Bounded-Response-Time Self-Stabilizing OPS5 Production Systems *

Albert Mo Kim Cheng and Seiya Fujii
Real-Time Systems Laboratory
Department of Computer Science
University of Houston
Houston, Texas 77204-3475
E-mail:{cheng,fujii}@cs.uh.edu

Abstract

This paper examines 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, and any possible infinite execution loops should be detected. 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. Introduction

Hard real-time systems must guarantee correct logical computation in a specified amount of time. Examples are flight control systems, command and control systems, process control systems, flexible manufacturing applications, space station, and space-based defense systems, and usually missing a single deadline may cause disastrous consequences. These 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 [7, 8]. 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."

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

In [4], we introduced self-stabilizing real-time EQL decision systems that react to the periodic sensor readings from the environment. EQL [3] is a zero-order-logic non-deterministic rule-based programming language. This work was applied to a NASA application: the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle Pressure Control System [10, 11]. systems, There were certain restrictions in the form of the programs that can be transformed into the ones that self-stabilize. 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 programs self-stabilize. 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 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 shorten response time. Our goal is to ensure the self-stabilizing versions of the OPS5 programs that also terminates in bounded response time.

In any system, 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 required in 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.

The remainder of this paper is organized as follows. A brief review of OPS5 production systems 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 in 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 [1, 6, 9]. The production-system model has been used to solve applications in the areas of artificial intelligence, expert systems, and cognitive psychology.

A production in OPS5 has the following general form:

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

The condition part is called *LHS* (left hand side) of the production, and the action part is called *RHS* (right hand side) of the production. The execution of the production system is known as *the recognize-act cycle* with following iterative sequential operations:

1. Match: evaluate 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 to 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.

There are three main actions to alter the WMEs: 1) Make: create a new WME; 2) Remove: delete an existing WME; and 3) Modify: update attribute-value elements.

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, the programmers can identify the possible infinite loop of the control flow of the program execution. This technique was introduced in [4, 5].

In order to implement self-stabilizing real-time rule-based systems using the OPS5 language, every production must have the following form:

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

We now consider the class of rule-based programs that satisfy the following conditions.

1. The **internal_var** 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 *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 appear 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 two internal variables a and b in the RHS of the production to be modified when the rule is fired. Then, the negation condition of the rule must express the condition $\overline{a} \vee \overline{b}$. Since OPS5 does not have capability to express disjunctions in either of two or more situations. Hence, the technique of *rule splitting* must be employed by separating rules for each disjunct. It is best described using an example:

```
(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)
```

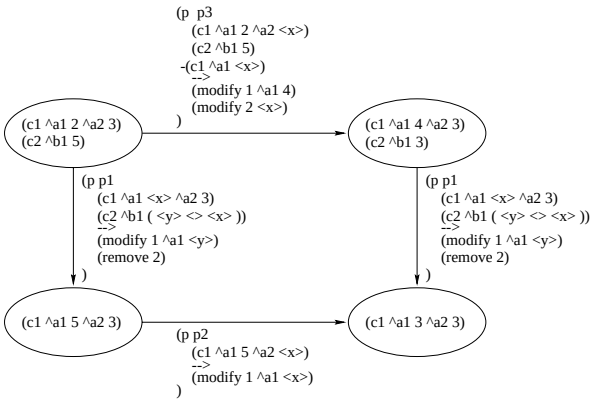

Figure 1. Sample State space graph

```
   (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 state space graph in Figure 1 shows rule $p1$ potentially enables rule $p2$, and rule $p3$ potentially enables rule $p1$.

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. 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 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 [4] using EQL rule-based language. It is possible to apply the same technique on OPS5 programs for the internal variables to make them self-stabilize. 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-stabilizes.

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* iff 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 iff 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* iff 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* iff at least one of the following conditions holds:

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

S2 $A_a \cap A_b = \emptyset$.

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 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 i = 1, ..., m 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 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.*

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

Due to space limitations, the proofs are omitted here but will appear later in the journal version.

4.2. Self-Stabilization of the Input Variables

The 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.

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

```
(p init_x_a
  (control ^rule init ^i_x 1) ; ini-
tially ^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) ; ini-
tially ^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.

Small example

```
; non-self-stabilizing rules

(literalize class1 b c)   ; input variables
(literalize class2 a1 a2) ; internal variables

(p p1                          (p p2
  (class1 ^b 1 ^c 1)             (class1 ^b 1 ^c 0)
  (class2 ^a1 <> 1)              (class2 ^a1 <> 1)
  -->                            -->
  (modify 2 ^a1 1)               (modify 2 ^a1 1)
)                              )

(p p3
   (class1 ^c 1)
   (class2 ^a2 <> 0)
   -->
   (modify 2 ^a2 0)
)

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

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

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

; Self-stabilizing rule for the internal var.

(p self-stable-1             (p self-stable-2
  (class1 ^b <> 1)             (class1 ^b <> 1 ^c <> 0)
  (class2 ^a1 <> 0)            (class2 ^a1 <> 0)
  -->                          -->
  (modify 2 ^a1 0)             (modify 2 ^a1 0)
)                            )

(p self-stable-3               (p self-stable-4
  (class1 ^b <> 1 ^c <> 1)       (class1 ^c <> 1)
  (class2 ^a1 <> 0)              (class2 ^a2 <> 1)
  -->                            -->
  (modify 2 ^a1 0)               (modify 2 ^a2 1)
)                              )
```

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 [2]. Because of the more restricted form of a self-stabilizing 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. 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.*

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 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 the 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 self-stabilizing rule q is

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

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. In the previous work of 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 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 application, the Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle Pressure Control System, as shown in [4]. It 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 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 at Rete network.

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

Furthermore, we think that the program 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.

References

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

[2] J. Chen and A. M. K. Cheng, *Predicting the Response Time of OPS5-style Production Systems*. 11th IEEE Conf. on A.I. Applications, Feb. 1995.

[3] A. M. K. Cheng, *Analysis and Synthesis of Real-Time Rule-Based Decision Systems*. Ph.D. Dissertation, Dept. of Computer Sciences, UT Austin, 1990.

[4] A. M. K. Cheng, *Self-Stabilizing Real-Time Rule-Based Systems*. Proc. of 11th Symp. on Reliable Distrib. Systems, 1992, pp. 172–179.

[5] A. M. K. Cheng and H. Tsai, *Timing Analysis of OPS5 Expert Systems*. Submitted for Publication. Dept. of Computer Science, Univ. of Houston, 1998.

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

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

[8] E. W. Dijkstra, *Self stabilizing systems in spite of distributed control*. CACM, 17:643–644, 1974.

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

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

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