N90-22318

Bounded-Time Fault-Tolerant Rule-Based Systems†

James C. Browne
Allen Emerson
Mohamed Gouda
Daniel Miranker
Aloysius Mok
Louis Rosier

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

Abstract

We introduce two systems concepts: bounded response-time and self-stabilization in the context of rule-based programs. These concepts are essential for the design of rule-based programs which must be highly fault-tolerant and perform in a real-time environment. The mechanical analysis of programs for these two properties will be discussed. We have also applied our techniques to analyze a NASA application.

Key words: rule-based programming, real-time, self-stabilization

1. Introduction

The operations and functions of systems that rely on the computer for real-time monitoring and control have become increasingly complex. However, there have been few attempts to formalize the question of whether rule-based systems can deliver adequate performance and be able to recover gracefully from transient faults in *bounded time*. In this paper, we provide a formal framework for answering these important questions.

The class of real-time programs that are investigated herein are called *equational rule-based (***EQL**) 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.

EQL differs from the popular *expert system* languages such as **OPS5** in some important ways. Whereas the interpretation of a language like **OPS5** is defined by the *recognize-act cycle* (Forgy 1981), the basic interpretation cycle of **EQL** is defined by fixed point convergence, and no perference is given to

† Work supported partly by a research grant from the Office of Naval Research under ONR contract number N00014-89-J-1913, ONR contract number N00014-89-J-1472, and also partly by a grant from Texas Instruments Inc.

any enabled rule for firing when two or more are enabled. The differences with **OPS** reflect the goal of our research, which is not to invent yet another *expert system shell.* We want to investigate whether a rule-based program is sufficiently fast to react to a change in the environment, and whether it is sufficiently robust to recover from a corruption of its internal state.

2. Equational Rule-Based Programs: the EQL Language

A **EQL** program consists of a finite set of rules each of which has three parts:

(1) LHS: the left-hand-side of a multiple assignment statement,
(2) RHS: the right-hand-side of a multiple assignment statement, and
(3) EC: the enabling condition.

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 evaluates to 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 format of a rule is:

<variable list> := <expression list> if <boolean expression>

The number of variables on the left hand side must be the same as the number of expressions on the right hand side, and 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 equational rule-based program is a sequence of rule firings (execution of assignment statements whose tests are true). When two or more rules are enabled, the selection of which rule to fire is nondeterministic, i.e., up to the run-time scheduler.

The variables in a **EQL** program are either *input variables* (and their values are determined by sensor readings from the external environment at the beginning of each invocation of the program) or *internal variables*. Input variables do not appear on the left hand side of any assignment statement.

An equational rule-based program is said to have reached a *fixed point* with respect to an internal variable x when either:

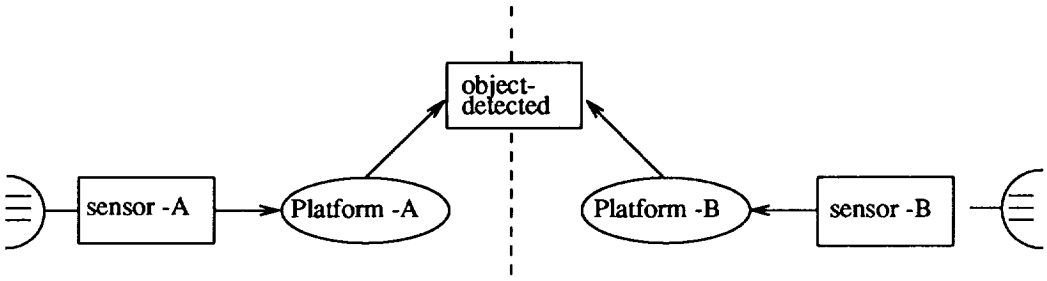
(1) none of the rules are enabled, or
(2) firing of any enabled rule will not change the value of x.

If a program reaches a fixed point with respect to all of its internal variables, then we say that the program has reached a fixed point. 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 internal 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.

3. Bounded Response Time and Recovery from Abnormal States

To evaluate whether an **EQL** program is sufficiently fast to react to a change in the environment, we define the **response time** of an **EQL** program to be the maximum length of a monitor-decide cycle in any execution of the program. The response time of a program is infinite if it is possible for it to never reach a fixed point from a launch state in finite time.

Example



Initially, object-detected = false

System goals are:

(1) Set object-detected to true if either sensor detects object.

(2) In any computation, object-detected should reach a fixed point.

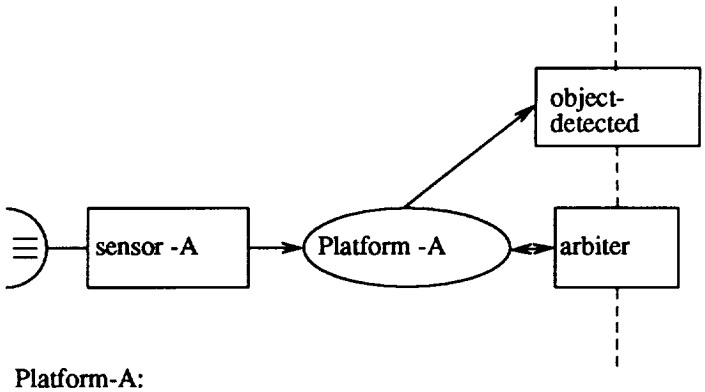
(3) The system should be self-stabilizing.

Fig 1: The system

Consider the parallel system shown in Figure 1 whose purpose is to determine whether an object has been detected by either of its two sensors. That is, the variable *object-detected*, initially set to *false*, is to be set to *true* by the code labelled *Platform_A* (*Platform_B*) whenever *sensor-A* (*sensor-B*) detects an object.

Figure 2 shows an attempt to implement this parallel system in terms of a rule-based program. Notice that this implementation does not reach a fixed point with respect to the variable *object-detected* since the value of *object-detected* will continually alternate between *true* and *false* if only one of the sensors detects an object, e.g., when sensor-A reports a 1 and sensor-B reports a 0. Thus this program has an infinite worst-case response time. Obviously, this is undesirable. One of our goals is to ensure that **EQL** programs for real-time applications must have bounded response time.

To evaluate whether an **EQL** program is sufficiently robust to recover from a transient upset, we are interested in the behavior of a program after some of its internal variables have been unintentionally modified by an external disturbance (e.g., a bit in dynamic memory may be flipped by cosmic ray). In such a case, we would like the program to be able to recover (through further execution) to a state that can be reached from some normal program execution path. A program that can always effect such a recovery is said to be self-stabilizing. It is also our goal to ensure that **EQL** programs for real-time applications are self-stabilizing.



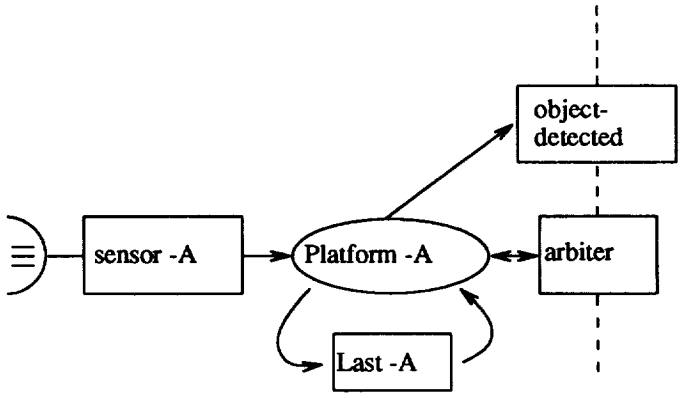
Platform-A:

object-detected, arbiter := true, B
if arbiter = A ∧ sensor-A = 1

[] object-detected, arbiter := false, B
if arbiter = A ∧ sensor-A = 0

Platform-B: (Summetric to A's code)

Fig 2: First attempt



Platform -A:

arbiter, object-detected, last-A := B, true, true
if arbiter = A ∧ sensor-A = 1

[] arbiter, object-detected, last-A := B, false, false
if arbiter = A ∧ sensor-A =0 ∧ last-A = false

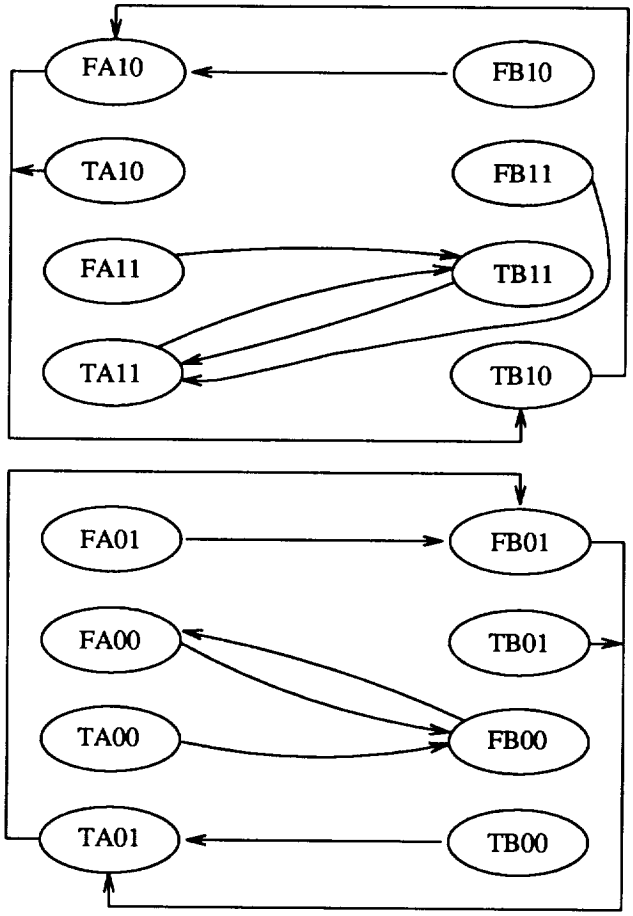
[] arbiter := B
if arbiter = A ∧ sensor-A = 0 ∧ last-A = true

Platform -B: (symmetric)

Fig 3: Second attempt

Figure 3 shows a second attempt to implement the system of Figure 1. Modulo a change in the sensor values (these are the system's input variables which are updated at the beginning of each sampling

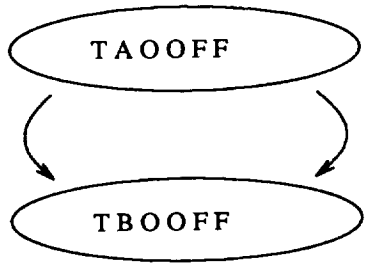
period), the implementation of Figure 3 will always reach a fixed point. Even though the implementation of Figure 2 does not always reach a fixed point, it is self-stabilizing, however, since with respect to any sensor reading any fixed point of *<arbiter, object-detected>* constitutes a normal state (see Figure 4).



state = <object-detected,arbiter,sensor-A,sensor-B>

Fig 4: State diagram

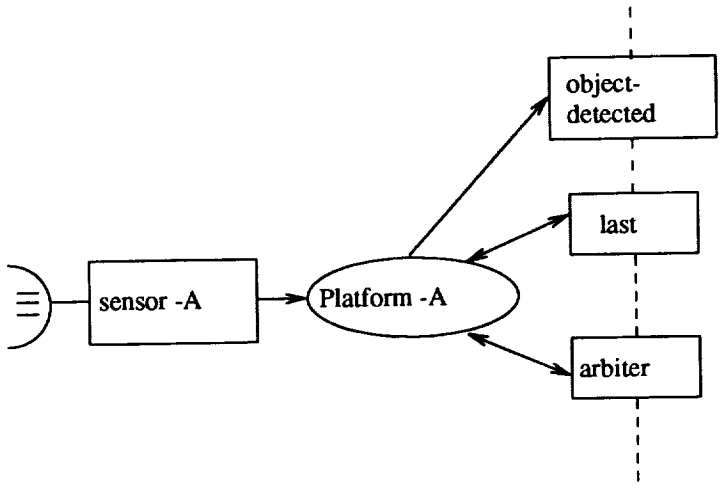
In contrast, the implementation of Figure 3 is not, however, self-stabilizing as the system in the abnormal state where *object-detected* = *true* , *last-A* = *last-B* = *false* , and *sensor-A* = *sensor-B* = 0 will be unable to recover (see Figure 5).



state = <object-detected,arbiter,sensor-a,sensor-B,last-A,last-B>

Fig. 5: Portion of the state diagram with
object-detected = true
last-A = last-B = false
sensor-A = sensor-B = 0

It is possible to design an implementation which always reaches a fixed point and is self-stabilizing. Such a program is shown in figure 6. A portion of its state-transition graph is shown in Figure 7.



Platform-A:

arbiter, object-detected, last := B, true, A
if arbiter = A ∧ sensor-A = 1

[] arbiter, object-detected := B, false
if arbiter = A ∧ sensor-A = 0 ∧ last = A

[] arbiter := B
if arbiter = A ∧ sensor-A = 0 ∧ last = B

Platform-B: (symmetric)

Fig. 6: Last attempt

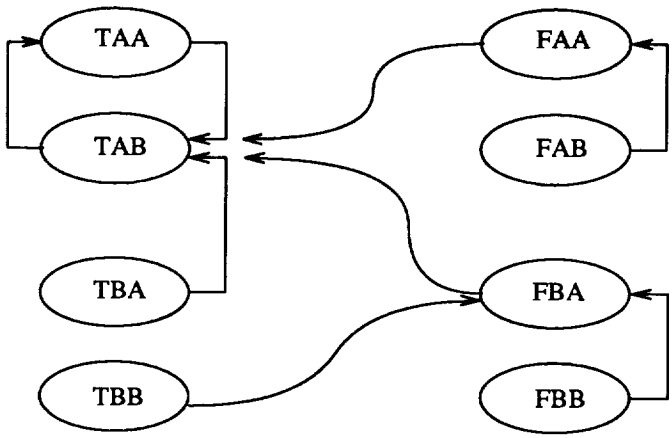


Fig 7: Portion of the state diagram
with sensor-A = 1 and sensor-B = 0

4. Formalization via State Space Representation

In order to formalize the response time and self-stabilization property of a rule-based program, we represent an **EQL** program in terms of its state space graph. The state space graph of an **EQL** 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 each $x_{i,\ 1 \le i \le n}$ is a value in the domain of the i^{th} input sensor variable and each $s_{j,\ 1 \le j \le p}$ is a value in the domain of the j^{th} internal variable. We say that a rule is **enabled** at vertex u iff its test is satisfied by the tuple of variable values at vertex u. E is a set of edges each of which denotes the firing of a rule: an edge (u,v) connects vertex u to vertex v iff there is a rule R which is enabled at vertex u, and firing R will modify the internal variables to have the same values as the tuple at vertex v. Whenever there is no confusion, we shall use the terms state and vertex interchangeably.

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 v 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., (v,v). Obviously, if the execution of a program has reached a fixed point, then every rule is either not enabled or its firing does not modify any of the variables.

An invocation of a rule-based program (a monitor-decide cycle) can be thought of as tracing a path in the state space graph. We say that a fixed point is an **end-point** of a state s if that fixed point is reachable from s. 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 internal 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).

With respect to a state space graph, a tuple is a **normal state** if it appears on a path from a launch state to an end-point. Tuples which are not normal states are **abnormal states**. In the absence of faults or malfunctions, the variables of a program are by definition always in normal states. When the variables of a program are in an abnormal state (e.g., due to a hardware fault which arbitrarily writes over some of the internal variables), firing of an enabled rule may or may not bring the program back to a normal state.

To model the effect of faults, we define a **deviation function** B which maps each normal state s into a set $B(s)$ of tuples which may be normal or abnormal states. Intuitively, if a fault occurs when the program is in state s, then the program will be in one of the states specified by $B(s)$. We note that in the case where faults can have arbitrary effects on the variables, $B(s)$ may be the entire set of $n+p$-tuples. However, we expect that hardware techniques (e.g., error-correcting code) can be used to restrict the effect of faults so that $B(s)$ need not be very large for any normal state s. We say that a program is **deviation-bounded** with respect to the function B if for every normal state s, a fault will put the system into only a state in $B(s)$.

We are interested in systems which can recover automatically from transient faults. In particular, we define a **recovery function** R which maps each normal state s into $R(s)$, a subset of the set of normal states. The goal is to design systems such that if a transient fault occurs in state s and puts the system in an abnormal state, then further execution of the program will automatically bring the system back to a normal state in $R(s)$.

We say that a program is **self-stabilizing** with respect to the function R if for every normal state s, $R(s)$ contains all fixed points reachable from any state that the system may enter from s after a transient upset. This is the case if the end-points of all the states in $B(s)$ are in $R(s)$.

5. Bounded Response Time Analysis

With the state space representation, the response time of a program can be measured by the maximum length of a path from a launch state to a fixed point. (We assume that the evaluation of the enabling conditions and the right-hand-side of the assignment statements takes bounded time.) The problem of interest is to decide whether a fixed point can always be reached from a launch state on any sufficiently long path and if so, whether all these paths are shorter than a given bound. Conversion of path length to real time is possible by introducing a time metric on the paths of the state space graph. This depends on the specific architecture of an implementation and will not be discussed here.

If the state space of a program is infinite, it is in general impossible to decide if an **EQL** program has finite response time. For programs whose state spaces are finite, we can determine the response time by a brute-force state space search. However, this approach is impractical for larger programs since the number of states can grow exponentially fast. The determination of response time for finite state programs can be shown to be PSPACE-hard (Mok 1989). For certain classes of **EQL** programs, it is not necessary to check the complete state space in order to solve the decision problem. If the rules of a program fall under certain special forms (templates), the program is guaranteed to always reach a fixed point in a finite number of iterations, and there are efficient procedures to determine whether a set of rules falls under these special cases. Matching special forms alone, however, is not very effective since they may cover only a small portion of practical rule-based programs. We have invented a general analysis strategy which combines the power of special forms with program rewriting to combat the combinatorial explosion problem.

5.1 A Special Form

As an example, one of these special forms which is especially useful will be given below. First, some definitions are in order.

For ease of discussion, we define three sets of variables for an **EQL** program:

L = { v | v is a variable appearing in LHS of some assignment statement }
R = { v | v is a variable appearing in RHS of some assignment statement }
T = { v | v is a variable appearing in EC of some assignment statement }

Let $T = \{ v_1, v_2, \ldots, v_n \}$ and let $\bar{v}$ be the vector $<v_1, v_2, \ldots, v_n>$. 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 { *true*, *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*. 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.

Let L_x denote the set of variables appearing in LHS of rule x. Two rules a and b are said to be **compatible** if 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.

We now give a special form of rules for which the decision problem can be solved efficiently.

• **Special Form A:**

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

(1) Constant terms are assigned to all the variables in L, i.e., $R = \emptyset$.
(2) All of the rules are compatible pairwise.
(3) $L \cap T = \emptyset$.

Theorem

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

The utility of special form A might seem quite limited since the rather restrictive conditions of the special form must be satisfied by the *complete* set of rules in a program. However, the main use of the special form in our analysis tools is not to identify special-case programs. We leverage the special form by applying it to appropriate subsets of rules in a program so as to find out if at least some of the variables must attain stable values in finite time.

5.2 The General Analysis Strategy

The exploitation of special forms in our general analysis strategy is best explained by an example.

Example

input: read(b, c)

1. $a1$:= true IF b = true $\wedge$ c = true
2. [] $a1$:= true IF b = true $\wedge$ c = false
3. [] $a2$:= false IF c = true
4. [] $a3$:= true IF $a1$ = true $\wedge$ $a2$ = false
5. [] $a4$:= true IF $a1$ = false $\wedge$ $a2$ = false
6. [] $a4$:= false IF $a1$ = false $\wedge$ $a2$ = true

For this program, $L \cap T \neq \emptyset$ and thus the rules are not of the special form described in the preceding section. However, observe that rules 1, 2 and 3 by themselves are of the special form A and that all the variables in these rules do not appear in the left-hand-side of the rest of the rules of the program and thus will not be modified by them. We can conclude that the variables $a1$ and $a2$ must attain stable values in finite 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 $\wedge$ $a2$ = false
5. [] $a4$:= true IF $a1$ = false $\wedge$ $a2$ = false
6. [] $a4$:= false IF $a1$ = false $\wedge$ $a2$ = 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 finite time. This guarantees that the original program must reach a fixed point in finite time.

There are in fact more special forms that can be exploited in the above fashion. Our general strategy for tackling the analysis problem is as follows.

Algorithm GIA

(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. 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. For this purpose, we use the model checking technique for the temporal logic **RTCTL** (Emerson, Mok, Srinivasan & Sistla 1989). 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).

Intuitively, the general strategy described above allows us to use a special form in the induction step of a proof, by structural induction, that an **EQL** program has bounded response time. Thus relatively restrictive special forms may be exploited to analyze a much larger class of programs.

6. Self-stabilization via Program Transformation

In general, it is not always possible to implement an application by a self-stabilizing program (Gouda, Howell & Rosier 1988). However, for a special class of **EQL** programs, it is always possible to transform a program in this class into an equivalent one which is deviation-bounded with respect to a function B and self-stabilizing with respect to a function R where: for any normal state s, $B(s)$ contains all tuples whose input-variable components agree with s, and $R(s)$ contains all end-points of s. Notice that this $B(s)$ requires the input variables remain unchanged by a transient upset. This may not be necessary if the input variables can be restored by repeating the sensor readings after the transient has subsided.

6.1 Acyclic Programs

We now consider the class of rule-based programs where each program P satisfies the following four conditions. Programs in this class are called acyclic programs.

[1] Syntax: Program P is defined by a finite set of assignment statements each of which is of the form:

$$x_i := B_i(x) \text{ if } C_i(x);$$

where x_i is a variable in P, x is the vector of all variables in P (thus x_i is a component of x), $B_i(x)$ is an expression of the same type as x_i and $C_i(x)$ is a boolean expression over the variables in P. Each internal variable x_i has an initial value denoted $\hat{x}_i$.

[2] Semantics: Consistent with **EQL** semantics, the assignment statements in P are executed one at a time. The order in which the statements are executed is arbitrary provided that each statement is executed infinitely often.

[3] Well-Formedness: For every pair of statements with the same left side

$$x_i := B_i(x) \text{ if } C_i(x);$$
$$x_i := D_i(x) \text{ if } E_i(x);$$

and for every value s of vector x, we have

$$C_i(s) \wedge E_i(s) \rightarrow B_i(s) = D_i(s)$$

[4] Acyclicity: The dependency graph of a program P is a directed graph where each node represents an internal variable of P, and there is an edge from node x_i to node x_j iff x_i appears in the right side of an assignment statement (in P) whose left side is x_j. A program is called **acyclic** iff its dependency graph is acyclic. P must be acyclic.

Acyclic programs can be shown to obey the following two theorems.

Theorem

Executing the statements of an acyclic program starting from any normal state leads eventually to a fixed point.

Theorem

If a program is acyclic, then it will be recognized by the GIA algorithm with special form A as having bounded response time.

6.2 Self-Stabilization

For an acyclic program P, it can be shown that for each pair s_1 and s_2 of distinct fixed points of P, there is at least one input variable whose value in s_1 is different from its value in s_2. In other words, the fixed point that an acyclic program P can reach starting from any state s, depends solely on the values of the input variables in s and not on the values of internal variables in s. Therefore, if program P is at a fixed point and the values of one or more internal variables change due to some failure, P is guaranteed to converge back to the same fixed point (as long as the values of its input variables remain unchanged).

6.3 Implementation

Given an acyclic program, we shall transform it into another program which is self-stabilizing. The transformed program must also implement the semantics of the original program.

A program P is said to **implement** program Q iff the following conditions hold:

[1] Programs P and Q have the same input variables.

[2] Each internal variable of Q is an internal variable of P.

[3] Each fixed point of P is a reachable fixed point of Q, and, if Q has a reachable fixed-point then P has a fixed point.

Theorem

For each acyclic program Q, there is an acyclic self-stabilizing program P that implements Q.

Proof: (by construction)

Every statement in Q is a statement in P. For each internal variable x_i in Q, do the following. Let all the statements in Q with x_i on the left side be:

$$x_i = B_i(x) \text{ if } C_i(x)$$
$$\ldots$$
$$x_i = D_i(x) \text{ if } E_i(x)$$

Then add the following statement to P:

$$x_i := \hat{x_i} \text{ if } \overline{C_i(x)} \wedge \cdots \wedge \overline{E_i(x)}$$

Here $\hat{x_i}$ is the initial value of the variable x_i. The resulting program P is acyclic, self-stabilizing and implements Q.

In fact, for acyclic programs, we can show that the end-point of any launch state is unique and depends only on the value of the input variables. Let $F(x_1, \ldots, x_n)$ be the function which maps a launch state into its end-point.

Theorem

For each acyclic program P, the self-stabilizing version of P has a recovery function given by: $R(x_1, \ldots, x_n, s_1, \ldots, s_p) = F(x_1, \ldots, x_n)$.

7. Application to NASA Program

We have taken a NASA application: the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure Control System (Helly 1984) and translated it directly into an EQL program. This program has 36 rules and 31 internal variables. We have mechanically verified that this program has bounded response time by using the GIA algorithm and Special Form A. The analysis took under 1 second of time on a SUN 3® workstation, whereas a brute-force state space search took over a week, even for a 20-rule subset of the program. We also determined that this program is not self-stabilizing but is acyclic. The transformation described in the paper was used to convert it into a self-stabilizing program.

8. Conclusion

In this paper, we have introduced two systems concepts: the notion of response time for a rule-based program and the application of self-stabilization to rule-based programs. These two concepts are essential for the design of rule-based programs which must be highly fault-tolerant and perform in a real-time

environment. The mechanical analysis of programs for response-time boundedness was discussed. We also gave an algorithm to convert non-self-stabilizing programs to self-stabilizing ones for the class of acyclic programs. These concepts have been applied to a NASA program, the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure Control System.

Much work remains to be done, such as implementation techniques for realizing a given deviation function, more powerful techniques for determining response time and transformation techniques for ensuring self-stabilization.

Bibliography

Emerson, E. A., Mok, A, Srinivasan, J. & Sistla, A. P. (1989). Quantitative Temporal Reasoning. *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.

Forgy, C. L. (1981). *OPS5 User's Manual.* Department of Computer Science, Carnegie-Mellon University, Tech. Rep. CMU-CS-81-135, July 1981.

Gouda, M., Howell, R. & Rosier, L. (1988). The Instability of Self-Stabilization. *manuscripted, submitted for publication, 1988*

Helly, J. J. (1984). Distributed Expert System for Space Shuttle Flight Control. *Ph.D. Dissertation*, Department of Computer Science, UCLA, 1984.

Mok, A. K. (1989). Formal Analysis of Real-Time Equational Rule-Based Programs. *Proceedings, 10th Real Time Systems Symposium*, Los Angeles, December 5-7, 1989, pp. 308-318.

Appendix

The following is the **EQL** version of a NASA application: the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure Control System. This program was shown to have bounded response time by using the GIA algorithm with Special Form A. It was also transformed into a self-stabilizing version.

```
* SSV Cryogenic Hydrogen Pressure Malfunction Procedure
* Non-self-stabilizing version
*
* Translated into EQL by Albert Mo Kim Cheng
* 36 rules


PROGRAM cryov63a;

RULES
  v63a2 := true IF (v63a1a = true)
[]v63a4 := true IF (v63a1c = true) AND (v63a3 = true)
[]v63a6 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = true)
[]v63a7 := true IF (v63a6 = true)
[]v63a9 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
                   (v63a8 = true)
[]v63a10 := true IF (v63a9 = true)
[]v63a14 := true IF ((v63a12 = true) OR ((v63a12 = false) AND (v63a13 = true)))
[]v63a15 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
                    (v63a8 = false) AND (v63a11 = false) AND
```

```
                    (v63a12 = true) AND (v63a14 = true)
[]v63a18 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
                    (v63a8 = false) AND (v63a11 = true) AND
                    (v63a16 = true) AND (v63a17 = true)
[]v63a19 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
                    (v63a8 = false) AND (v63a11 = true) AND (v63a16 = true)
[]v63a20 := true IF (v63a1c = true) AND (v63a3 = false) AND (v63a5 = false) AND
                    (v63a8 = false) AND (v63a11 = true) AND
                    (v63a16 = true) AND (v63a17 = false)
[]v63a21 := true IF ((v63a19 = true) OR (v63a20 = true))
[]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)
[]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)
[]v63a27 := true IF (v63a26 = true) AND (((v63a23 = true) AND (v63a1b = true)) OR
                                         (v63b7 = true))
[]v63a28 := true IF ((v63a25 = true) OR (v63a15 = true))
[]v63a30 := true IF (((v63a1b = true) AND (v63a23 = true)) OR (v63b7 = true)) AND
                    (v63a26 = false) AND (v63a29 = true)
[]v63a33 := true IF (v63a32 = false) AND (v63a31 = true) AND (v63a29 = false) AND
                    (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a35 := true IF (v63a32 = true) AND (v63a31 = true) AND (v63a29 = true) AND
                    (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a36 := true IF (v63a34b = true) AND (v63a31 = false) AND (v63a29 = false) AND
                    (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a37 := true IF (v63a30 = true) AND (v63a33 = false) AND (v63a35 = false) AND
                    (v63a38 = true)
[]v63a38 := true IF (v63a34a = true) AND (v63a31 = false) AND (v63a29 = false) AND
                    (v63a26 = false) AND (v63a23 = true) AND (v63a1b = true)
[]v63a39 := true IF (v63a36 = true)
[]v63a42 := true IF (v63a41 = true) AND (v63a40 = true) AND (v63a23 = false) AND
                    (v63a1b = true)
[]v63a43 := true IF (v63a41 = false) AND (v63a40 = true) AND (v63a23 = false) AND
                    (v63a1b = true)
[]v63a44 := true IF (v63a42 = true) OR (v63a47 = true)
[]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)
[]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)
[]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)
[]v63a51 := true IF (v63a50 = true)
[]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)
[]v63a53 := true IF (v63a52 = true)

END.
```