low1)
                    AND work_rule_base = fc_volts7_3b
(* 89-67 *)
      [] load_status := untied_short
            IF load_status = tied_short AND
               mn_bus = mn_id AND
               bus_tie_status = untied AND
               voltage = low1 AND
         amps = high1
               AND work_rule_base = fc_volts7_3b

(*----------------------------------------------------------------------*)
(*      rule class  fc_amps7_3c                                        *)
(*----------------------------------------------------------------------*)

(* 90-68 *)
      [] status := degraded
            IF mn_bus = mn_id AND
               status = disconnected AND
               load_status = perf_low AND
               voltage = high1
               AND work_rule_base = fc_amps7_3c
(* 91-69 *)
      [] status := degraded_bad
            IF mn_bus = mn_id AND
         status = disconnected AND
         load_status = perf_low AND
         voltage <> high1
```

```
                 AND work_rule_base = fc_amps7_3c
(* 92-70 *)
      [] load_status := perf_low
                IF voltage = ok AND
                 amps = low3 AND
                 fc_mn_conn = on
                 AND work_rule_base = fc_amps7_3c
(* 93-71 *)
      [] load_status := lightly_loaded
                IF voltage = high1 AND
             amps = low3 AND
             fc_mn_conn = on
                 AND (work_rule_base = fc_amps7_3c OR
                      work_rule_base = fc_volts7_3b)
(* 94 *)
      [] message23 := true
                IF tce = high AND
                 cool_rtn_t = high1
(* 95 *)
      [] message24 := true
                IF (tce = low OR tce = low_confirmed) AND
                 (koh_in_conc = high OR koh_out_conc = high)
(* 96 *)
      [] message25 := true
                IF tce = high AND
                 (prd_h20_lnt = high OR
                  koh_in_conc = low OR
                  koh_out_conc = low)
(* 97 *)
      [] message26 := true
```

```
                IF tce = low AND
                  prd_h20_lnt = low
(* 98 *)
        [] message27 := true
                IF (cool_p = high1 OR cool_p = high2) AND
                  (h2_flow_rate = high2 OR o2_flow_rate = high2)
(* 99 *)
        [] message28 := true
                IF (cool_p = high2 OR cool_p_rate = incr) AND
                  cool_pump_dp = low
(* 100 *)  (* Missing condition *)
        [] message29 := true
                IF (cool_p = high1 OR h2_flow_rate = high1)
(* 101 *)
        [] message30 := true
                IF (cool_p = high1 OR cool_p = high2) AND
                  h2_flow_rate = ok AND
                  o2_flow_rate = ok AND
                  cool_p_rate = ok

        TRACE
        PRINT work_rule_base, o2_reac_vlv, status, status1, status2, status3,
                fc_mn_dv, load_status, stk_t_status, cool_loop_status, prd_h20_lnt,
                h20_rlf_timer, ss1_dv, ss2_dv, ss3_dv, ph,
                bus_tie_statusa, bus_tie_statusb, bus_tie_statusc,
                busa_tie_status, busb_tie_status, busc_tie_status,
                stack_t, stack_t2, stack_t3, fc_mn_conn,
                message1, message2, message3, message4, message5,
                message6, message7, message8, message9, message10,
                message11, message12, message13, message14, message15,
                message16, message17, message18, message19, message20,
                message21, message22, message23, message24, message25,
                message26, message27, message28, message29, message30

END. (* fuel_cell_expert_system *)
```

Bibliography

[Benda 87]

M. Benda, ''Real-Time Applications of AI in the Aerospace Industry,'' Presentation at the Fall School on Artificial Intelligence, The Research Institute of Ecole Normal Superieure, France, September 4, 1987.

[Browne, Cheng & Mok 88]

J. C. Browne, A. M. K. Cheng and A. K. Mok, ''Computer-Aided Design of Real-Time Rule-Based Decision Systems,'' to appear in *IEEE Transactions on Software Engineering*.

[Cheng 87]

A. M. K. Cheng, ''Real-Time Rule-Based Decision Systems,'' *AAAI Workshop on Real-Time Processing in Knowledge-Based Systems*, Seattle, Washington, July 14, 1987.

[Cheng 88]

A. M. K. Cheng, ''Timing Analysis of Self-Stabilizing Programs,'' Technical Report, Department of Computer Science, Univ. of Texas at Austin, 1988.

[Cheng, Browne, Mok & Wang 90]

A. M. K. Cheng, J. C. Browne, A. K. Mok and R.-H. Wang, ''Estella: A Facility for Specifying Behavioral Constraint Assertions in Real-Time Rule-Based Systems,'' submitted to *13th International Conference on Software Engineering*, Austin, Texas, May 1991.

[Cheng & Wang 90]

A. M. K. Cheng and C.-K. Wang, ''Fast Static Analysis of Real-Time Rule-Based Systems to Verify Their Fixed Point Convergence,'' *Proceedings of the 5th Annual IEEE Conference on Computer Assurance (COMPASS 90)*, National Institute of Standards and Technology, Gaithersburg, Maryland, June 1990.

[Chandy & Misra 88]
K. M. Chandy and J. Misra, ''Parallel Program Design: A Foundation'' *Addison-Wesley Publishing Company,* 1988.

[Clarke, Emerson & Sistla 86]
E. M. Clarke, E. A. Emerson, and A. P. Sistla, ''Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications,'' *ACM Transactions on Programming Languages and Systems,* Vol. 8, No. 2, April 1986, pp. 244-263.

[Dershowitz & Manna 79]
N. Dershowitz and Z. Manna, ''Proving Termination with Multiset Orderings,'' *Communications of the ACM*, Vol. 22, No. 8, pp. 465-476, 1979.

[Forgy 81]
C. L. Forgy, ''OPS5 User's Manual,'' Department of Computer Science, Carnegie-Mellon University, Tech. Rep. CMU-CS-81-135, July 1981.

[Garey & Johnson 79]
M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, New York, 1979.

[Gouda & Rosier 88]
M. G. Gouda and L. E. Rosier, ''Self-Stabilization of Acyclic Programs,'' internal note, Department of Computer Science, Univ. of Texas at Austin, Sept. 1988.

[Gries 81]
D. Gries, *The Science of Programming*, Springer-Verlag, 1981.

[Haddawy 86]
P. Haddawy, ''Implementation of and Experiments with a Variable Precision Logic Inference System,'' *Proceedings of the AAAI Conference,*

1986, pp. 238-242.

[Haddawy 87]
P. Haddawy, ‘‘A Variable Precision Logic Inference System Employing the Dempster-Shafer Uncertainty Calculus,’’ M.S. Thesis, Department of Computer Science, University of Illinois, Urbana, 1987.

[Helly 84]
J. J. Helly, ‘‘Distributed Expert System for Space Shuttle Flight Control,’’ Ph.D. Dissertation, Department of Computer Science, UCLA, 1984.

[Koch et al 86]
D. Koch, K. Morris, C. Giffin, and T. Reid, ‘‘Avionic Sensor-based Safing System Technology,’’ Presentation at the Tri-Service Software System Safety Working Group in association with *IEEE COMPASS Conference,* 1986.

[Horowitz and Sahni 82]
E. Horowitz and S. Sahni, ‘‘Fundamentals of Data Structures,’’ Computer Science Press, 1982.

[Laffey et al 88]
T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, ‘‘Real-Time Knowledge-Based Systems,’’ *AI Magazine,* Vol. 9, No. 1, Spring 1988, pp. 27-45.

[Michalski & Winston 86]
R. S. Michalski and P. H. Winston, ‘‘Variable Precision Logic,’’ *Artificial Intelligence Journal,* Vol. 29, No. 2, 1986, pp. 121-146.

[Mok 89]
A. K. Mok, ‘‘Formal Analysis of Real-Time Equational Rule-Based Systems,’’ *10th Real-Time Systems Symposium (RTSS)*, December 1989.

[O'Reilly & Cromarty 85]
C. A. O'Reilly and A. S. Cromarty, ‘‘"Fast" is not "Real-time": Designing

Effective Real-time AI Systems,'' *Applications of Artificial Intelligence, John F. Gilmore, editor, Proceedings of SPIE, 485*.

[Tarjan 72]
R. E. Tarjan, ''Depth First Search and Linear Graph Algorithms,'' *SIAM J. Computing* **1**, pp. 146-160, 1972.

[Ullman 85]
J. D. Ullman, ''Implementation of Logical Query Languages for Databases,'' *ACM Transactions on Database Systems*, Vol. 10, No. 3, pp. 289-321, Sept. 1985.

[Ullman & van Gelder 88]
J. D. Ullman and A. van Gelder, ''Efficient Tests for Top-Down Termination of Logical Rules,'' *Journal of the ACM*, Vol. 35, No. 2, pp. 345-373, April 1988.

[Wang, Mok & Cheng 90]
C.-K. Wang, A K. Mok, and A. M. K. Cheng, ''MRL: A Real-Time Rule-Based Production System,'' to appear in *11th Real-Time Systems Symposium (RTSS)*, December 1990.