Self-Stabilizing Real-Time OPS5 Production Systems[1]

Albert M. K. Cheng and Seiya Fujii

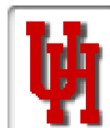
Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

UH-CS-04-07
November 15, 2004

Keywords: rule-based systems, real-time systems, self-stabilization, OPS5, fault tolerance

Abstract

We examine the task of constructing bounded-time self-stabilizing rule-based systems that take their input from an external environment. Bounded response-time and self-stabilization are essential for rule-based programs that must be highly fault-tolerant and perform in a real-time environment. We present an approach for solving this problem using the OPS5 programming language as it is one of the most expressive and widely used rule-based programming languages. Bounded response-time of the program is ensured by constructing the state space graph so that the programmer can visualize the control flow of the program execution. Potential infinite firing sequences, if any, should be detected and the involved rules should be revised to ensure bounded termination. Both the input variables and internal variables are made fault tolerant from corruption caused by transient faults via the introduction of new self-stabilizing rules in the program. Finally the timing analysis of the self-stabilizing OPS5 program is shown in terms of the number of rule firings and the comparisons performed in the Rete network.



[1]This material is based upon work supported in part by the National Science Foundation under Award No. IRI-9526004.

Self-Stabilizing Real-Time OPS5 Production Systems *

Albert M. K. Cheng and Seiya Fujii
Real-Time Systems Laboratory
Department of Computer Science
University of Houston
Houston, Texas 77204-3010, USA

Abstract

We examine the task of constructing bounded-time self-stabilizing rule-based systems that take their input from an external environment. Bounded response-time and self-stabilization are essential for rule-based programs that must be highly fault-tolerant and perform in a real-time environment. We present an approach for solving this problem using the OPS5 programming language as it is one of the most expressive and widely used rule-based programming languages. Bounded response-time of the program is ensured by constructing the state space graph so that the programmer can visualize the control flow of the program execution. Potential infinite firing sequences, if any, should be detected and the involved rules should be revised to ensure bounded termination. Both the input variables and internal variables are made fault tolerant from corruption caused by transient faults via the introduction of new self-stabilizing rules in the program. Finally the timing analysis of the self-stabilizing OPS5 program is shown in terms of the number of rule firings and the comparisons performed in the Rete network.

Index terms: rule-based systems, knowledge-based systems, expert systems, production systems, real-time, OPS5, self-stabilization, fault tolerance.

*This material is based upon work supported in part by the National Science Foundation under Award No. IRI-9526004. This paper is an extended and revised version of a preliminary work presented at IEEE IPDPS 2000.

1 Introduction

Hard real-time systems are categorized as ones that guarantee the correct logical computation in a specified amount of time. Examples are flight control systems, command and control systems, process control systems, flexible manufacturing applications, the space shuttle avionics system, the space station, and space-based defense systems, and usually missing a single deadline may cause disastrous consequences. These kind of systems are very complex, and require a high degree of fault tolerance. Hence, it is necessary for the embedded system to tolerate transient faults and recover automatically. The notion of self-stabilization was introduced by Dijkstra [33, 34]. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps."

Self-stabilization has been studied extensively to make distributed systems [5, 6, 8, 40, 47, 48, 58, 63], special classes of graph algorithms such as spanning trees [2, 11, 47], load-balancing [43] and connected components/matching [57, 56], and networks [1, 10, 14, 16, 39, 41, 42, 50] more robust. A comprehensive bibliography on stabilization can be found in [53]. Although a number of researchers address timing issues such as clock synchronization [4, 7, 18, 29, 30, 31, 37, 38, 39] and time complexities of self-stabilization [2, 9, 36, 35], there is little work on predicting and guaranteeing in absolute (standard or wall clock) time an upper bound on stabilization. The area of bounded-execution-time rule-based systems is even less explored.

Many techniques have been developed for the verification of rule-based systems [64, 65], but few consider timing correctness other than termination [12]. For instance, Rosenwald and Liu [64] propose a method to validate a rule-based system by automatically identifying the equivalence classes for fundamental tasks to help determine incorrect knowledge and rule inconsistencies, but they do not consider the execution time of the expert system. Schmolze [65] describes a decidable method for terminating rule sets to remove redundant rules without timing considerations. This is performed at pre-runtime and is based on term rewrite semantics. Also recently, Baralis, Ceri, and Paraboschi [13, 60] have developed a new approach, similar to our earlier work [66] cited in their paper, to detect termination of rules in active database systems. Their approach combines compile-time static termination analysis and runtime detection of infinite rule executions caused by cyclic rule firings. However, their technique cannot determine the execution time of rules and no optimization strategies are presented to meet timing con-

straints. Also, there is a difference between rulebases and databases in that a single invocation (execution) of a rule-based program consists of many matching (querying) steps [59, 55], so the working memory changes more often than in a database system. Thus, advanced and complex database query algorithms may not be directly applicable here.

In [22], we first introduced self-stabilizing real-time rule-based decision systems that react to the periodic sensor readings from the environment. For this purpose, we used EQL [21, 26, 25, 69], a zero-order-logic nondeterministic rule-based programming language, and introduced fast timing analysis algorithms. Our approach was applied to a NASA application: the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle Pressure Control System [52, 61]. While this work was a breakthrough in fault-tolerance for rule-based expert systems, there were certain restrictions in the form of the programs that can be transformed into the ones that self-stabilize. For instance, this method only applies to zero-order-logic EQL programs that assign constants to variables. One approach to overcome or at least compensate the limitations is to use a more expressive rule-based programming language.

In this paper, we show how to make a class of OPS5 [15, 32, 44] programs self-stabilize. As stated in [15], OPS5 stands for Official Production System, Version 5, and it is the most widely used language in the family of languages specifically designated to simplify rule-based programming, namely production systems. OPS5 has been widely used both in the academia and the industry. The obvious advantage of OPS5 over EQL is the difference in conflict resolution strategies. While the control flow in EQL is completely non-deterministic, in an OPS5 program, preferences may be given to some instantiations in the conflict set by their recency and specificity. This allows the program to have some control in rule firing sequences, and hence, it is possible to obtain shorter response time. This bounded response time is crucial for our real-time applications. Our goal is to ensure that the self-stabilizing versions of the OPS5 programs also terminate in bounded response time. OPS5 is an expert systems language, as well as CLIPS [49].

Implicit in the design of any system is a labeling of its states as *safe* or *unsafe*. We identify *safe* as those states that occur under the correct execution of a system. All other states are considered *unsafe*. A system is said to be self-stabilizing when regardless of its initial state, it is guaranteed to converge to a safe state in a finite number of steps. A system which is not self-stabilizing may stay in an unsafe state forever.

During the construction of the self-stabilizing version of a real-time OPS5 program, we first guarantee the bounded response time of the program. In order to analyze the timing behavior of an OPS5 program, we formalize a graphical representation of rule-based programs. The state space graph is defined to capture the control flow of the program. By using this graph, we can identify the possible infinite cycles of the execution in the program. After the successful timing analysis, the self-stabilization technique is applied to the program while ensuring its bounded response time. There are still some restrictions (described later) on the form or style of the program that might seem quite restrictive. However, we believe that they are necessary and reasonable conditions for the programs that react to the external environment.

Our earlier work [27] introduces a framework for self-stabilization of OPS5 programs but does not verify their logical, timing, and fault-tolerance properties. This paper provides formal proofs for these properties and describes additional details on the timing analysis. Furthermore, the paper puts this framework in perspective with related work. The remainder of this paper is organized as follows. A brief review of OPS5 production systems and the Rete matching algorithm is given in Section 2. Section 3 explains the class of OPS5 programs which can be transformed into the one that self-stabilizes, and the use of state space graph to ensure the bounded response time of the programs. Section 4 shows the techniques to convert an OPS5 program to the one that self-stabilizes. The timing analysis is given is Section 5 to determine the upper bound of the response time of the self-stabilizing OPS5 program. Finally, Section 6 concludes this paper.

2 OPS5 Programming Language

This section briefly introduces OPS5, one of the most powerful production-system languages. The production-system model has been used to solve applications in the areas of artificial intelligence, expert systems, and cognitive psychology.

2.1 Overview

An OPS5 rule-based program consists of a finite set of rules (called productions) each of which is of the form:

```
(p rule-name
  (condition-element-1)
  (condition-element-2)
               :
  (condition-element-m)
-->
  (action-1)
  (action-2)
      :
  (action-n))
```

and a database of assertions each of which is of the form

```
(class-name     ^attribute-1  value-1
                              :
                ^attribute-p  value-p)
```

The symbol '^' means there is an attribute name following it. The set of rules is called the *production memory* (PM) and the database of assertions is called the *working memory* (WM). Each assertion is called a *working memory element* (WME). A rule has three parts:

- The name of the rule, rule-name,
- The left-hand-side (LHS), i.e., a conjunction of the condition elements each of which can be either a positive condition element or negative condition element; and
- The right-hand-side (RHS), i.e., the actions, each of which may make, modify, or delete a WME, perform I/O, or halt.

All atoms are literals unless put in variable brackets '<>'. A variable in a LHS-value is recognized as a symbol which begins with a character < and ends with a character >. The variable <v> would match the value of variable v. If a variable appears more than once in the same production, all of them must match the same value. The scope of variables is a single rule. A WME is an instance of an *element class*. An element class defines a WME structure in the same way a C data type defines the structure of entities in a C program. An element class is the template from which instances are made. It is identified by class-name and by a collection of attributes describing characteristics relevant to the entity. The following is an OPS5 rule for processing sensor information from a radar system:

```
(p radar-scan                          ; an OPS5 rule
  (region-scan1 ^sensor object) ; positive condition element
  (region-scan2 ^sensor object) ; positive condition element
  (status-check ^status normal) ; positive condition element
- (interrupt ^status on)               ; negative condition element
{ <Uninitialized-configuration> ; positive condition element
  (configuration ^object-detected 0) }
-->
  (modify <Uninitialized-configuration> ^object-detected 1))    ; action
```

If both radars (region-scan1) and (region-scan2) detect an object, the status of the radar system is normal, there is no interrupt, and the attribute object-detected in the element class configuration is 0, then assign 1 to object-detected. The notation '<name> WME' is used to name the matching WME for use in this action. Hence, '<Uninitialized-configuration>' refers to the "configuration" WME matched in the LHS. Otherwise, the number of the matching conditions in the LHS may be used in modify and delete commands. Comments are given following the semicolon ';'. When the working memory contains the WMEs

```
(region-scan1 ^sensor object)
(region-scan2 ^sensor object)
(status-check ^status normal)
(configuration ^object-detected 0)
```

but does not contain the WME (interrupt ^status on), then the above rule is said to have a successful matching. More precisely, a rule is enabled if each of its positive condition element is matched with a WME in the working memory and each of its negative condition element is not matched by any WME in the working memory. A rule firing is the execution of the RHS actions in the order they appear in the rule. The above rule fires by modifying the attribute object-detected in the element class configuration to have the value 1. A condition element can consist of value tests other than equality that a matching WME value must satisfy.

The execution of the production system is known as *the recognize-act (or MRA) cycle*. It consists of the iterative sequential operations of the followings:

1. Match: evaluates the matching of the LHS of each production with WMEs. The production with successful match is a candidate for the execution. The result of a successful match is called an *instantiation*. The set of all satisfied production instantiations are referred as the *conflict set*.

2. Conflict resolution: only one instantiation is chosen from the conflict set for the firing. If there are no productions to choose from, the execution terminates.

3. Act: The RHS of the chosen production is performed. This usually results in changing one or more WMEs.

The production system repeats the MRA cycle until the conflict set is empty or an explicit halt command is executed.

We now briefly describe the two common conflict resolution (rule selection) strategies. LEX is the simpler of the two. The first step is *refraction*, that is, all previously selected and fired rule instantiations are deleted from the conflict set unless one of the WMEs of an instantiation has been modified. The second step partially orders the remaining rule instantiations in the conflict set based on the recency of the time tags corresponding to the WMEs matching the condition elements. If no single rule instantiation dominates the conflict set following the above two steps, then step three employs the principle of specificity to partially order the rule instantiations based on the total number of tests in all conditions of a rule. If this step fails to determine a single dominant instantiation, then the final step randomly selects a rule instantiation from those remaining in the conflict set.

MEA differs from LEX in that it emphasizes the recency of a WME that matches the first condition of the rule. Following the first step in LEX, MEA compares rule instantiations based on the recency of the first condition element. If there is no dominant rule instantiation, the remaining steps of LEX are performed.

The computing power of this model of OPS5 programs with the above inference engine is Turing-complete because these programs can encode two-counter machines [21].

Every production has to be entered into the production memory. The RHS of each production is composed of a sequence of actions. As stated earlier, there are three main actions to alter the WMEs:

1. Make: used to create a new WME

2. Remove: used to delete an existing WME

3. Modify: used to update attribute-value elements

Moreover, we will use the action Build for creating new rules (Section 4.2).

2.2 The Rete Match Algorithm

Rete [44] is a widely used production match algorithm. The Rete algorithm was developed to eliminate extra work that would be performed by an unoptimized pattern matcher. There are two main optimizations in Rete: *sharing* and *state-saving*. Sharing common parts of condition elements in a single production or across different productions reduces the number of tests required to do match. State-saving accumulates partially completed matches from previous recognize-act cycles for use in future cycles. Even if the working-memory elements or *WMEs* generated in a cycle fail to match a production, the partial match is saved. Thus, if a new WME is added in a new cycle, only the new WME has to be matched; the partial match from the previous cycles is not repeated.

Of these three phases, the match phase is by far the most expensive, accounting for more than 90 percent of execution time in some experiments [45, 51, 54]. Therefore, to maximize the efficiency of an OPS5 program, a fast match algorithm is necessary. The Rete match algorithm has become the standard sequential match algorithm. A new version called Rete II was introduced in [46].

The Rete algorithm compiles the LHS patterns of the production rules into a discrimination network in the form of an augmented dataflow network [62]. The state of all matches is stored in the memory nodes of the Rete network. Since a limited number of changes are made to the working memory after the firing of a rule instantiation, only a small part of the state of all matches needs to be changed. Thus, rather than checking every rule to determine which rules are matched by the WM in each recognize-act cycle, Rete maintains a list of matched rules and determines how these matches change due to the modification of the WM by the firing of a rule instantiation. The top portion of the Rete network contains chains of tests that perform the select operations. Tokens passing through these chains partially match a particular condition element and are stored in alpha-memory nodes. The alpha-memory nodes are connected to the two input nodes that find the partial binding between condition elements. Tokens with consistent variable bindings are stored in beta-memory nodes. At the end of two-input nodes are the terminal nodes, which signify that a consistent binding for a particular rule is found. The terminal nodes send the rule bindings to the conflict set. An example of a Rete network is

```
(p p1
  (c1 ^a1 <x> ^a2 10)
  (c2 ^a1 <x>)
-->
  (remove 2))
```

```
(p p2
  (c1 ^a1 <y> ^a2 10)
  (c3 ^a1 2 ^a2 <y>)
 -(c4 ^a1 <y>)
-->
  (modify 1 ^attri 4))
```

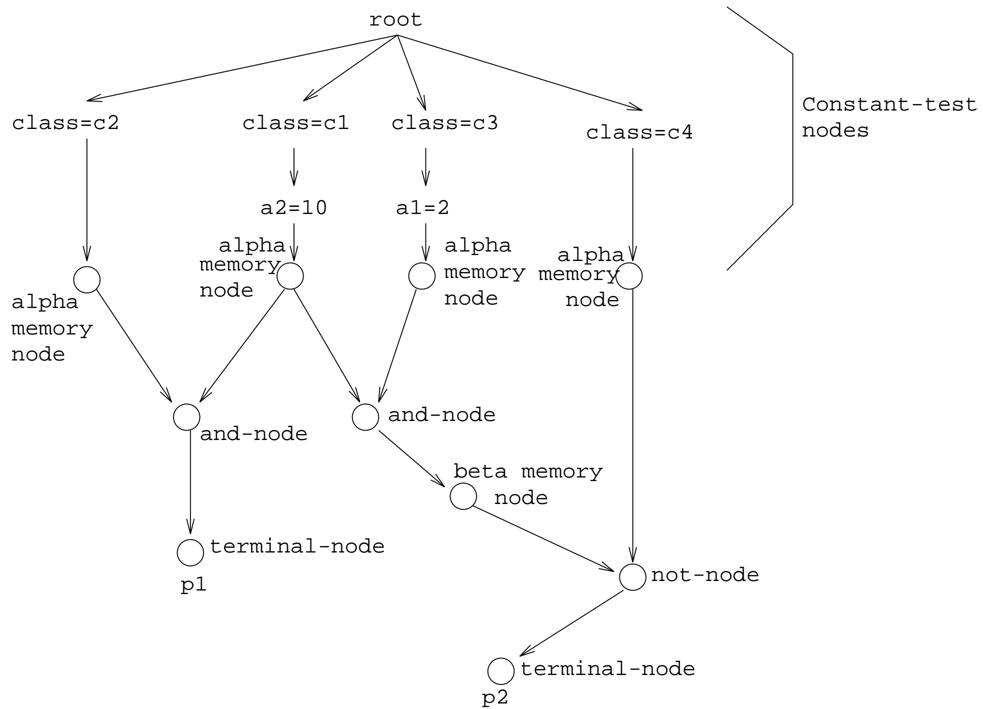


Figure 1: An example of Rete Network

shown in Figure 1.

3 Bounded-Time Analysis

In order to formalize the response time of an OPS5 program, we represent it in terms of its state space graph. By constructing the state space graph, a software analysis tool [22, 24] can identify the possible infinite loop of the control flow of the program execution. This section describes the general production model which shows the programming style for the purpose of self-stabilizing OPS5 programs first. Then, the state space graph for bounded response-time is explained next.

3.1 General Production Model

An OPS5 program implementing a real-time rule-based system has a set of productions of the form: if the left-hand-side conditions are satisfied, then execute the right-hand-side actions, where the **internalVar** and the **val** both in the (n+1)th condition element and in the action are identical. More precisely, every production in this OPS5 program has the following form:

```
(p production_name
   (condition_1)
         :
   (condition_n)
   (class_name ^internalVar <> val)
   -->
   (modify n+1 ^internalVar val)
)
```

We now consider the class of rule-based programs that satisfy the following conditions with regard to the above general production form.

1. The **internalVar** and the **val** both in the (n+1)th condition element and in the action are identical.
2. Every input variable is compared against a constant, and every internal variable is assigned a constant.

3. The condition element 1 through n does not contain any internal variables. The system is solely input dependent, and therefore, the internal variables may not affect the firings of any rules.

4. The (n+1)th condition element is called the *negation condition*. This condition guarantees that the firing of its rule will make actual changes in the WM. Refraction does not apply here because the WMEs may have the new recency number with the **modify** action.

5. Each attribute in the OPS5 program is treated as either an input variable or an internal variable. Hence, an attribute appears only once in the entire set of WMEs.

6. The input variables are not to be altered by the **modify** action. They represent the direct input from the sensor readings.

Suppose there are three internal variables $y1$, $y2$ and $y3$ in the RHS of a production to be modified when the rule is fired. Then, the negation condition of the rule must express the condition $\overline{y1} \vee \overline{y2} \vee \overline{y3}$. Since OPS5 does not have the capability to express disjunctions in either of two or more situations, the technique of *rule splitting* must be employed by separating rules for each disjunct. It is best described using an example. Consider the following possible rule before splitting which cannot be defined in OPS5, where the parallel bars denote 'or'.

```
(p p1_1
   (c1 ^a1 3 ^a2 5)
   (c2 ^a8 2)
   (c3 ^y1 <> 7 || ^y2 <> 10 || ^y3 <> 9)
   -->
   (modify 3 ^y1 7 ^y2 10 ^y3 9)
)
```

After rule splitting, we obtain three rules:

```
(p p1_1
   (c1 ^a1 3 ^a2 5)
   (c2 ^a8 2)
   (c3 ^y1 <> 7)
   -->
   (modify 3 ^y1 7 ^y2 10 ^y3 9)
)

(p p1_2
   (c1 ^a1 3 ^a2 5)
```

```
    (c2 ^a8 2)
    (c3 ^y2 <> 10)
    -->
    (modify 3 ^y1 7 ^y2 10 ^y3 9)
)

(p p1_3
    (c1 ^a1 3 ^a2 5)
    (c2 ^a8 2)
    (c3 ^y3 <> 9)
    -->
    (modify 3 ^y1 7 ^y2 10 ^y3 9)
)
```

The three productions in the above example are identical except for the third condition in each production, which was added by splitting each disjunct. This rule splitting technique does not increase the number of rule firings during the program execution because firing any one of the identical productions will falsify the remaining identical productions. Hence, there is always only one of the productions in an identical set which gets fired.

3.2 Analysis of Control Flow

The definition of the state space graph is introduced in [24]. It is restated here as follows.

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

A path in the state space graph is a sequence of distinct nodes v_1, v_2, $\ldots$, v_i, v_{i+1}, $\ldots$, 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 path corresponds to the sequence of states generated by a sequence of rule firings of the corresponding program.

Definition 2 *Rule a is said to potentially enable rule b if and only if there exists at least one reachable state in the state space graph of the program where:*

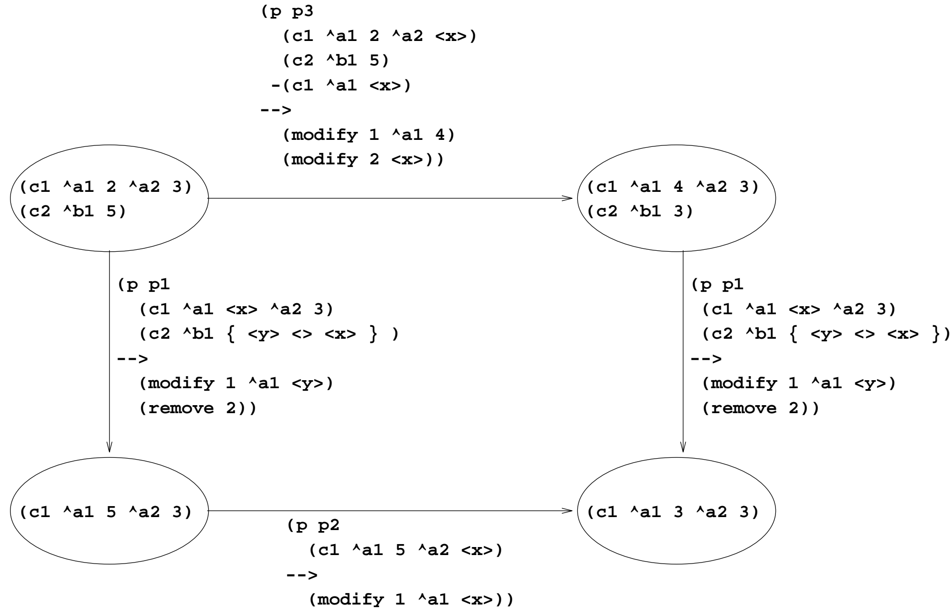


Figure 2: State space graph of an OPS5 program

(1) the enabling condition of rule b is false, and

(2) firing rule a causes the enabling condition of rule b to become true.

In Figure 2,

```
-(c1 ^a1 <x>)
```

of rule p3 means that the value of attribute a1 is not equal to the value of attribute a2 (which is 3). The state space graph in this Figure shows that rule $p1$ potentially enables rule $p2$, and rule $p3$ potentially enables rule $p1$. The `modify` action is equivalent to a `remove` action followed by a `make` action.

A node v in a state space graph is said to be a fixed point if it does not have any out-edges. Hence, if the execution of a program has reached a fixed point, then the conflict set is empty, and no rule will be fired. In Figure 2, the node labeled with

```
(c1 ^a1 3 ^a2 3)
```

is a fixed point.

A computation of a rule-based program traces a path in the state space graph. A fixed point is an endpoint of a state s if that fixed point is reachable from s. After a program reaches a fixed point, it remains there until the sensor readings are updated, and then program gets invoked again. The state space graph may not be connected. Moreover, the state space graph must be an acyclic graph. If it is not an acyclic graph, then the execution of the program may not always terminate. By detecting the possible infinite execution path, it is up to the programmer how to modify the program so that the program is always guaranteed to terminate. Suppose we have m different attributes in all classes, each attribute has n data items, and each WME is unit in the WM, then we have n^m possible WMEs. In the state space graph, there would be 2^{n^m} states.

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

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

where v1, v2 ..., and vn are either variables or constant values and each attribute can be omitted. For example,

```
(class ^a1 3 ^a2 <x>)
```

can be a symbolic form of the following WMEs.

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

Note that in order to determine with certainty whether a rule enables, rather than potentially enables, another rule, and thus determine whether the condition elements of a rule actually have a matching, would require us to know the contents of the working memory at runtime. This a

priori knowledge of the WM cannot be obtained statically. Therefore, the above conservative, approximation potentially enable relation is used instead. This approximation does not affect the validity of any of the proposed analysis techniques because the approximately potentially enable relation is a superset of the potentially enable relation [26].

Each rule's action(s) must be checked to see if they may enable another rule. Suppose there are k rules in a program. Each rule r_i has p_i WMEs and q_i actions, $i = 1, ..., k$. Let $p = p_1 + ... + p_k$ be the total number of WMEs and $q = q_1 + ... + q_k$ be the total number of actions in the program. All approximately potentially enable relations can be easily determined in polynomial time $O(pq)$.

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

Example 1 An example of a potentially enables b.

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

   (p b
     (class_a ^a1 <x> ^a2 off)
     (class_c ^c1 <y>)
   -->
     (modify 1 ^a2 open))
```

The symbolic matching method actually detects the enabling relation by checking the attribute ranges. This information can be found by analyzing the semantics of the rules, partially represented by an enable-rule (ER) graph.

ER graphs were first introduced by the first author in [20, 28]. Shortly later, triggering graphs (similar to ER graphs) in active databases were developed by Ceri and Widom in [17, 3].

Recently, Baralis, Ceri, and Paraboschi [13] have developed a new analysis approach, similar to our earlier work [66] cited in their paper, to detect termination of rules in active database systems.

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

Note that an edge from a to b in the ER graph does not mean that rule b will fire immediately after rule a. The fact that rule b is potentially enabled only implies the instantiation of rule b may be added to the conflict set to be fired. The space required to store the ER graph is polynomial, proportional to k (the number of rules in the program) with at most k^2 edges.

The previous analysis is useful since it does not require us to know the contents of working memory which cannot be obtained statically. Further details on the algorithms for detecting termination, finding enabling conditions of a cycle, and preventing cycles can be found in [24].

4 Self-Stabilization

In this section, we examine self-stabilizing OPS5 programs in the context of real-time decision systems that take their input from an external environment. A self-stabilizing program guarantees that an incorrect decision will be corrected and future decisions will be correct if no more failures occur. Different self-stabilizing techniques are used for the input variables and for the internal variables.

In general, it is not always possible to construct a self-stabilizing OPS5 program for an application. However, with some restrictions in the form of the program, it is always possible to transform a program into an equivalent one that is self-stabilizing. The bounded response-time must be guaranteed before the self-stabilizing techniques are applied.

We approach the self-stabilization of the internal variables first. This technique was originally introduced by the first author in [22] using the EQL rule-based language. It is possible to apply the same technique on OPS5 programs for the internal variables to make them self-stabilize. This earlier approach also works for OPS5 programs because it basically adds new code

that would re-initialize the internal variables if they become corrupted, regardless of whether these variables are simple (EQL) or structured (OPS5). The next subsection describes how this approach can be adapted to suit the syntax and semantics of OPS5. Then, the self-stabilization of the input variables is introduced next. It creates a new set of rules at run-time to make the program self-stabilized.

4.1 Self-Stabilization of the Internal Variables

An *initial state* s of a program p is a state in which each internal variable x_i is assigned its initial value in s. A state s of p is called *reachable* if and only if s can be reached from some initial state of p by executing a finite number of the rules of p. A program p is said to *implement* program q if and only if the following conditions hold:

I1 Programs p and q have the same input variables and the same internal variables.

I2 Each internal variable of q is an internal variable of p.

I3 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.

Two enabling conditions of rules a and b are *mutually exclusive* if and only if they cannot be true at the same time. Let A_x denote the set of attributes appearing in RHS of rule x.

Two rules a and b are said to be *compatible* if and only if at least one of the following conditions holds:

S1 Enabling conditions of the rules a and b are mutually exclusive.

S3 $A_a \cap A_b = \emptyset$.

S3 Suppose $A_a \cap A_b \neq \emptyset$. Then, for every attribute v in $A_a \cap A_b$, the same expression must be assigned to v in both rule a and b.

If a set of productions in a program p is compatible, it can be transformed into a self-stabilizing program q that implements p as follows. For every set of m productions in program p with the same attribute x in the RHS:

```
(p rule-i
   (condition-element-1)
             :
   (condition-element-n)
   (input-var-class ^x <> val-i)
   -->
   (modify n+1 ^x val-i)
)
```

where rule-1,...,rule-m are productions, we add a set of all the possible distinct productions that satisfies the following condition:

$$\overline{rule_1} \wedge \ldots \wedge \overline{rule_m}$$

In the RHS of these rules, the initial value of x is assigned to the attribute x. It is easy to show that the resulting program implements p since conditions I1, I2 and I3 are satisfied. The number of self-stabilizing rules for the internal variable x can be as many as the multiplication of the number of all the input variables in each rule which has the same internal variable on RHS. This is again because of the lack of the way to express disjunction relationship in OPS5. However, if one of them gets fired, the attribute x will have the initial value of it, and that will remove all other productions in the same self-stabilizing rule set from the conflict set.

Theorem 1 *Every production in the self-stabilizing program obtained by the above method is also compatible.*

Proof:

Each enabling conditions of the self-stabilizing production and enabling conditions of the productions i, $i = 1, \ldots, m$, are mutually exclusive (condition S1 is satisfied) because conjunction and negation cannot be true at the same time and thus the new production is compatible with every production in the corresponding non-self-stabilizing production set. This new production is compatible with every other new rule not in this non-self-stabilizing production set since the attributes on the RHS are also different and thus condition S2 is also satisfied. The above reasoning applies to every new production added to p. Hence, we can conclude that all rules in q are compatible pairwise.

□

Now we show that the program q is self-stabilizing.

Theorem 2 *Given an OPS5 program p whose productions are compatible pairwise, the transformed program q obtained by the above method is self-stabilizing.*

Proof:

Suppose q is a self-stabilizing program that implements p. We show that for each pair of distinct fixed points s_1 and s_2 of q, there is at least one input variable whose value in s_1 is different from its value in s_2. Recall that each state in the state space graph is labeled by a set of Working Memory Elements (WMEs).

Suppose s_1 and s_2 contain the set of input variables whose values are identical, while one or more of the internal variable contains different value. Then, there must be at least one internal variable y_j whose value in y_{s_1} is different from its value in y_{s_2}. Since each production fires at most once by refraction, there must be two productions p_1 and p_2 that assign different values to y_j, resulting in two distinct fixed points s_1 and s_2. However, p_1 and p_2 must be mutually exclusive (condition S1 must hold) since p_1 and p_2 in q satisfy neither condition S2 nor condition S3. If p_1 and p_2 are mutually exclusive, then there must be at least one attribute v in the LHS of two productions whose value determines which of these two productions is enabled. Therefore, v must have different values in s_1 and s_2. Since all the attributes in the LHS are the input variables, v cannot be an internal variable; otherwise, v must be a constant. Thus, there is at least one input variable whose value in s_1 is different from its value in s_2.

Suppose that s_1 and s_2 contain the set of internal variables whose values are identical, while one or more of the input variable contains different value. This is possible since two distinct initial states whose input variables are different may lead to the same fixed point. Obviously there is at least one input variable whose value in s_1 is different from its value in s_2.

□

4.2 Self-Stabilization of the Input Variables

A different self-stabilization technique is used for the input variables. Suppose there are n input variables. Then, n rules are fired first during the execution of the program to construct the

self-stabilizing productions. Hence, the only requirement for this technique is that the input variables do not corrupt during this initialization phase. All the input variables are treated as boolean (0 or 1) in this paper, but it can be extended easily to allow more than only 0 or 1 using the approach proposed in [69].

For each input variable x, create a pair of productions like these

```
(p init_x_a
   (control ^rule init ^i_x 1) ; initially ^i_x is 1
   (class ^x 1)
   -->
   (build init_x_b
      (class ^x <> 1)
      -->
      (modify 1 ^x 1)
   )
   (modify 1 ^i_x 0)
)

(p init_x_b
   (control ^rule init ^i_x 1) ; initially ^i_x is 1
   (class ^x 0)
   -->
   (build init_x_a
      (class ^x <> 0)
      -->
      (modify 1 ^x 0)
   )
   (modify 1 ^i_x 0)
)
```

The MEA strategy is enforced so that the productions that contain **(control ^rule init)** have the priority over all the rest of the productions. Hence, at the initialization phase, one of the above two productions is fired for each input variable. Another control variable **^i_x** is to ensure that the rule is fired only once.

The **build** action adds a new rule to an executing program. When either of the rule is fired, the **build** action will create a new rule using the name of the other production of the same pair. This will disable the original rule and the new one is built.

When the faults occur in the input variables during the execution, it is not likely that the self-stabilization takes place first, because there is only one condition element in each newly created self-stabilizing production. If other regular productions have higher specificity, they will

be fired before the self-stabilizing rules. In the worst case, all the rules that are instantiated by the wrong input values are fired before the self-stabilizing rules can be fired. However, we know that this occurs in bounded time because the firing of any rules will not instantiate any other rules ($LHS \cap RHS = \emptyset$). When finally the self-stabilizing rules are fired and the input variables are corrected, the new correct instantiations will be found. And, we know that with certain combination of the input variables, the program will always reach the same fixed point in bounded time. Example 2 illustrates this technique.

Example 2

```
; non-self-stabilizing rules

(literalize class1 b c)   ; input variables
(literalize class2 x1 x2) ; internal variables

(p p1
   (class1 ^b 1 ^c 1)
   (class2 ^x1 <> 1)
   -->
   (modify 2 ^x1 1)
)

(p p2
   (class1 ^b 1 ^c 0)
   (class2 ^x1 <> 1)
   -->
   (modify 2 ^x1 1)
)

(p p3
   (class1 ^c 1)
   (class2 ^x2 <> 0)
   -->
   (modify 2 ^x2 0)
)

; add the following rules to make the above program self-stabilize
; Self-stabilizing rules for the input variables

(p init_b_A
   (control ^rule init ^i_b 1)
   (class ^b 1)
   -->
   (build init_b_B
      (class ^b <> 1)
      -->
      (modify 1 ^b 1)
   )
   (modify 1 ^i_b 0)
)

(p init_b_B
   (control ^rule init ^i_b 1)
   (class ^b 0)
   -->
   (build init_b_A
      (class ^b <> 0)
      -->
      (modify 1 ^b 0)
   )
   (modify 1 ^i_b 0)
)

(p init_c_A
   (control ^rule init ^i_c 1)
   (class ^c 1)
   -->
   (build init_c_B
      (class ^c <> 1)

(p init_c_B
   (control ^rule init ^i_c 1)
   (class ^c 0)
   -->
   (build init_c_A
      (class ^c <> 0)
```

```
    -->
    (modify 1 ^c 1)
  )
  (modify 1 ^i_c 0)
)
```

```
    -->
    (modify 1 ^c 0)
  )
  (modify 1 ^i_c 0)
)
```

```
; Self-stabilizing rule for the internal variables

(p self-stable-1
  (class1 ^b <> 1)
  (class2 ^x1 <> 0)
  -->
  (modify 2 ^x1 0)
)

(p self-stable-2
  (class1 ^b <> 1 ^c <> 0)
  (class2 ^x1 <> 0)
  -->
  (modify 2 ^x1 0)
)

(p self-stable-3
  (class1 ^b <> 1 ^c <> 1)
  (class2 ^x1 <> 0)
  -->
  (modify 2 ^x1 0)
)

(p self-stable-4
  (class1 ^c <> 1)
  (class2 ^x2 <> 1)
  -->
  (modify 2 ^x2 1)
)
```

Due to space limitations, this small example is shown. This technique can be applied to large systems as well.

5 Timing Analysis

In this section, the maximum response time of the self-stabilizing OPS5 programs is analyzed. The response time of the program is investigated in two respects: the maximal number of rule firings and the maximal number of basic comparisons made by the Rete network during the program execution. The original analysis technique of the response time of a general OPS5 program is found in [19]. Because of the more restricted form of a self-stabilizing OPS5 program, it is simpler to determine the upper bound on its execution time.

5.1 The Number of Rule Firings

As discussed earlier, a self-stabilizing OPS5 program is guaranteed to terminate in a bounded number of recognize-act cycles. Hence, we are able to find its finite response time.

In general OPS5 programs, the number and WMEs can increase or decrease which can only be known at run time. However, in a self-stabilizing OPS5 program, only the **modify** actions

appear on RHS. The **modify** action does not change the number of WMEs but it only changes the attribute values.

Theorem 3 *During the execution of self-stabilizing OPS5 program, each production or rule can fire at most once. Hence, the upper bound of the number of rule firings is the number of productions in the program.*

Proof:
Since the internal variables appear only on the RHS and the input variables appear only on the LHS, any rule firings will not cause any rules to be instantiated. Only **modify** actions can appear on the RHS of any productions, and thus, they will not increase the number of matching WMEs. In fact, the self-stabilizing OPS5 program has only one WME for each class throughout its execution.

□

After the self-stabilizing technique is applied to the OPS5 program, the number of productions will increase. Suppose there are k regular rules in the program. For n internal variables, there will be at most n^k new self-stabilizing rules, and for m input variables, there will be $2m$ new self-stabilizing rules. If no transient faults occur during the execution, the upper bound of rule firings is the number of the regular rules plus m self-stabilizing rules of input variables that will be fired during the initialization phase. When there is a transient fault in the internal variables, it takes only one firing to correct one variable. For each transient-fault of input variables, the upper bound can be the additional number of all the regular rules plus the number of faults which causes the self-stabilizing rules to fire.

5.2 The Match Time

Now we compute an upper bound on the time required during the match phase in terms of the number of comparisons made by the Rete algorithm. The Rete match algorithm is explained in Section 2.2. The comparisons are made for each attribute in the LHS of the productions. One comparison is conducted by each constant test node when a token is passed to it. On the other hand, And-node may conduct many comparisons, since many pairs of tokens may have

to be checked whenever a token is passed to it. However, in self-stabilizing OPS5 program, at most one token is passed to each constant test node, and all the comparisons are made with constant value. Hence, the And-node does not need to check for any cross-reference caused by the variables (those enclosed in <>).

Based on the discussion above, we compute an upper bound on the number of comparisons made in the match phase as follows. Assume p is an n-rule self-stabilizing OPS5 program. For each rule r, let R_r represent the Rete sub-network which corresponds to the r. Let T^α denote the maximal number of comparisons made when a token of the class α is passed to R_r. To compute the value of T^α, we need to add up the number of comparisons respectively performed by individual nodes when a token of α is passed to R_r. For each node v, if v is one of the constant test nodes, the value of T^α is increased by 1. If v is a class-checking node and the received token is not of the class required, then the token is discarded by v; otherwise, a copy of this token is passed to each of v's successor(s). If v is an and-node, no comparison is made.

None of the productions in the self-stabilizing OPS5 program produces the tokens. In other words, there is no **make** or **remove** action on the RHS. Having obtained all the T^αs, we can compute, for each rule $r \in p$, an upper bound on the number of comparisons made by the network as a result of one firing of r. Let T_r denote this upper bound,

$$T_r = \sum_\alpha T^\alpha$$

For the non-self-stabilizing program p with k rules, l classes, n internal variables and m input variables, $T_r = 3(m+1)$ because, in the rule r there can be as many as m input variables in its LHS to be compared against constants. Each input variable on LHS can each belong to its own class. It takes one constant test node to check the class and two constant test nodes to check the input variable with a constant. Then, there is a negation condition as the last condition element in each production.

In the self-stabilizing version q of the program p, it takes exactly eight comparisons for each self-stabilizing rule of the input variables at the initialization phase. Each self-stabilizing rule of the internal variables contains input variables on LHS as many as the number of all the regular rules. In the worst case, each of these input variables could be in its own distinct class. Hence, for a set of k non-self-stabilizing rules that have the same input variable on their RHS, its each self-stabilizing rule will make as many as $3(k+1)$ comparisons. In the program q, each

regular rule may have the control variable as the first condition element. This will add 3 more comparisons to make.

Let T_p denote an upper bound on the number of comparisons made by the Rete network during the execution. Since the maximal number of firings by each rule is known to be 1, the T_p is equal to

$$T_p = \sum_{r \in p} T_r$$

For the self-stabilizing program q of the non-self-stabilizing program p described above, there will be $3(m+2)$ comparisons for k regular rules where m is the number of the input variables, eight comparisons in each of $2m$ self-stabilizing rules of the input variables, and $3(k+1)$ comparisons for each self-stabilizing rule of the internal variable. And, it is shown that the number of this type of rules can be as many as n^k, while the number of the comparisons in the non-self-stabilizing program p is at most $3k(m+1)$. Hence, the maximal comparisons made in the self-stabilizing rule q is

$$T_p = 3k(m+2) + 16m + 3(k+1)n^k$$

In the worst case, the maximal number of comparisons made in a self-stabilizing program is exponential.

6 Conclusion

We have focused on two systems concepts: bounded response-time and self-stabilization in the context of rule-based programs. The state space graph is used to ensure the bounded response-time. Previous work on self-stabilizing rule-based systems focused only on the transient faults in the internal variables, and it was assumed that the transient faults do not occur in the input variables. In reality, however, we cannot make such an assumption. In this paper, self-stabilization of the input variables are also considered, and with this new technique, the rule-based system can be more stable and reliable.

We have shown that in order for a terminating program to be self-stabilizing, the relation it implements must be verifiable in one step of the program. In the face of transient failures, no assumptions about the history of the program execution can be made. In real-time decision

systems, the ability to terminate in a self-stabilizing manner may be viewed as the ability to make an informed correct decision in the face of transient failures.

This kind of program is most suitable for real-time monitoring and control decision systems such as the NASA Space Shuttle application, the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle Pressure Control System shown in [22], and the NASA Mars Rover executive [67, 68]. The Space Shuttle application is invoked periodically to monitor and diagnose the condition of the Cryogenic Hydrogen Pressure System, and to make the decision for correcting the diagnosed malfunctions. The Mars Rover executive is an embedded autonomy software that interacts with the external environment by taking sensor readings and computing control decisions based on sensor readings and stored state information to ensure the safety and progress of the robotic rover.

The timing analysis gives upper bounds of the program execution in terms of the number of the rule firings and the comparisons made by the Rete network. Because of the lack of capability of representing the disjunction relations of the conditions in OPS5 program, the self-stabilizing program may contain many more rules than the original non-self-stabilizing program. This will increase the number of both the rule firings and the comparisons in the Rete network.

The conditions set for the form/style of the non-self-stabilizing rules remain restrictive, and with the expressive rule-based language like OPS5, it is almost discouraging to have such restrictions. For future work, we shall investigate the restricted use of other action commands such as **make** or **remove**.

Furthermore, we think that a software tool which automatically converts the non-self-stabilizing program to the equivalent self-stabilizing version should be investigated and implemented. At this moment, the self-stabilizing rules need to be created by the programmer, and it may be time-consuming task as the number of self-stabilizing rules could be very large. With this tool, we will test our approach on a large number of OPS5 programs and report the experimental performance results.

References

[1] Y. Afek and G. M. Brown, "Self-Stabilization of the Alternating-Bit Protocol," *Proc. of 8th Symp. on Reliable Distributed Systems*, Seattle, Washington, Oct. 1989.

[2] S. Aggarwal and S. Kutten, "Time optimal self-stabilizing spanning tree algorithm," *FSTTCS93 Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer LNCS:761, pages 400-410, 1993.

[3] A. Aiken, J. Widom, and J. M. Hellerstein, "Behavior of Database Production Rules: Termination, Confluence and Observable Determinism," *Proc. Intl. Conf. on Management of Data (SIGMOD)*, San Diego, California, 1992.

[4] A. Arora, S. Dolev, and M. G. Gouda, "Maintaining digital clocks in step," *Parallel Processing Letters*, 1:11-18, 1991.

[5] A. Arora and M. G. Gouda, "Closure and convergence: a foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, 19:1015-1027, 1993.

[6] A. Arora and M. G. Gouda, "Distributed reset," *IEEE Transactions on Computers*, 43:1026-1038, 1994.

[7] A. Arora, S. Kulkarni, and M. Demirbas, "Resettable vector clocks," *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 269-278, 2000.

[8] A. Arora and M. Nesterenko, "Unifying stabilization and termination in message-passing systems," *ICDCS01 The 21st IEEE Intl. Conf. on Distributed Computing Systems*, pages 99-106, 2001.

[9] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese, "Time optimal self-stabilizing synchronization," *STOC93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 652-661, 1993.

[10] B. Awerbuch and R. Ostrovsky, "Memory-efficient and self-stabilizing network reset," *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 254-263, 1994.

[11] B. Awerbuch, B. Patt-Shamir, and G. Varghese, "Bounding the unbounded (distributed computing protocols)," *Proceedings IEEE INFOCOM 94 The Conference on Computer Communications*, pages 776-783, 1994.

[12] F. Barachini, "Frontiers in Run-Time Prediction for the Production-System Paradigm," *AI Magazine*, Vol. 15, No. 3, pp. 47-61, Fall 1994.

[13] E. Baralis, S. Ceri, and S. Paraboschi, "Compile-Time and Runtime Analysis of Active Behaviors," *IEEE Trans. on Software Eng.*, Vol. 10, No. 3, pp. 353-370, May/June 1998.

[14] G. M. Brown, M. G. Gouda, and C.-L. Wu, "Token Systems that Self-Stabilize," *IEEE Trans. on Computers*, Vol. 38, No. 6, June 1989, pp. 845-852.

[15] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1986.

[16] J. E. Burns and J. Pachl, "Uniform Self-Stabilizing Rings," *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 2, April 1989, pp. 330-344.

[17] S. Ceri and J. Widom, "Deriving Production Rules for Constraint Maintenance," *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, Brisbane, Queensland, Australia, August 1990.

[18] S. Chandrasekar and P. K. Srimani, "A self-stabilizing algorithm to synchronize digital clocks in a distributed system," *Computers and Electrical Engineering*, 20(6):439-444, 1994.

[19] J. Chen and A. M. K. Cheng, *Predicting the Response Time of OPS5-style Production Systems.* Eleventh IEEE Conference on AI Applications, February 1995.

[20] A. M. K. Cheng, " Timing Analysis of Self-Stabilizing Programs," Technical Report, Department of Computer Sciences, University of Texas at Austin, 1988.

[21] A. M. K. Cheng, *Analysis and Synthesis of Real-Time Rule-Based Decision Systems.* Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, 1990.

[22] A. M. K. Cheng, *Self-Stabilizing Real-Time Rule-Based Systems.* Proceedings of 11th Symposium on Reliable Distributed Systems, 1992, pp. 172–179.

[23] A. M. K. Cheng and J.-R. Chen, "Response Time Analysis of OPS5 Production Systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 3, pp. 391-409, May/June 2000.

[24] A. M. K. Cheng and H. Tsai, *A Graph-Based Approach for Timing Analysis and Refinement of OPS5 Knowledge-Based Systems*, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, No. 2, pages 271-288, February 2004.

[25] J.-R. Chen and A. M. K. Cheng, "Response Time Analysis of EQL Real-Time Rule-Based Systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 1, pp. 26-43, Feb. 1995.

[26] A. M. K. Cheng, J. C. Browne, A. K. Mok, and R.-H. Wang, "Analysis of Real-Time Rule-Based Systems With Behavioral Constraint Assertions Specified in Estella," *IEEE Transactions on Software Engineering*, Vol 19, No. 9, pp. 863-885, Sept. 1993.

[27] A. M. K. Cheng, and S. Fujii, "Bounded-response-time self-stabilizing OPS5 production systems," Proceedings. 14th International, Proc. Parallel and Distributed Processing Symposium, (IPDPS 2000), pages 399-404, 1-5 May 2000.

[28] A. M. K. Cheng and C.-K. Wang, "Fast Static Analysis of Real-Time Rule-Based Systems to Verify Their Fixed Point Convergence," *Proc. 5th IEEE Conf. on Computer Assurance*, U.S. National Institute of Standards and Technology, Gaithersburg, Maryland, pp. 46-56, June 1990.

[29] G. Ciardo and C. Lindemann, "Comments on 'Analysis of self-stabilizing clock synchronization by means of stochastic Petri nets'," 1995.

[30] A. Ciuffoletti, "Using simple diffusion to synchronize clocks in a distributed system," *Proceedings of IEEE 14th International Conference on Distributed Computing Systems*, pages 484-491, 1994.

[31] A. Ciuffoletti, "Self-stabilizing clock synchronization in a hierarchical network," *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 86-93, 1999.

[32] T. Cooper and N. Wogrin, *Rule-based Programming with OPS5*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

[33] E. W. Dijkstra, EWD391 *Self-stabilization in spite of distributed control.* Reprinted in Selected Writings on Computing: A personal Perspective, Springer-Verlag, Berlin, 1982, pp. 41–46.

[34] E. W. Dijkstra, *Self stabilizing systems in spite of distributed control.* Communications of the ACM, 17:643–644, 1974.

[35] S. Dolev, "Optimal time self-stabilization in uniform dynamic systems," *6th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 25-28, 1994.

[36] S. Dolev, A. Israeli, and S. Moran, "Self-stabilization of dynamic systems assuming only read/write atomicity," *Distributed Computing*, 7:3-16, 1993.

[37] S. Dolev and J. L. Welch, "Wait-free clock synchronization," *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pages 97-107, 1993.

[38] S. Dolev and J. L. Welch, "Self-stabilizing clock synchronization in the presence of byzantine faults," *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 9.1-9.12, 1995.

[39] S. Dolev, "Possible and impossible self-stabilizing digital clock synchronization in general graphs," *Journal of Real-Time Systems*, 12(1):95-107, 1997.

[40] S. Dolev and T. Herman, "Superstabilizing protocols for dynamic distributed systems," *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.

[41] M. Flatebo and A. K. Datta, "Two-state self-stabilizing algorithms for token rings," *IEEE Transactions on Software Engineering*, 20:500-504, 1994.

[42] M. Flatebo, A. K. Datta, and A. A. Schoone, “Self-stabilizing multi-token rings,” *Distributed Computing*, 8:133-142, 1994.

[43] M. Flatebo, A. K. Datta, and B. Bourgon, “Self-stabilizing load balancing algorithms,” *IEEE 13th Annual International Pheonix Conference on Computers and Communications*, 1994.

[44] C. L. Forgy, *OPS5 Users Manual.* Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, 1981.

[45] C. L. Forgy, “Rete: A fast algorithm for many pattern/many object pattern match problem,” *Artif. Intell.*, 19, 1982.

[46] C L. Forgy, “The OPS Languages: An Historical Overview,” *PC AI*, Sep. 1995.

[47] S. Ghosh, “Binary Self-Stabilization in Distributed Systems,” *Information Processing Letters*, Vol. 40, Nov. 1991, pp. 153-159.

[48] S. Ghosh, “Agents, distributed algorithms, and stabilization,” *Computing and Combinatorics* (COCOON’2000), Springer LNCS:1858, pages 242-251, 2000.

[49] J. Giarratano and G. Riley, Expert systems: Principles and Programming. PWS Pub, 1994.

[50] M. G. Gouda and N Multari, “Stabilizing communication protocols,” *IEEE Transactions on Computers*, 40:448-458, 1991.

[51] A. Gupta, “Parallelism in Production Systems,” PhD thesis, Carnegie-Mellon University, 1985.

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

[53] T. Herman, “A Comprehensive Bibliography on Self-Stabilization,” Working Paper in the Chicago Journal of Theoretical Computer Science, 2002.

[54] T. Ishida, “Parallel rule firing in production systems,” *IEEE Transactions on Knowledge and Data Engineering*, 3(1), March 1991.

[55] J. A. Kang and A. M. K. Cheng, “Reducing Matching Time for OPS5 Production Systems,” *COMPSAC*, Chicago, IL, pp. 429-434, 2001.

[56] M. H. Karaata and K. A. Saleh, “A distributed self-stabilizing algorithm for finding maximum matching,” *Computer Systems Science and Eng.* 15(3):175-180, 2000.

[57] M. H. Karaata and F. Al-Anzi, “A dynamic self-stabilizing algorithm for finding strongly connected components,” *PODC99 Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 276, 1999.

[58] S. Katz and K. Perry, "Self-Stabilizing Extensions for Message-Passing Systems," *Proc. 9th Annual Symp. on Principles of Distributed Computing*, 1990, pp. 91-101.

[59] P.-Y. Lee and A. M. K. Cheng, "HAL: A Faster Match Algorithm," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 14, No. 5, Sept./Oct. 2002.

[60] M. Lin, "Timing Analysis of PL Programs," *Proc. 24th IFAC/IFIP Workshop on Real-Time Prog. & 3rd Intl. Workshop on Active & Real-Time Database Systems*, Saarland, Germany, May-June 1999.

[61] C. A. Marsh, *The ISA Expert System: A Prototype System for Failure Diagnosis on the Space Station.* MITRE Report, The MITRE Corporation, Houston, Texas, 1988.

[62] D. P. Miranker, *TREAT: A new and efficient algorithm for AI production systems.* PhD thesis, Columbia University, 1987.

[63] F. Petit and V. Villain, "A space-efficient and self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems," *Euro-Par'97 Parallel Processing, Proceedings*, LNCS:1300, pages 476-479, 1997.

[64] G. W. Rosenwald and C.-C. Liu, "Rule-Based System Validation through Automatic Identification of Equivalence Classes," *IEEE Trans. on Knowledge & Data Eng.*, vol. 9, No. 1, pp. 24-31, Jan./Feb. 1997.

[65] J. G. Schmolze, "Detecting Redundant Production Rules," *Proc. 14th AAAI Conf.* 1997.

[66] H.-Y. Tsai and A. M. K. Cheng, "Termination Analysis of OPS5 Expert Systems," *Proc. 12th National Conf. on Artificial Intelligence (AAAI)*, Seattle, WA, pp. 193-198, Aug. 1994.

[67] R. Washington, "Autonomous Rovers for Mars Exploration," *Proc. IEEE Aerospace Conf.*, Aspen, CO, 1999.

[68] R. Washington, "On-Board Real-Time State and Fault Identification for Rovers," *Proc. IEEE Intl. Conf. on Robotics and Automation*, 2000.

[69] B. Zupan and A. M. K. Cheng, "Optimization of Rule-Based Systems Using State Space Graphs," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 2, pp. 238-254, March/April 1998.