Deriving Response-Time Bounds for Equational Rule-Based Programs *

Rwo-Hsi Wang Aloysius K. Mok

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

Abstract

The response time of a rule-based program is defined as the maximum number of rule firings before a fixed point of the program is reached from a start state. In this paper, we present several principles which make use of two relations, potential-trigger and suppression, for deriving tight response-time bounds. While the computation of these two relations is costly in general, we show how they can be efficiently approximated by refining the necessary/sufficient conditions for these relations to be satisfied. A response-time analyzer based on the theories in this paper has been implemented to analyze programs whose potential-trigger relations are acyclic. We demonstrate the analysis process with an example program which has infinite state space. Our analyzer takes only seconds to derive a tight bound on the example program's response time.

1 Introduction

In recent years, there has been increasing interest in the use of rule-based programs in time-critical applications. In these applications, the rule-based program is often used to monitor the environment and must compute an output in response to significant changes that are observed in the environment.

In [5], the concept of a response time is defined for the equational rule-based program (**EQL**) by the number of rule-firings before the program reaches a fixed point (a program state where all the program variables attain stable values). It was shown in that paper that the determination of the response time for arbitrary programs is in general undecidable, and is PSPACE-hard for finite-state programs. A divide-and-conquer strategy was also proposed for the response-time analysis of practical rule-based programs. While the strategy in [5] is efficient in determining the existence of response-time bounds, much work is still needed for deriving tight bounds.

In this paper, we present several principles for deriving tight response-time bounds. These principles make use of relations defined on the rules of a program, namely, the potential-trigger and suppression relations. While they may in general be difficult to compute exactly, we exploit efficient ways to compute approximations to these relations that are useful for time-bound analysis. A response-time analyzer based on the theory in this paper has been implemented to analyze programs whose potential-trigger relations are acyclic. We demonstrate the analysis process with two practical examples.

The remainder of this paper is organized as follows. The **EQL** language is described in Section 2, together with program state space and program execution. In Section 3 we define our problem, response-time analysis, and present several principles for deriving response-time bounds for a special class of programs. These principles make heavy use of two rule relations, the potential-trigger and suppression relations. In Section 4 we show how to approximate these relations. An analysis example is presented in Section 5, followed by concluding remarks and future research topics in Section 6. The source code of the analyzed program is given in the Appendix for reference.

2 The EQL language

EQL is a rule-based programming language designed for developing real-time decision systems [2]. An **EQL** program consists of a finite set of rules each of which is of the form:

$$v_1 := e_1 \ ! \ v_2 := e_2 \ ! \ \ldots \ ! \ v_m := e_m \ \text{IF} \ \textit{test}$$

where e_i is an integer expression whose value is to be assigned to variable v_i when the rule fires, $m \geq 1$, and

*The research reported herein is supported by a research grant from the Office of Naval Research under ONR contract number N00014-89-J-1472, a research funding from the Texas Advanced Technology Program (TATP) under contract number 003658-250, and a grant from Texas Instruments Corporation.

test is a predicate on the variables in the program. 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.

A rule is *enabled* at some instant if its test is evaluated to true according to the values of its variables at that moment, otherwise it is *disabled.* A rule is *conflictive* at some instant if any of its LHS variable is not equal to the corresponding RHS expression. A *rule firing* is the execution of the multiple assignment statement, which consists of the evaluation of all the RHS expressions, followed by updating the LHS variables with the values of the corresponding expressions. A rule is *firable* iff it is enabled and conflictive.

An *execution* of an **EQL** program is a sequence of rule firings. When two or more rules are firable, the selection of which rule to fire is nondeterministic. An **EQL** program is said to have reached a *fixed point* when none of its rules are firable.

2.1 State space representation

The *state space* S of an **EQL** program is the cartesian product of the domains of each of its variables. At each moment during an execution, the state of an **EQL** program is determined by the values of its variables at that instant. As the result of a rule firing, the values of some variables will change, leading the program to a new state. The *state space graph* of an **EQL** program is an irreflexive digraph $G = (S, E)$, where $E \subseteq S \times S$ and denotes state transitions. Let F_s be the set of firable rules of a program at state s. And let $A_r(s)$ be the new state reached by firing rule r at state s. There is an edge (s_i, s_j) in G iff there exists a rule r in the program such that $r \in F_{s_i}$ and $A_r(s_i) = s_j$.

From the viewpoint of state space representation, a program execution can be mapped to a (possibly infinite) path in G. The starting vertex of the execution path is the initial state; while the ending vertex, if any, is a *fixed point*, which is a vertex in G whose out-degree is 0. The length of an execution path equals the number of rule firings of the execution. In the following we shall use $\phi(s, r)$ to denote a firing of rule r at state s. When there is no confusion the part (s, r) can be dropped for convenience, and ϕ can have an optional subscript to refer to a particular firing in a firing sequence. We are now ready to formally define a program execution.

Given an **EQL** program (and thus its state space S) and an initial state $s_0 \in S$, we say a firing sequence $\Phi = \langle \phi_1(s_1, f_1), \phi_2(s_2, f_2), \ldots, \phi_n(s_n, f_n)\rangle$, $n \geq 0$, is an *execution* of the program iff

- $n = 0$ and $F_{s_0} = \emptyset$, or
- $n > 0$ and
  1. $s_1 = s_0$, and
  2. $f_i \in F_{s_i}$, for $1 \leq i \leq n$, and
  3. $A_{f_i}(s_i) = s_{i+1}$, for $1 \leq i < n$, and
  4. if n is finite then $A_{f_n}(s_n)$ is a fixed point.

We call n the *length* of the execution.

3 Response-time analysis

We define the *response time* of an **EQL** program to be the length of the longest execution of the program. The goal of our response-time analysis is to determine whether the response time of an **EQL** program is bounded, and if it is, to find a reasonably tight bound. Since the length of an execution equals the sum of number of firings of each rule during the execution, if we know the bound on number of firings in any execution for each rule then the response time of the program is bounded by the sum of these bounds. More generally, if $\{\pi_1, \pi_2, \ldots, \pi_k\}$ is a partition on the set of rules of an **EQL** program, and we know n_i, $1 \leq i \leq k$, where n_i is the bound on number of total firings of the rules in π_i during any execution, then the response time of the program is bounded by $\sum_{i=1}^{k} n_i$. In the following we focus on how to derive such bounds for a set of rules.

Given a program execution $\langle \phi_1(s_1, f_1), \phi_2(s_2, f_2), \ldots, \phi_n(s_n, f_n)\rangle$, we say the firing $\phi_i(s_i, f_i)$ is a *trigger* of rule b,denoted by $\phi_i(s_i, f_i) \rightsquigarrow b$, iff

1. $b \in F_{s_{i+1}} - F_{s_i}$, or
2. $b = f_i$ and $b \notin F_{s_i} - F_{s_{i+1}}$.

Given a program execution $\langle \phi_1(s_1, f_1), \phi_2(s_2, f_2), \ldots, \phi_n(s_n, f_n)\rangle$, we say the firing $\phi_k(s_k, f_k)$ is triggered by the firing $\phi_i(s_i, f_i), i < k$, (or, equivalently, the firing $\phi_i(s_i, f_i)$ is a trigger of the firing $\phi_k(s_k, f_k), i < k$), denoted by $\phi_i(s_i, f_i) \rightsquigarrow \phi_k(s_k, f_k)$, iff

1. $\phi_i(s_i, f_i) \rightsquigarrow f_k$, and
2. $f_k \in F_{s_j}$ and $f_k \neq f_j$ for $i < j < k$.

A firing is called a *triggered firing* iff it has a trigger.

The trigger relation captures the causal relation among rule firings and is closely related to the response time of a program execution. This is manifested in the following theorems.

Lemma 1 *Any two triggered firings of the same rule must be triggered by different firings.*

Proof. Let $\langle \phi_1(s_1, f_1), \phi_2(s_2, f_2), ...\rangle$ be a program execution. Suppose that two firings of rule b, ϕ_k, ϕ_l, are both triggered by firing ϕ_i. Without losing generality, we assume $k < l$. Since $\phi_i \rightsquigarrow \phi_l$, by definition, $b \neq f_j$, for $i < j < l$. And since $\phi_i \rightsquigarrow \phi_k$, it follows from definition that $i < k$. Therefore $b = f_k$ and $i < k < l$. However, this is a contradiction. □

Corollary 1 *For each firing of rule a at most one firing of rule b can be triggered by the firing of a.*

Lemma 2 *A triggered firing has exactly one trigger.*

Proof. By definition we know that a triggered firing has at least one trigger. Next we show that a triggered firing has only one trigger. Let $\langle \phi_1(s_1, f_1), \phi_2(s_2, f_2), ... \rangle$ be a program execution. Suppose two firings $\phi_i(s_i, f_i)$, $\phi_j(s_j, f_j)$, $i \neq j$, are both triggers of firing $\phi_k(s_k, f_k)$. Without losing generality, we assume $i < j$. By definition, we know $i < j < k$, $f_k \in F_{s_l}$ and $f_k \neq f_l$ for $i < l < k$, and $\phi_j(s_j, f_j) \rightsquigarrow f_k$. Thus we have either of the following situations:

1. $f_k \in F_{s_{j+1}} - F_{s_j}$. This means $f_k \notin F_{s_j}$. But this is a contradiction.
2. $f_k = f_j$ (and $f_k \notin F_{s_j} - F_{s_{j+1}}$). This is also a contradiction. □

Lemma 3 *The trigger relation on the firings of an execution is a forest.*

Proof. By definition and Lemma 2 we know that every firing has at most one trigger. To show that there is no cycle in the trigger relation notice that there is a temporal precedence requirement in the definition of trigger. That is, a firing can only be triggered by an earlier firing. Thus a cycle in the trigger relation would mean some firing is triggered by a later firing, which is contradictory. □

Lemma 4 *If a rule is initially firable, it can be fired for the first time without a trigger. Other than that, every rule firing must have a trigger.*

Proof. Let $\langle \phi_1(s_1, f_1), \phi_2(s_2, f_2), ...\rangle$ be a program execution. Suppose rule b is fired at s_k, i.e., $b = f_k$. There is a smallest i, $1 \leq i \leq k$, such that $b \in F_{s_j}$ and $b \neq f_j$ for $i \leq j < k$. (Note that k is a candidate and therefore such i always exists.) Then we must have one of the following situations:

1. $i = 1$. Since $b \in F_{s_1}$, b is initially firable. And since $b \neq f_j$ for $1 \leq j < k$, ϕ_k is the first firing of b.
2. $i > 1, b \notin F_{s_{i-1}}$. Since $b \in F_{s_i}$, by definition $\phi_{i-1} \rightsquigarrow b$. And since $b \in F_{s_j}$ and $b \neq f_j$ for $i \leq j < k$, by definition, $\phi_{i-1} \rightsquigarrow \phi_k$.
3. $i > 1$, $b = f_{i-1}$. Since $b \in F_{s_{i-1}}$ and $b \in F_{s_i}$, it follows that $\phi_{i-1} \rightsquigarrow b$. And since $b \in F_{s_j}$ and $b \neq f_j$ for $i \leq j < k$, by definition, $\phi_{i-1} \rightsquigarrow \phi_k$. □

Motivated by the result of Lemma 3, we have the following definitions.

The *weight* of a firing ϕ_i in an execution Φ, denoted by $\omega_\Phi(\phi_i)$, is the size of the subtree whose root is ϕ_i in the trigger relation on the firings. Formally,

$$\omega_\Phi(\phi_i) \triangleq |\{\phi_j \ : \ \phi_i \rightsquigarrow^* \phi_j\}| \tag{1}$$

where $\rightsquigarrow^*$ is the reflexive and transitive closure of $\rightsquigarrow$. The reader can verify that

$$\omega_\Phi(\phi_i) = 1 + \sum_j \omega_\Phi(\phi_j), \ \phi_i \rightsquigarrow \phi_j \tag{2}$$

The *weight* of a rule r in an execution Φ, denoted by $\omega_\Phi(r)$, is defined as

$$\omega_\Phi(r) \triangleq \begin{cases} 0 & \text{if } r \text{ is never fired in } \Phi \\ \omega_\Phi(\phi) & \phi \text{ is } r\text{'s first firing in } \Phi \end{cases} \tag{3}$$

Theorem 1 *The length of a program execution equals the sum of the weights of the rules whose first firings are not triggered firings.*

Proof. By Lemma 3 we know the length of an execution equals the sum of the sizes of the trees in the trigger relation on the firings. The size of each tree, by definition, is the weight of its root, which is a non-triggered firing. By Lemma 4 we know that each non-triggered firing must be the first firing of some rule. Therefore the weight of a non-triggered firing of some rule equals, by definition, the weight of the rule. □

Since our goal is to find an upper bound on program response time, and since the length of a program execution equals the sum of some rule weights in the execution, it seems worth while to study the bound on the maximum weight of a rule among all possible executions. This leads to the following definitions.

Let $\omega(r)$ be the largest weight of rule r among all possible executions of the **EQL** program containing r. A *weight bound* of r, denoted by $\Omega(r)$ is an integer satisfying $\Omega(r) \geq \omega(r)$.

Strictly speaking, not every rule has a weight bound by this definition. Those which may indirectly trigger infinite firings do not. However, once we determine that the response time of a program is unbounded, it is no longer meaningful to compute the value of weight bounds. To simplify the reasoning, we shall treat ∞ as a special "integer" so that every rule has a weight bound.

Lemma 5 *The weight of any firing of a rule r in any execution is less than or equal to* $\Omega(r)$.

Proof sketch. Suppose $\omega_\Phi(\phi_i(s_i, r))$ in some execution Φ is greater than $\Omega(r)$. Then there exists an execution Φ' of the same program where s_i is the initial state s'_0 and the suffix of Φ starting from ϕ_i constitutes Φ'. It follows that $\omega_{\Phi'}(\phi'_1(s'_1, r)) = \omega_\Phi(\phi_i(s_i, r))$. Since $\phi'_1(s'_1, r)$ is the first firing of r in Φ', it is contradictory that $\omega_{\Phi'}(\phi'_1(s'_1, r)) > \Omega(r)$. □

Given an **EQL** program, we say rule a is a *potential trigger* of rule b, denoted by $a \rightsquigarrow b$, iff there exists an execution of the program in which some firing of a is the trigger of a firing of b. When $a = b$ we also say a is a *potential self-trigger*.

The *potential-trigger* (PT) graph of an **EQL** program is a digraph $G = (V, E)$, where V is the set of vertices each of which represents a distinct rule, and E is the set of edges such that there is an edge from vertex a to vertex b iff $a \rightsquigarrow b$.

Theorem 2 (spawn principle) *If rules b_1, b_2, ..., b_k are the only rules that can be potentially triggered by rule a, then*

$$\omega(a) \leq 1 + \sum_{i=1}^{k} \Omega(b_i) \tag{4}$$

Proof. By definition and Corollary 1, for any firing ϕ of a in an execution Φ at most one firing from each of $b_1, b_2, ..., b_k$ can be triggered. It follows from Lemma 5 that the weights of these triggered firings, if any, are bounded by $\Omega(b_1), \Omega(b_2), ..., \Omega(b_k)$ respectively. Since $\omega_\Phi(\phi) = 1 + \sum_j \omega_\Phi(\phi_j), \phi \rightsquigarrow \phi_j$, it follows that $1 + \sum_{i=1}^{k} \Omega(b_i)$ is a weight bound of a. □

The spawn principle can be used to derive a response-time bound for a special class of programs.

Algorithm 1 *Compute a response-time bound for an* **EQL** *program whose PT graph is acyclic.*

Input. PT graph of an **EQL** program.

Output. A response-time bound of the program if the PT graph is acyclic.

Method.

1. Keep a weight (a counter) for each vertex.
2. Topologically sort the vertices in PT graph. If the PT graph is cyclic, stop.
3. Pick one vertex at a time, in reverse topological order, and set its weight to 1 plus the sum of its children weights.
4. Repeat step 3 until all vertex weights are set.
5. Report the sum of all vertex weights as the response-time bound, then stop.

Lemma 6 *The vertex weight computed in step 3 of Algorithm 1 is a weight bound of the corresponding rule.*

Proof. Since the vertices are picked in reverse topological order, the first vertex picked must be a leaf in the PT graph. By the spawn principle we know that 1 is a weight bound of the rule. Assume that the weights computed for the first k picked vertices are weight bounds of the corresponding rules. When the next vertex is picked all its children have been assigned weights already, due to the reverse topological order. From the induction hypothesis and by the spawn principle it follows that the weight of this vertex computed in step 3 is a weight bound of the corresponding rule. □

Theorem 3 *Given an acyclic PT graph $G = (V, E)$ of an* **EQL** *program, Algorithm 1 computes a response-time bound for the program in time $O(|V| + |E|)$.*

Proof. By Theorem 1 if follows that the sum of the weights of all the rules in an execution is a bound on the number of total firings. Thus the sum of the weight bounds of all the rules is a response-time bound of the program. From the previous lemma we know the sum reported in step 5 is a response-time bound.

As for the time complexity, each of steps 1 and 5 takes $O(|V|)$ time. Step 2 can be done in $O(|V| + |E|)$ time. (See [4], for example.) Each vertex is picked once in step 3 and for each vertex its every outgoing edge is traversed once. Thus steps 3 and 4 take $O(|V| + |E|)$ time. It follows that $O(|V| + |E|)$ is the time complexity of the algorithm. □

The spawn principle derives a weight bound for a rule r by assuming that for each firing of r all the children of r in the PT graph will be triggered and then fired before the next firing of r. However, this is not always true. In some cases the children have mutually exclusive enabling conditions, thus at most one of them can be enabled (and triggered). In some other cases firing one rule among the children disables the other rules, thus at most one among the triggered children will be fired. Based on these observations we usher in another important rule relation.

Rule a *suppresses* rule b at state s if

1. $a \notin F_s$, or
2. $b \notin F_s$, or
3. $b \in F_s - F_{A_a(s)}$.

Rule a *suppresses* rule b, denoted by $a \oslash b$, if at every reachable state a suppresses b.

A *suppression group* is a set of rules each of which suppresses every other rule in the group.

Lemma 7 *If rule a is a potential trigger of every rule in a suppression group, then for each firing ϕ of a at most one firing in the group can be triggered by ϕ.*

Proof. Let $\langle \phi_1(s_1, f_1), \phi_2(s_2, f_2), ..., \rangle$ be a program execution. And let $b_1, b_2, ..., b_k$ be the rules of a suppression group. Suppose $\phi_m(s_m, b_i)$ and $\phi_n(s_n, b_j)$, $1 \leq i, j \leq k$, are both triggered by $\phi_l(s_l, a)$. By definition, $b_i \in F_{s_t}$ for $l < t \leq m$, and $b_j \in F_{s_t}$ for $l < t \leq n$. Without losing generality we assume $m < n$. This implies $b_j \in F_{s_m}$ and $b_j \in F_{s_{m+1}}$. However, since $b_i \oslash b_j$, it follows from definition that either $b_i \notin F_{s_m}$, or $b_j \notin F_{s_m}$, or $b_j \notin F_{s_{m+1}}$. Any of the 3 situations is contradictory. □

Theorem 4 (select principle) *If rules b_1, b_2, ..., b_k form a suppression group, and are the only rules that can be potentially triggered by rule a, then*

$$\omega(a) \leq 1 + \max_{1 \leq i \leq k}(\Omega(b_i)) \tag{5}$$

Proof. By Lemma 7, for any firing ϕ of a at most one firing can be triggered. The triggered firing, if any, is a firing among $b_1, b_2, ..., b_k$. By Lemma 5 the weight of the triggered firing is bounded by $\max_{1 \leq i \leq k}(\Omega(b_i))$. It follows that the weight of ϕ is bounded by $1 + \max_{1 \leq i \leq k}(\Omega(b_i))$. □

To make use of the select principle to get tighter bounds, step 3 of Algorithm 1 can be modified such that the children of the picked vertex are checked first to see if they form a suppression group. If they do then the select principle is applied, otherwise the spawn principle is applied. Step 5 is also similarly modified such that if the entire program is a suppression group then the maximum vertex weight is reported as the response-time bound. These modifications yield tighter bounds, while retaining similar complexity (assuming the suppression relation is also given).

The drawback of the select principle is that it is too restrictive in terms of applicability. There are cases in which only some of the children of a vertex form a suppression group, but not all of them. For these cases the select principle cannot be directly applied, yet the bounds derived by using the spawn principle can still be made tighter. Observant readers might have noticed that the spawn principle and the select principle can be combined to derive tighter bounds, as depicted below.

To get a succinct representation, for a rule set $\pi = \{r_1, ..., r_k\}$ we shall use $\Omega(\pi)$ to denote $\max_{1 \leq i \leq k}(\Omega(r_i))$.

Theorem 5 (hybrid principle) *If b_1, b_2, ..., b_k are the only rules that can potentially be triggered by rule a, and they can be partitioned into $\{\pi_1, \pi_2, ..., \pi_l\}, l \leq k$, such that every partition element is a suppression group, then*

$$\omega(a) \leq 1 + \sum_{i=1}^{l} \Omega(\pi_i) \tag{6}$$

Proof. By Lemma 7, for any firing ϕ of a in an execution Φ at most one firing from each of π_1, π_2, ..., π_l can be triggered. It follows from Lemma 5 that the weights of these triggered firings, if any, are bounded by $\Omega(\pi_1)$, $\Omega(\pi_2)$, ..., $\Omega(\pi_k)$ respectively. Since $\omega_\Phi(\phi) = 1 + \sum_j \omega_\Phi(\phi_j), \phi_i \rightsquigarrow \phi_j$, it follows that $1 + \sum_{i=1}^{l} \Omega(\pi_i)$ is a weight bound of a. □

The spawn principle and the select principle are in fact special cases of the hybrid principle. When the partition consists of only one suppression group, the hybrid principle is equivalent to the select principle. When every suppression group is a single rule, the hybrid principle degenerates into the spawn principle. Therefore the hybrid principle is more versatile in terms of applicability and output quality.

To apply the hybrid principle in the bound derivation algorithm it would be nice if we can find a "best" partition on a set of rules first, that is, a partition consisting of suppression groups that causes the right-hand side of (6) to be minimum. Unfortunately, this is a difficult problem. The NP-complete clique cover problem (also known as PARTITION INTO CLIQUES in [3]), can be reduced to this problem by assigning unit weight bound to each vertex and replacing each edge with two opposite, directed edges, given a problem instance. The original problem instance has a clique cover of size K or less iff the best partition for the transformed problem instance consists of K or less suppression groups (and thus the right-hand side of (6) is $1 + K$ or less). Thus our problem is NP-hard. Knowing this we do not enforce ourselves to find such a best partition in our implementation. Instead, a polynomial-time, greedy method is used, which finds a maximal suppression group in each iteration. Steps 3 and 5 of Algorithm 1 are then appropriately modified to make use of the hybrid principle.

4 Finding potential triggers and suppression groups

We have so far shown how potential triggers and suppression groups can be used to derive program response-time bounds. In this section we discuss how to detect these relations for a given set of rules.

Note that the two relations are both defined in terms of states. A naive approach to compute them may incur an unacceptable cost in checking every state in the state space. Since our goal is to find a reasonably tight bound on the maximum execution length rather than the exact value, we have some leeway for using approximation algorithms. In the following, we present some necessary/sufficient conditions for satisfying the relations which we shall use in approximating them. We shall of course choose an approximation only if it results in a conservative estimation of the response-time bound.

4.1 Necessary conditions for potential trigger

First we introduce some definitions.

Rule a and rule b are *mutually exclusive* if they cannot be enabled at the same time.

Rule a *potentially enables* rule b if there exists a state where (1) a is firable, (2) b is disabled, and (3) b becomes enabled after firing a.

Rule a *disables* rule b if for all states where a is firable and b is enabled, b becomes disabled after firing a.

Rule a *potentially conflicts* with rule b if there exists a state where (1) a is firable, (2) b is not conflictive, and (3) b becomes conflictive after firing a.

Rule a *dominates* rule b if for all states where a is firable and b is conflictive, b becomes not conflictive after firing a.

Lemma 8 *Let a and b be two different rules of an* **EQL** *program. Rule a is a potential trigger of rule b only if*

- *either (trigger by enabling)*

  (PT1a) *b can be disabled while a is firable, and*

  (PT1b) *a potentially enables b, and*

  (PT1c) *b remains (or becomes) conflictive after a's firing,*

- *or (trigger by conflicting)*

  (PT2a) *b can be enabled and not conflictive at the same time, and*

  (PT2b) *a and b are not mutually exclusive, and*

  (PT2c) *a potentially conflicts with b, and*

  (PT2d) *a does not disable b.*

Proof sketch. Let a and b be two different rules of an **EQL** program. By definition we know that $a \rightsquigarrow b$ only if there exists an execution in which b is triggered by some firing $\phi_i(s_i, a)$. It follows that $b \in F_{A_a(s_i)} - F_{s_i}$, that is, b is unfirable at s_i and becomes firable after ϕ_i. Thus we have either of the following situations:

1. b is disabled at s_i, and becomes enabled and conflictive after ϕ_i: It would be contradictory if any of the PT1 conditions does not hold.

2. b is enabled but not conflictive at s_i, and becomes conflictive and remains enabled after ϕ_i: This cannot happen if any of the PT2 conditions is false. □

Lemma 9 *Rule a is a potential self-trigger only if*

1. *a does not disable itself, and*

2. *its LHS variables and RHS variables are disjoint.*

Proof. By definition we know that $a \rightsquigarrow a$ only if there exists an execution in which a is triggered by some firing $\phi_i(s_i, a)$. It follows by definition that $a \notin F_{s_i} - F_{A_a(s_i)}$. Thus a is firable at s_i and stays firable at $A_a(s_i)$. If a disables itself then it cannot be enabled at $A_a(s_i)$. If a's LHS and RHS do not have common variables then it cannot be conflictive at $A_a(s_i)$. In either case a cannot stay firable at $A_a(s_i)$. □

4.2 Sufficient conditions for suppression

Lemma 10 *Rule a suppresses rule b if*

1. *a and b are mutually exclusive, or*

2. *a disables b, or*

3. *a dominates b.*

Proof. At any state s conditions 1 implies that either $a \notin F_s$ or $b \notin F_s$. If $a \in F_s$ and $b \in F_s$ and a is fired at s, condition 2 implies that b is disabled after the firing; while condition 3 implies that b is not conflictive after the firing. In either case $b \in F_s - F_{A_a(s)}$. □

We have thus shown that the potential-trigger and suppression relations can be approximated by testing other finer conditions, such as potential enableness and mutual exclusion. These conditions themselves can be efficiently approximated. Most of them have been implemented earlier for our another analysis tool [8].

5 Analysis of a collision avoidance system

Based on the bound derivation principles and the approximation conditions presented in this paper we have implemented Response-Time Analyzer for **EQL** programs. To demonstrate the utility of this analyzer, we present as an example the analysis of a real-time rule-based program, the CAS (Collision Avoidance System).

The CAS application was presented in [6] and was originally written in CROPS5. Carried on board of an airplane, CAS listens to the transponder signals from other planes. This information is used by CAS to compute various useful information for generating appropriate advice to the pilot so as to avoid possible collision with nearby planes. The run-time environment consists of "streams", each of which has a private working memory dedicated to one nearby plane. To reduce execution-time variance, the program execution is divided into two phases. In the first phase streams are selected to run in a round-robin fashion to determine the potential danger of collision posed by the nearby planes; a priority is assigned to each plane according to its potential threat. Streams are then scheduled to run their second phase according to their assigned priority.

To facilitate our analysis, the CROPS5 rules were first translated into **MRL**[1] rules [7], and then expanded into **EQL** rules. The 7-rule **EQL** program corresponding to a stream in phase one is given in Appendix A. When the response-time analyzer is invoked to analyze the program it first determines the various (approximated) rule relations:

```
Potential enableness relation:
   1 -> 2                      2 -> 3 4 5 6 7
   5 -> 7                      6 -> 7
Disableness relation:
   1 -> 1 3 4 5 6 7            2 -> 1 2
   3 -> 1 2 3 4 5 6 7          4 -> 1 2 3 4 5 6 7
   5 -> 3 4 5 6                6 -> 3 4 5 6
   7 -> 1 2 3 4 5 6 7
Mutual exclusion relation:
   1 -> 2 3 4 5 6 7            2 -> 1 3 4 5 6 7
   3 -> 1 2 5 6 7              4 -> 1 2 5 6 7
   5 -> 1 2 3 4 6 7            6 -> 1 2 3 4 5 7
   7 -> 1 2 3 4 5 6
Dominance relation:
   1 -> 1                      2 -> 2
   3 -> 3 4 7                  4 -> 3 4 7
   5 -> 5                      6 -> 6
   7 -> 7
Conflicting assignment relation:
   1 -> 2 3 4 7                2 -> 1 3 4 7
   3 -> 1 2 5 6                4 -> 1 2 5 6
   5 -> 3 4 6                  6 -> 3 4 5
   7 -> 1 2
Suppression relation:
   1 -> 1 2 3 4 5 6 7          2 -> 1 2 3 4 5 6 7
   3 -> 1 2 3 4 5 6 7          4 -> 1 2 3 4 5 6 7
   5 -> 1 2 3 4 5 6 7          6 -> 1 2 3 4 5 6 7
   7 -> 1 2 3 4 5 6 7
Potential trigger relation:
   1 -> 2                      2 -> 3 4 5 6 7
   5 -> 7                      6 -> 7
```

With the potential trigger and the suppression relations, the weight bounds for the rules are then computed according to the hybrid principle. The output shows that, for example, the weight bound of rule 1 is 4.

```
Rule weight bounds: 4 3 1 1 2 2 1
```

Finally the hybrid principle is applied again to compute a response-time bound for the program. Since the entire program is a suppression group, the select principle is also applicable, which means the largest weight bound of the program rules is a response-time bound.

```
The program always reaches a fixed point in 4 rule
  firings.
```

The analysis time on a 33 MHz 486 IBM[2] PC-compatible is less than 1 second. By manual analysis we know that the reported bound is optimal. The longest firing sequence is either $\langle 1, 2, 5, 7\rangle$ or $\langle 1, 2, 6, 7\rangle$.

[1] **MRL** (Macro Rule-based Language) is an extension of **EQL**.

[2] IBM is a registered trademark of International Business Machines Corporation.

6 Conclusion

In this paper, we study the response-time analysis for equational rule-based programs. Because of the inherent undecidability of the general problem, we focus our analysis on special cases where response-time bounds can be efficiently computed. Two rule relations, potential-trigger and suppression, are shown to be closely related to program response time. A principle is presented, which makes use of the two relations to derive a response-time bound for any program with an acyclic potential-trigger relation. Refinement of the necessary or sufficient conditions for the two relations are then exploited so that the two relations can be efficiently approximated, subject to the requirement that the integrity of the derived bounds must be preserved. A response-time analyzer based on the theory in this paper has been implemented, and an analysis example is given to illustrate the practicality of our technique. In this example, the state space is infinite and thus cannot be analyzed by direct, brute-force methods such as using state-space checker. Our response-time analyzer takes only seconds to derive a bound on the program response time. By manual analysis, we can show that the reported bound is tight.

Ongoing research topics include improving the bound-derivation algorithm to get tight bounds for wider classes of programs, and adapting the research results to more expressive rule-based languages such as **MRL**. Lastly, it should be mentioned that our approximation techniques also seem to be useful in reducing the computation time for model checking of state-space structures.

References

[1] Browne, J. C., Cheng, A. M. K. and Mok, A. K., "Computer-Aided Design of Real-Time Rule-Based Decision Systems," to appear in *IEEE Transactions on Software Engineering*.

[2] Cheng, Albert M. K., "**EQL**: A Rule-Based Language for Programming Real-Time Decision Systems," technical report, Dept. of Computer Sciences, Univ. of Texas at Austin, March 1989.

[3] Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[4] Knuth, D. E., *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*, 2nd Ed., Addison-Wesley, 1973.

[5] Mok, Aloysius K., "Formal Analysis of Real-Time Equational Rule-Based Systems," *Proceedings of*

the *10th Real-Time Systems Symposium*, December 1989.

[6] Paul, C. J., Acharya, A., Black, B. and Strosnider, J. K., "Reducing Problem-Solving Variance to Improve Predictability," *CACM*, August 1991.

[7] Tsou, Duu-chung, "Translating the Collision Avoidance System Program into MRL," technical report, Dept. of Computer Sciences, Univ. of Texas at Austin, December 1991.

[8] Wang, Rwo-Hsi, *A Customizable Fixed-Point Convergence Analyzer for Real-Time Rule-Based Systems*, Master's thesis, Dept. of Computer Sciences, Univ. of Texas at Austin, December 1991.

A Phase one of the collision avoidance system

```
PROGRAM cas_1;
CONST
   check_for_data   = 0;    manage_global_adv   = 1;
   reset_global_adv = 2;    reset_local_adv     = 3;
   set_sched_pri    = 4;    generate_local_adv = 5;
   switch_streams   = 6;    set_global_adv      = 7;
   inactive = 0;     active  = 1;     notified = 2;
   danger   = 0;     invalid = 1;     ignore   = 2;
   none     = 0;     climb   = 1;     dive     = 2;
   valid    = 0;     left    = 1;     right    = 2;
VAR
   stream_id, goal_state, goal_status,
   range_plane_id, range_range, range_delta_range,
   range_xy_range, range_z_range, alert_plane_id,
   alert_status, alert_pri, adv_alt_plane_id,
   adv_alt_adv, adv_alt_adv_status,
   adv_turn_plane_id, adv_turn_adv,
   adv_turn_adv_status, appr_plane_id,
   appr_proximity, appr_ave_velocity,
   appr_time_to_pass : INTEGER;
RULES
(* 1: new_pos:calculate_new_parameters *)
   goal_state := reset_local_adv IF
      goal_state        = check_for_data  AND
      goal_status       = active

(* 2: new_pos:reset_adv *)
[] goal_state := set_sched_pri !
   goal_status := active !
   adv_alt_adv_status := invalid !
   adv_turn_adv_status := invalid IF
      goal_state        = reset_local_adv AND
      goal_status       = active          AND
      adv_alt_plane_id  = stream_id       AND
      adv_turn_plane_id = stream_id

(* 3: too_far_away:xy:set_sched_pri *)
[] goal_state := generate_local_adv !
   goal_status := active !
   alert_status := ignore !
   alert_pri := 200000 IF
      goal_state        = set_sched_pri   AND
      goal_status       = active          AND
      range_plane_id    = stream_id       AND
      range_xy_range    > 26400           AND
      alert_plane_id    = stream_id       AND
      alert_status      = invalid

(* 4: too_far_away:z:set_sched_pri *)
[] goal_state := generate_local_adv !
   goal_status := active !
   alert_status := ignore !
   alert_pri := 200000 IF
      goal_state        = set_sched_pri   AND
      goal_status       = active          AND
      range_plane_id    = stream_id       AND
      range_z_range     > 2000            AND
      alert_plane_id    = stream_id       AND
      alert_status      = invalid

(* 5: danger_zone:calc_pri: promixity < 2000 *)
[] alert_status := danger !
   alert_pri := 10 * appr_time_to_pass +
      appr_proximity / appr_ave_velocity IF
      goal_state        = set_sched_pri   AND
      goal_status       = active          AND
      range_plane_id    = stream_id       AND
      range_z_range     <= 2000           AND
      range_xy_range    <= 26400          AND
      alert_plane_id    = stream_id       AND
      alert_status      = invalid         AND
      appr_plane_id     = stream_id       AND
      appr_proximity    < 2000

(* 6: danger_zone:calc_pri: promixity >= 2000 *)
[] alert_status := danger !
   alert_pri := 100000 + appr_time_to_pass IF
      goal_state        = set_sched_pri   AND
      goal_status       = active          AND
      range_plane_id    = stream_id       AND
      range_z_range     <= 2000           AND
      range_xy_range    <= 26400          AND
      alert_plane_id    = stream_id       AND
      alert_status      = invalid         AND
      appr_plane_id     = stream_id       AND
      appr_proximity    >= 2000

(* 7: danger_zone:set_sched_pri *)
[] goal_state := generate_local_adv !
   goal_status := active IF
      goal_state        = set_sched_pri   AND
      goal_status       = active          AND
      alert_plane_id    = stream_id       AND
      alert_status      = danger
END.
```