An EFSM-Based Passive Fault Detection Approach

Hasan Ural and Zhi Xu

School of Information Technology and Engineering (SITE)
University of Ottawa, Ottawa, Ontario, Canada, K1N 6N5
{ural,zxu061}@site.uottawa.ca

Abstract. Extended Finite State Machine (EFSM)-based passive fault detection involves modeling the system under test (SUT) as an EFSM M, monitoring the input/output behaviors of the SUT, and determining whether these behaviors relate to faults within the SUT. We propose a new approach for EFSM-based passive fault detection which randomly selects a state in M and checks whether there is a trace in M starting from this state which is compatible with the observed behaviors. If a compatible trace is found, we determine that observed behaviors are not sufficient to declare the SUT to be faulty; otherwise, we check another unchecked state. If all the states have been checked and no compatible trace is found, we declare that the SUT is faulty. We use a Hybrid method in our approach which combines the use of both Interval Refinement and Simplex methods to improve the performance of passive fault detection.

1 Introduction

Passive fault detection is a fundamental part of passive testing which determines whether a system under test (SUT) is faulty by observing the input/output (I/O) behaviors of the SUT without interfering with its normal operations [10]. Compared with active fault detection, in which a tester has complete control over the inputs and devises a test sequence to reveal possible faults of the SUT, passive fault detection is more applicable under circumstances where the control is impractical or impossible, such as network fault management [10].

In Extended Finite State Machine (EFSM)-based passive fault detection, the specification of an SUT N is modeled as an EFSM M, N is treated as a blackbox, and the observed I/O behaviors of N is represented as a sequence E of observed I/O events. Determining whether N is faulty with respect to M is then based on the existence of traces in M that are compatible with E, i.e., a trace in M is compatible with E if E maps to a sequence of consecutive transitions of M starting at a state s of M. If the number of traces in M compatible with E is zero, then E is sufficient to determine that N is faulty. Otherwise, E is declared to be insufficient to determine whether N is faulty, i.e., there is at least one trace in M compatible with E and E needs to be augmented with additional I/O events of N to continue with passive fault detection.

A. Petrenko et al. (Eds.): TestCom/FATES 2007, LNCS 4581, pp. 335–350, 2007.

Usually, EFSM-based passive fault detection approaches are derived from Finite State Machine-based passive fault detection approaches. The FSM-based fault detection approach in [9] checks the observed sequence of I/O events one-by-one from the beginning, and reduces the size of the set S' of possible current states by eliminating impossible states until either S' is empty (N is faulty) or there is at least one state in S' (no fault is detected). The approach in [9] has been applied for passive fault detection in FSM-based systems [22, 23]. This approach has been extended to systems specified in the EFSM model by [7, 10, 11, 21] and adopted to systems specified in the Communicating Finite State Machine (CFSM) model by [14, 15, 16, 17, 18]. Another approach to EFSM-based passive fault detection focuses on characterizing specifications of an SUT in terms of invariants [3, 4, 5, 6].

This paper proposes a new approach for EFSM-based passive fault detection which is summarized as follows: assume that the subset S_0 of states of M contains all possible starting states of E. Randomly pick a state s in S_0 and determine whether there exists a trace in M that starts at s and is compatible with E. If such a trace is found, then stop and declare that E is not sufficient to determine whether N is faulty. In this case, the starting state and the current state of N can be determined readily using this trace. Otherwise, continue to check other states in S_0. After checking all the states in S_0, if no trace in M is found to be compatible with E, then N will be declared faulty.

The proposed approach provides information about possible starting state and possible trace compatible with E at the end of passive fault detection. Such information cannot be provided by the existing approaches derived from [9] unless a post-processing is performed or a backward checking approach is taken for exploring the information about possible starting state and possible trace [1, 2]. In addition, the proposed approach utilizes a Hybrid method to evaluate constraints in predicates associated with transitions in an EFSM which combines the use of both Interval Refinement [8, 19] and Simplex [13] methods for performance improvement during passive fault detection. We show that using only the Interval Refinement method has a similar performance to the Hybrid method but suffers from inaccuracy whereas using only the Simplex method has the same accuracy as the Hybrid method but suffers from poor performance.

The rest of the paper is organized as follows. Section 2 gives preliminaries needed for our discussion, including definitions and notations used in our presentation. Section 3 presents the proposed approach for EFSM-based passive fault detection in detail. Section 4 provides experimental evaluations. Section 5 concludes this paper with some final remarks and directions for future research.

2 Preliminaries

The proposed approach for EFSM-based passive fault detection is based on the specification of SUT N given as a Simplified Extended Finite State Machine (SEFSM) and the sequence of I/O behaviors a tester observes during the execution of N given as a sequence E of observed I/O events.

A *Simplified Extended Finite State Machine* (SEFSM) M is $(S, E_m, \bar{x}, T)$:

1. $S = \{s_1, \ldots, s_n\}$ is a finite set of states;
2. E_m is a finite set of I/O events. $e(\bar{y}) \in E_m$ is an input or output event, and $\bar{y} = (y_1, y_2, \ldots, y_p)$ is a vector of parameters of the I/O event e, called *local variables*;
3. $\bar{x} = (x_1, \ldots, x_r)$ is a vector of *global variables* which are accessible within all transitions;
4. T is a finite set of transitions.

The difference between $\bar{y}$ and $\bar{x}$ is that $\bar{y}$ is observable from SUT N while $\bar{x}$ is unobservable. Note that all variables are integers. An example SEFSM is shown in Figure 1.

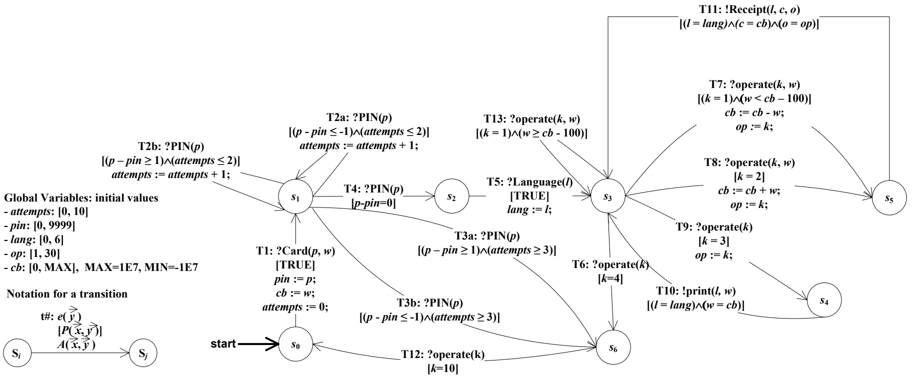


Fig. 1. The SEFSM *ATM* for an Automatic Teller Machine (ATM) system

A *transition* $t \in T$ in an SEFSM is $(s_i, s_j, e(\bar{y}), P(\bar{x}, \bar{y}), A(\bar{x}, \bar{y}))$:

1. s_i is the starting state of t;
2. s_j is the ending state of t;
3. $e(\bar{y}) \in E_m$ is an input event prefixed with "?" or output event prefixed with "!" that can be observed once t is activated;
4. $P(\bar{x}, \bar{y})$ is a predicate expressing the conditions to be satisfied for the activation of t which consists of conjunctive terms, each of which is defined as a *constraint*, connected by " $\wedge$ " (and) operators;
5. $A(\bar{x}, \bar{y})$ is an action consisting of a sequence of assignment statements, each updating a global or local variable as a function of elements of $\bar{x}$ and $\bar{y}$.

Examples of an I/O event, predicate, and action are: "!display(y)" is an I/O event "display" which outputs the value of y, "$(3 \times x_1 + (-1) \times x_2 \ \geq 0) \wedge (1 \times x_1 + 4 \times y_2 \ \leq 4)$" is a predicate, and "$x_3 := 3 \times x_1 +$ (-1) $\times\ x_2 +$ (-5); $x_1 := x_3$;" is an action, respectively.

Because $\bar{y}$ is observable from N while $\bar{x}$ is unobservable, the I/O events with global variables as parameters must be modified. For example, if x is a global variable, an input event "?read(x)" will be transformed to "?read(a) x:=a;" where a is a local variable and the action "x:=a;" assigns the value of a to x; similarly, an output event "!display(x)" will be transformed to "!display(a) [a = x]" where the predicate "[a = x]" guarantees the output value is equal to the value of x.

In this paper, a constraint cs is represented by $\sum_{i=1}^{k} a_i x_i = I$ (a_i is a coefficient, x_i is a global variable, I is an interval) after replacing the local variables of $\bar{y}$ by the actual values of the parameters observed during the execution of N. For example, the constraint "3 × x_1 + (-1) × x_2 ≥ 0" is represented by the expression "3 × x_1 + (-1) × x_2 = [0, MAX]". MAX is defined as 1 $\times 10^7$ and MIN is defined as -1 $\times 10^7$ in this paper.

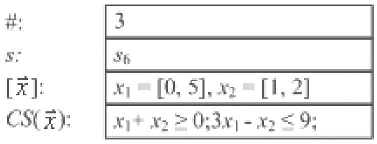
Note that an event-driven extended finite state machine (EEFSM) model is used in [10]. The differences between EEFSM and SEFSM models are as follows: the SEFSM model simplifies the structure of predicates in transitions by eliminating the "or" operator in EEFSM. Therefore, in SEFSM, a transition is executable if and only if all the constraints in the predicate are evaluated to be TRUE. Also, in actions associated with transitions in EEFSM, [10] only considered the assignment statements where the left hand side is a global variable, whereas we consider both global and local variables to be on the left hand side of assignment statements.

The sequence E of observed I/O events represents a sequence of I/O behaviors a tester observed during the execution of N, i.e., $e_1 e_2 \ldots e_n$. Like an I/O event in E_m, an observed I/O event e_i, $1 \leq i \leq n$, in E is also categorized as an observed input event prefixed with "?" or an observed output event prefixed with "!". Different from the I/O event in E_m, an observed I/O event in E contains determined values instead of symbols for variables. For example, "?read(3)" is an observed I/O event in E while "?read(y)" is an I/O event in E_m.

A *configuration* depicts a possible status of the SUT N during EFSM-based passive fault detection. A configuration c is a quadruple $(\#, s, [\bar{x}], CS(\bar{x}))$ where

1. # is the number of observed I/O events that have been checked to reach the configuration;
2. s is the possible current state of N;
3. $[\bar{x}]$ is a vector of *interval*s which represents the ranges of possible values which the variables in $\bar{x}$ can take;
4. $CS(\bar{x})$ records the constraints on variables in $\bar{x}$. These constraints are obtained from both predicates and actions. As $CS(\bar{x})$ contains only global variables, we shall henceforth use CS as the abbreviation of $CS(\bar{x})$.

For example, $c = (3, s_6, \{x_1 = [0,5], x_2 = [1,2]\}, \{x_1 + x_2 \geq 0; 3x_1 - x_2 \leq 9;\})$ is a configuration. (see Figure 2) According to configuration c in Figure 2, 3 observed I/O events have been checked; the current possible state of N is s_6; the value of x_1 is greater or equal to 0 and less than or equal to 5, and the value of x_2 is greater or equal to 1 and less than or equal to 2; the values of x_1 and x_2 must satisfy two constraints "$x_1 + x_2 \geq 0$" and "$3x_1 - x_2 \leq 9$" at the same time.



#:	3
s:	s_6
$[\vec{x}]$:	$x_1 = [0, 5], x_2 = [1, 2]$
$CS(\vec{x})$:	$x_1 + x_2 \geq 0; 3x_1 - x_2 \leq 9;$

Fig. 2. A configuration c

A *trace* represents the sequence of status of the SUT N during EFSM-based passive fault detection. *Trace-Tree* records all the traces that have been checked during EFSM-based passive fault detection.

1. A trace *trace* is a sequence of configurations, which are connected by transitions;
2. A Trace-Tree *Tree* for s consists of all the traces starting from a state $s \in S_0$. Each node in *Tree* represents a configuration and each edge stands for a transition between two configurations. Every trace $trace_i$ of length k, from s to a leaf in *Tree*, is compatible with a prefix of E $(e_1 e_2 \ldots e_k, k \leq |E|)$;
3. A trace in M compatible with E, henceforth called *compatible trace of E*, is defined as a trace in Trace-Tree for s with length equal to $|E|$.

3 The Proposed Approach

Given a specification SEFSM M of an SUT N, a sequence E of observed I/O events, and $S_0 \subseteq S$, the proposed approach proceeds as follows:

1. Pick an unchecked state s from S_0;
2. Build a Trace-Tree for s by finding all the possible traces starting from state $s \in S_0$;
3. If a compatible trace of E is found, declare this trace as a compatible trace of E; if no compatible trace of E can be found in Trace-Tree for s, go to (1);
4. If all states in S_0 have been checked and no compatible trace of E is found, declare that "N is faulty".

3.1 Algorithm Main

In algorithm $Main$, we randomly select a state s from $S_0 \subseteq S$ of SEFSM M and try to find a compatible trace of E starting from s. If *trace* is found to be a compatible trace of E, this algorithm will terminate and declare *trace* as a compatible trace of E; if all the states in S_0 have been checked and no compatible trace of E is found, the algorithm will report "N is faulty".

3.2 Algorithm Search_Trace_Tree

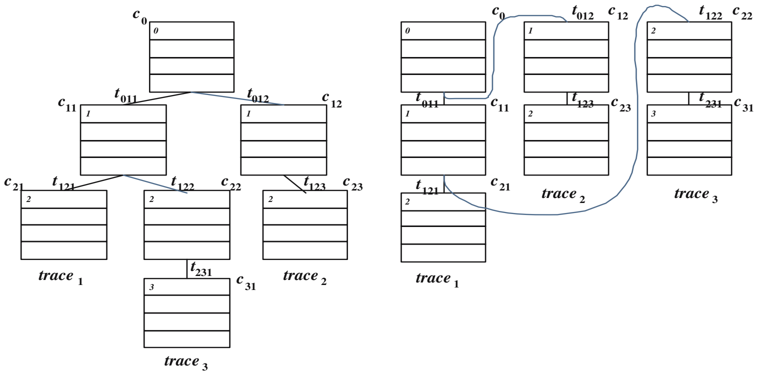
Algorithm *Search_Trace_Tree* searches for a compatible trace of E starting from a state s using the data structures for configuration and Trace-Tree.

Algorithm 1. Algorithm Main

```
Given: an SEFSM $M$,
  a sequence $E$ of observed I/O events, and
   $S_0 = \{ s_1, s_2, \cdots, s_n \}$
Return: "$N$ is faulty", or "trace is a compatible trace of $E$"
Begin:
  while ($S_0 \neq \emptyset$)
    randomly select a state $s$ from $S_0$;
    $S_0 \leftarrow S_0 \setminus \{ s \}$ ;
    trace ← Search_Trace_Tree(M, s, E);{search for a compatible trace of $E$}
    If(trace ≠ NULL)
      return ("trace is a compatible trace of $E$");
  endwhile
  return ("$N$ is faulty");   {no compatible trace of $E$ is found}
End
```

3.3 Algorithm Check_Trace and the Hybrid Method

A trace consists of a sequence of configurations which represents the sequence of changes in the status of N through E. Algorithm ***Check_Trace***$($$M$**,** ***trace*****,** E**,** ***Tree*****)** checks if there is a trace compatible with E. It first initializes the current configuration $c_{current}$ to the first configuration from *trace*, sets the current possible state s to the state in $c_{current}$ and gets the observed I/O event e to be considered from E. Then, all transitions in M starting from s (i.e., set T_s of transitions) are checked one by one. Those transitions passing both control portion and data portion fault detection will be considered as executable transitions corresponding to the observed I/O event e. As there may be more than one executable transition, algorithm *Check_Trace* picks the first one of them to



The architecture of a Trace-Tree *Tree*

Checking the Trace-Tree *Tree* by the proposed approach
Tree = (*trace* $_1$, *trace* $_2$, *trace* $_3$)

Fig. 3. The architecture of a Trace-Tree and its representation during passive fault detection

Algorithm 2. Algorithm *Search_Trace_Tree*(M, s, E)

```
Given: an SEFSM M,
  a state $s \in S_0$, and
  a sequence $E$ of observed I/O events
Return: a compatible trace trace, or NULL
Begin:
   Tree ← NULL;   {initialize the Trace-Tree Tree}
   trace ← NULL;   {initialize the trace trace}
   $[\bar{x}]_0$ ← set the initial intervals of the global variables in $M$;
   $c_0$ ← (0, $s$, $[\bar{x}]_0$, $\emptyset$ ); {create the initial configuration $c_0$ =(#, $s$, $[\bar{x}]$, $CS$)}
   trace.add($c_0$);   {add $c_0$ as the first configuration in this trace}
   Tree.add(trace);
   while (Tree ≠ $\emptyset$ )
      trace ← Tree.get(0);  {get the first trace in Tree}
      succ ← Check_Trace(M, trace, E, Tree);  {check if this trace is compatible with E}
      if (succ = TRUE)  {if trace is compatible with E}
         return (trace);
      else
         Tree.delete(trace);  {delete trace from Tree}
   endwhile
   return (NULL);  {no trace compatible with E has been found}
End
```

continue checking and adds all other transitions as branches into the Trace-Tree *Tree*. The procedure of checking a Trace-Tree is described in Figure 3. In Figure 3, *Tree* consists of three traces. For example, when checking configuration c_{11}, there exist two executable transitions, t_{121} and t_{122}. For each executable transition, a new configuration will be built. For c_{21}, which corresponds to the first executable transition t_{121}, we add c_{21} to the end of $trace_1$; for c_{22}, we build a new trace, $trace_3$, and set c_{22} as the starting configuration of $trace_3$. Then we continue checking $trace_1$ with c_{21}. $trace_3$ will be checked if and only if $trace_1$ and $trace_2$ are determined not compatible with E. Whenever a compatible trace of E is found, algorithm *Check_Trace* returns this trace.

When searching for executable transitions within algorithm *Check_Trace*, two steps are applied to a transition $t \in T_s$: In the first step, which corresponds to function *control_portion_checking* in algorithm *Check_Trace*, we compare the I/O event associated with transition t with the observed I/O event e (in E) by the prefix symbol, event name and possibly the number of parameters. If this comparison produces a mismatch, we stop processing transition t. Otherwise, we continue with the second step, which corresponds to the data portion fault detection, where we replace the local variables of $\bar{y}$ in predicate $t.P(\bar{x}, \bar{y})$ by the actual values of the parameters of the observed I/O event e and then transform the predicate into a list of constraints stored in *newCS*. After the replacement, the data portion fault detection problem is reduced to a *Constraint Satisfaction*

Problem (CSP) which is defined as follows: given (1) a configuration c, in which $c.[\bar{x}]$ contains a vector of intervals representing the ranges of possible values of global variables and $c.CS$ stores existing constraints on $\bar{x}$; and (2) a set *newCS* of new constraints, which is generated from $t.predicate(\bar{x}, \bar{y})$, determine if there exists at least one combination of values, called *solution*, in $c.[\bar{x}]$ that satisfies the existing constraints in $c.CS$ and new constraints in *newCS* simultaneously. If there exists a solution, the predicate $t.predicate(\bar{x}, \bar{y})$ will be considered consistent with the configuration c. If no solution exists, it means that an inconsistency has been detected.

To solve this CSP, the Interval Refinement method can be used, as done in [10]. However, because of the dependency problem, the results of the Interval Refinement method may not be accurate, i.e., some transitions may falsely be reported as executable. For example: assume a configuration c with $c.[\bar{x}] : x_1 = [1, 2], x_2 = [1, 2], x_1$ and x_2 are integers; $c.CS$: $\{cs : x_1 - x_2 = 0;\}$, and check two transitions t_1 with a constraint cs_1 in its predicate as: $x_1 + x_2 = 3$; t_2 with a constraint cs_2 in its predicate as: $x_1 + x_2 \leq 4$. By applying the Interval Refinement method, both transition t_1 and t_2 will be judged as executable. However, t_1 is not executable because x_1 and x_2 are integers and there is no solution for both cs and cs_1 at the same time. To guarantee the correctness of results, the Simplex method can be used instead of the Interval Refinement method, as done in [11]. Although the Simplex method is accurate, it is slower than the Interval Refinement method. Another difference between these two methods is that, in the Interval Refinement method, the intervals are narrowed; while in the Simplex method, the intervals will be untouched.

To combine the advantages of both the Interval Refinement and Simplex methods, we propose a Hybrid method, which is as accurate as the Simplex method and as efficient as the Interval Refinement method. The proposed Hybrid method uses both of these two methods judiciously as follows: given the set T_s of transitions, the current configuration $c_{current}$, and an observed I/O event e, first the Interval Refinement method, together with function *control_portion_checking*, is used to decide which transitions in T_s are executable. If no transition in T_s is evaluated to be executable, the current trace will be determined not compatible with E. If more than one transition is evaluated to be executable, the Simplex method will be applied to check the correctness of the Interval Refinement method in declaring these transitions executable. If only one transition is evaluated to be executable by the Interval Refinement method, the Simplex method will not be applied because this transition will be evaluated by the Simplex method implicitly by checking the last configuration of this trace. That is, at the end of a trace, before the trace is determined to be compatible with E, the Simplex method is applied to confirm that there exists no inconsistency in the last configuration of this trace. For example, consider a trace *trace* $(c_1 c_2 \ldots c_k, k \leq |E|)$ in the Trace-Tree *Tree*. If c_k is checked by the Simplex method and no inconsistency is found, *trace* is guaranteed to be compatible with a prefix of E $(e_1 e_2 \ldots e_k, k \leq |E|)$ because c_k contains all the constraints within the configurations from c_1 to

c_{k-1}. Therefore, if no inconsistency found in the last configuration of *trace* by the Simplex method, the transitions associated with *trace* are all executable.

After evaluating all the transitions in T_s, we continue to perform actions by function $\boldsymbol{action(t_c, e, c)}$ on the configurations in C with their corresponding transitions in T_s. After performing actions, we add the first configuration in C to the end of *trace* and continue to check *trace* starting from this configuration. Other configurations in C will be considered as the initial configuration of new branches, which are represented as new traces in the Trace-Tree.

In function ***Interval_Refinement*(*c.*[$\bar{x}$], *c.CS*, *newCS*)**, the interval arithmetic operations are applied to narrow the intervals of variables in constraints [19]. During refinement, if the interval of a variable is empty, an inconsistency is detected and function *Interval_Refinement* returns FALSE. Otherwise, *c.*[$\bar{x}$] is updated based on the new constraints *newCS* and *newCS* is added into the set *c.CS*.

In function ***Simplex*(*c.*[$\bar{x}$], *c.CS*, $\emptyset$)**, we adopt an open source tool *lp_solve* which is a free linear programming solver based on the revised Simplex method and the Branch-and-bound method [11, 12]. If no solution exists, function *Simplex* returns FALSE. Both *c.*[$\bar{x}$] and *c.CS* are unchanged within function *Simplex*.

The worst case computational complexities of Interval Refinement and Simplex methods are exponential. [10, 11, 20] show that the average complexities of both methods in practice are polynomial. However, because the Simplex method is more complex than the Interval Refinement method, the speed of the Simplex method is slower than that of the Interval Refinement method. However, the use of the Simplex method in conjunction with the Interval Refinement method does not adversely affect the efficiency of the Hybrid method because the frequency of applying the Simplex method in the Hybrid method is very low; and the Interval Refinement method narrows the intervals which helps reduce the cost of applying the Simplex method.

3.4 Function *action*

When a transition has been evaluated to be executable, a new configuration will be constructed to record the status of SUT N after this transition. The construction of a new configuration depends on the *action* part, $A(\bar{x},\bar{y})$, in the transition which consists of a sequence of assignment statements. Given a configuration c in the set of configurations built for all executable transitions, an observed I/O event e and a transition t_c corresponding to c, function $action(t_c, e, c)$ performs the actions associated with t_c, and builds a new configuration c_{next} which stands for the status of SUT N after t_c. The details of algorithm *action* are presented as follows: In the first step, we replace the local variables in the right hand expression (RHE) of an assignment statement by their values in e which gives an $RHE = \sum_{i=1}^{k} a_i x_i$. After the replacement, RHE without local variables is used to update the value of the left hand variable (LHV) in the configuration c. If LHV is a local variable, we use the value of RHE in the assignment statement to replace the existing value of this local variable. If LHV

Algorithm 3. Algorithm *Check_Trace*(*M*, *trace*, *E*, *Tree*)

```
Given: an SEFSM $M$,
  a trace trace,
  a sequence $E$ of observed I/O events, and
  a Trace-Tree Tree,
Return: FALSE, or   {trace is not a compatible trace of $E$}
          TRUE   {trace is a compatible trace of $E$}
Begin:
  $c_{current} \leftarrow$ trace.get(0);{get the first configuration}
  while ($c_{current} \neq$ NULL and $c_{current}.\# \neq E.\#$){ if there is an observed I/O event to be checked}
    $s \leftarrow c_{current}.s_c$;
    $T_s \leftarrow$ all transitions in $M$ starting at $s$;
    $e \leftarrow E.get(c_{current}.\# + 1)$;{ get the observed I/O event $e$}
    $C \leftarrow \emptyset$ ;
    for each transition $t$ in $T_s$   {evaluate transitions}
      $c \leftarrow c_{current}$;
      if (control_portion_checking($c$, $t$, $e$) = FALSE)
        end the for loop;  {the control portion is inconsistent}
      else   {the data portion fault detection commences}
        newCS $\leftarrow$ replace($t.P(\bar{x}, \bar{y})$, $e$);{eliminate local variables}
        if (Interval_Refinement($c.[\bar{x}]$, c.CS, newCS) = FALSE)
          end the for loop;  {the data portion is inconsistent}
        else
          $C \leftarrow C \cup \{ c \}$ ; {$c$ is modified and needs to be added to $C$}
    endfor
    if ($C = \emptyset$ ) return (FALSE); {if no executable transition is found}
    else
      if ( $|C| > 1$ or $c_{current}.\# + 1 = |E|$ ) {checking by the Simplex method}
        for each configuration $c$ in $C$
          if (Simplex($c.[\bar{x}]$, c.CS, $\emptyset$ ) = FALSE) $C \leftarrow C \setminus \{ c \}$ ;
        endfor
      else
        continue;
    if ($C = \emptyset$ ) return (FALSE); {if no configuration in $C$ is consistent}
    else
      for each configuration $c$ in $C$
        $c \leftarrow$ action($t_c$, $e$, $c$) ; {perform actions associated with $t_c$ which is the executable transition corresponding to $c$}
        if ($c$ = NULL)
          end the for loop;
        else
          if ($c$ is the first configuration in C)
            add $c$ to trace;
            $c_{current} \leftarrow c$;
          else
            build a new trace branch_trace;
            add $c$ to branch_trace; {create a new branch}
            add branch_trace to tree;
      endfor
  endwhile
  return (TRUE); {a trace compatible with $E$ is found}
End
```

Algorithm 4. Algorithm *action*$(t_c,\ e,\ c)$

```
Given: a transition $t_c$,
  an observed I/O events $e$, and
  the current configuration $c$
Return: new configuration $c_{next}$, or  {the configuration after transition $t_c$}
          NULL  {construction failed}
Begin:
  local_var ← set the values of the set of local variables according to $e$;
  $c_{next} \leftarrow c$;
  assignments ← $t_c.A(\bar{x}, \bar{y})$;  {put the assignments in $t_c.A(\bar{x}, \bar{y})$ into a vector}
  while(assignments is not an empty sequence)
    $a$ ← remove($a$, assignments);  {pick the first assignment}
    replace the local variables in $a$ using local_var;  {the first step}
    if ($a.LHV$ is a local variable) {the second step}
      $q$ ← find the index of variable $a.LHV$ in local_vars;
      local_vars[$q$] ← $a.RHE$; {replace by the value of $RHE$}
    else
      $q$ ← find the index of variable $a.LHV$ in $c.[\bar{x}]$;
      $[x_q] \leftarrow R(a.RHE)_{[\bar{x}]}$; {update the interval of $a.LHV$ in $[\bar{x}]$}
      if ($a.LHV$ appears in $a.RHE$)
        for every constraint $cs$ in $c_{next}.CS$ that contains $a.LHV$
          replace the $a.LHV$ in $cs$ by $(a.LHV - \sum_{i=1,i\neq q}^{k} a_i x_i)/a_q$;
        endfor
      else  {if $a.LHV$ does not appear in $a.RHE$}
        for every constraint $cs$ in $c_{next}.CS$ that contains $a.LHV$
          replace the variable $a.LHV$ in $cs$ with $[x_q]$;
          change $a$ to a new constraint $cs'$;
          $c_{next}.CS \leftarrow c_{next}.CS \cup cs'$; {add this new constraint}
        endfor
  endwhile
  return ($c_{next}$);
End
```

is a global variable, we first replace the interval of LHV in $c.[\bar{x}]$ by the value of interval $R(RHE)_{[\bar{x}]}$, then update the constraints containing LHV in $c.CS$. If LHV appears in RHE, for every constraint cs in $c.CS$ that contains LHV, we replace LHV in cs by $(a.LHV - \sum_{i=1,i\neq q}^{k} a_i x_i)/a_q$. If LHV does not appear in RHE, for every constraint cs in $c.CS$ that contains LHV, we replace the occurrences of LHV with $[x_q]$ and add the assignment to $c.CS$ as a new constraint. For example, the assignment "$x_1 := x_2 + x_3$ - 3" can be added as a constraint "$x_2 + x_3$ - x_1 = 3". Note that in [10], in the situation where LHV does not appear in RHE, all the constraints in $c.CS$ containing LHV will be discarded. However, those discarded constraints may contain constraints on not only LHV but also other global variables. Considering this, we keep those constraints and replace LHV in them by the interval of LHV in $[\bar{x}]$.

3.5 Optimization on Constraints

In algorithm *Check_Trace* and function *action*, evaluating and storing constraints are complex and time consuming. In order to reduce the complexity, we optimize the constraint related operations as follows: First, the values of global variables are represented by intervals. For a variable $x_i = [\underline{x_i}, \overline{x_i}]$, if its lower bound is equal to its higher bound (i.e. $\underline{x_i} = \overline{x_i}$), [10] considers the value of variable x_i as a *determined value*. Whenever the value of a global variable is determined, [10] replaces this variable in constraints with its determined value. For example, given the variable x_1 = [1, 1] and a constraint *cs*: $x_1 + x_2 - x_3$ = [-1, 5], x_1 in *cs* can be replaced by 1. Therefore, the new constraint after replacement would be *cs*: $x_2 - x_3$ = [-2, 4]. We adopt this replacement strategy in our approach.

Second, consider the situation in which a new constraint *cs* contains a single variable in the expression, for example $x_1 \leq 8$. It would be unnecessary to check *cs* with former constraints in *c.CS* and keep it in *c.CS*. Instead, we use *cs* to directly narrow the interval of this variable in $[\bar{x}]$. For example, given the existing interval of x_1 in $[\bar{x}]$ as x_1 = [0, 20], and a new constraint *cs* as $x_1 \leq 8$, the narrowed interval is x_1 = [0, 8]. If the narrowed interval is not empty, we use the narrowed interval to replace the existing interval in $[\bar{x}]$. Otherwise, we report that an inconsistency is found.

Third, when searching for a compatible trace of *E*, a transition *t* in *M* may be encountered more than once, i.e. the observed I/O event e_i and e_k $(i \neq k)$ in *E* may correspond to the same transition *t* in *M*. In this case, we may have two constraints cs_1: and cs_2:$cs_1 : \sum_{i=1}^{k} a_i x_i = I_1$ and $cs_2 : \sum_{i=1}^{k} b_i x_i = I_2$ such that $\forall i, 1 \leq i \leq k$, $a_i = z \times b_i$ where z is a constant. We will call cs_1 and cs_2 *similar*. For example, $x_1 + x_2$ = [1, 2] and $3x_1 + 3x_2$ = [0, 9] are similar. Then, given a new constraint *cs*, if there is a constraint within $c_{current}.CS$ that is similar to *cs*, we can reduce the number of constraints that need to be checked by the Hybrid method. In order to determine whether there is a constraint cs' in current *CS* that is similar to *cs*, we apply the following algorithm (called *Similarity_Checking*) before checking *cs* with function *Interval_Refinement.* If a constraint cs' similar to *cs* is found, we replace the interval of constraint $cs'.I$ by $(cs'.I \times z) \cap cs.I$. Thus, by applying algorithm *Similarity_Checking*, we can reduce the number of constraints that need to be checked by the Hybrid method.

4 Experiments

We made an experimental comparison of Interval Refinement, Simplex and Hybrid methods for EFSM-based passive fault detection on the ATM system of Figure 1. Within the SEFSM *ATM*, there are five global variables, i.e. $\bar{x}$ = (*attempts*, *pin*, *lang*, *op*, *cb*); seven states, i.e. $S_0 = \{s_0, s_1, \ldots, s_6\}$; and fifteen transitions. Local variables are defined within transitions. Each global variable is assigned an interval standing for its initial values. S_0 is determined by the

Algorithm 5. Algorithm *Similarity_Checking*

```
Given: a new constraint $cs$ ($\sum_{i=1}^{k} a_i x_i = I_1$), and
        a set of existing constraints $CS$
Return: FALSE, or    {inconsistency detected}
        TRUE    {no inconsistency detected}
Begin:
  for each constraint $cs'$ in $CS$        {$cs'$:$\sum_{i=1}^{k} b_i x_i = I_2$}
    if ($cs$ and $cs'$ are similar)
      $z \leftarrow a_i/b_i$;
      $cs.I \leftarrow (cs'.I \times z) \cap cs.I$;
      if ($cs.I = \emptyset$ ) return (FALSE);  {$cs$ is inconsistent with $cs'$}
      else
        $cs'.I \leftarrow cs.I$;return (TRUE);  {$cs$ is consistent with $cs'$}
  endfor
  return (TRUE);    {no inconsistency is found by $cs$}
End
```

tester according to the specific application at hand. In this experiment, S_0 is chosen to be equal to S.

In the experiment, we considered two cases. In Case I, called *correct implementation*, there is at least one trace in M that is compatible with E and this compatible trace is expected to be reported. In this case, we randomly generate a sequence E_s of observed I/O events ($|E_s| = 1000$) based on the SEFSM ATM and starting from state s_0. Within E_s, we randomly select five sequences with lengths of 20, 50, 100, 200, and 500 observed I/O events.

In Case II, called *faulty implementation*, there is no trace in M that is compatible with E and "faulty" is expected to be reported. First, we create a faulty specification ATM' from ATM by altering the next state, expanding a constraint in the predicate, or narrowing a constraint in the predicate of a randomly selected transition. Then, we randomly generate a sequence E_s of observed I/O events ($|E_s| = 1000$) based on the SEFSM ATM' and starting from state s_0. Within E_s, we randomly select ten sequences containing the altered transition with length of 30 observed I/O events.

We compared three implementations. The first implementation is the Hybrid method; the second implementation replaces the Hybrid method by the Interval Refinement method so that a transition is checked only by the Interval Refinement method (the same as in [10]); the third implementation replaces the Hybrid method by the Simplex method so that a transition is checked only by the Simplex method (the same as in [11]).

According to the results, in Case I, all three implementations successfully find the corresponding traces. In Case II with next state fault and expanded constraint fault, all three implementations report fault correctly. But, the fault with narrowed constraint cannot be detected by all the three implementations because an observed I/O event generated by narrowed constraint will certainly satisfy the original constraint.

Figure 4, left, compares the efficiency of these three implementations in terms of the average time cost. According to the results, the Interval Refinement method requires the least amount of time; the Hybrid method requires a little bit more time than the Interval Refinement method; and the Simplex method is the most expensive in terms of time. As the length of sequence E of observed I/O events increases, the time consumed for these three methods all increases.

Moreover, to compare the rate of increase of time costs, along with the increase of $|E|$, we compute the average rate of time costs between (1) Simplex method and Interval Refinement method (Simplex/IR); (2) Hybrid method and Interval Refinement method (Hybrid/IR). In Figure 4, right, we see that the time costs of the Interval Refinement method and Hybrid method are quite similar and, with the increase in the length of E, the difference between these two methods is not noticeable. We also see that the time cost of the Simplex method is much more than that of the Interval Refinement method, and as the length of E increases, the disparity between these two methods also increases.

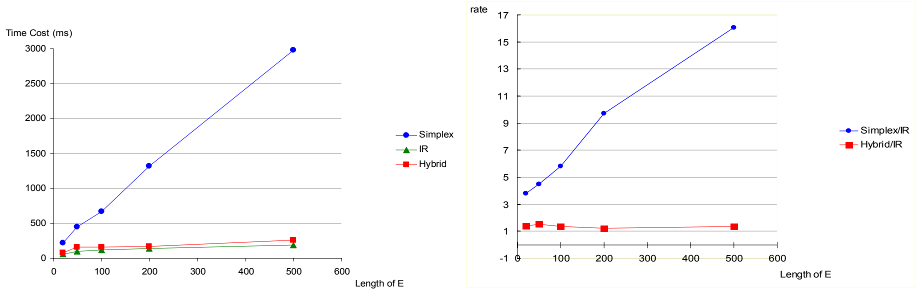


Fig. 4. The results of Case I by applying the Hybrid method, Interval Refinement method, and Simplex method (left) and rates of time cost of three methods (right)

5 Conclusions

In this paper, we have proposed an approach for EFSM-based passive fault detection which provides information about possible starting state and possible trace at the end of passive fault detection; and utilizes a Hybrid method which combines the use of both Interval Refinement and Simplex methods for performance improvement during passive fault detection. Through experiments, we show that, compared with using only the Interval Refinement or only the Simplex method, the Hybrid method guarantees the correctness of results with a reasonable time cost.

In future research, some model checking techniques can be adopted in the proposed approach for EFSM-based passive fault detection to help exploring the Trace-Tree. Also, it would be interesting to see how our proposed approach can help solving the problems of fault location and fault identification.

Acknowledgments

This work is supported in part by the Natural Science and Engineering Research Council of Canada under grant RGPIN 976 and CITO/OCE of the Government of Ontario. The authors wish to thank Dr. Fan Zhang for many useful discussions.

References

1. Alcalde, B., Cavalli, A., Chen, D., Khuu, D., Lee, D.: Network protocol system passive testing for faulty management - a backward checking approach. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 150–166. Springer, Heidelberg (2004)
2. Alcalde, B., Cavalli, A.: Parallel passive testing of system protocols c towards a real-time exhaustive approach. In: International Conference on Network, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICN/ICONS/MCL 06), pp. 42–42 (2006)
3. Arnedo, J.A., Cavalli, A., Nunez, M.: Fast testing of critical properties through passive testing. In: Hogrefe, D., Wiles, A. (eds.) TestCom 2003. LNCS, vol. 2644, pp. 295–310. Springer, Heidelberg (2003)
4. Bayse, E., Cavalli, A., Nunez, M., Zaidi, F.: A passive testing approach based on invariants: Application to the wap. Computer Networks and ISDN Systems 48, 247–266 (2005)
5. Cavalli, A., Gervy, C., Prokopenko, S.: New approaches for passive testing using an extended finite state machine specification. Information and Software Technology 45, 837–852 (2003)
6. Cavalli, A., Vieira, D.: An enhanced passive testing approach for network protocols. In: International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICN/ICONS/MCL06), pp. 169–169 (2006)
7. Chen, D., Wu, J., Chu, T.: An enhanced passive testing tool for network protocols. In: International Conference on Computer Networks and Mobile Computing (ICCNMC 03), pp. 513–516 (2003)
8. Hansen, E., Walster, G.: Global Optimization Using Interval Analysis, 2nd edn. New York: Marcel Dekker Inc (2004)
9. Lee, D., Netravali, A.N., Sabnani, K.K., Sugla, B., John, A.: Passive testing and applications to network management. In: IEEE International Conference on Network Protocols (ICNP97), pp. 113–122 (1997)
10. Lee, D., Chen, D., Hao, R., Miller, R.E., Wu, J., Yin, X.: A formal approach for passive testing of protocol data portions. In: IEEE International Conference on Network Protocols (ICNP02), pp. 122–131 (2002)
11. Lee, D., Chen, D., Hao, R., Miller, R.E., Wu, J., Yin, X.: Network protocol system monitoring c a formal approach with passive testing. IEEE/ACM Transactions on Networking 14, 424–437 (2006)
12. LP_Solve: Tool lp_solve, version 5.5.0.9 (2007) `http://lpsolve.sourceforge.net/5.5/`
13. Marriott, K., Stuckey, P.: Programming with Constraints: An Introduction. MIT Press, Cambridge, Mass (1998)
14. Miller, R.: Passive testing of networks using a cfsm specification. In: IEEE International Performance, Computing and Communications Conference (IPCCC98), pp. 111–116 (1998)

15. Miller, R., Arisha, K.: On fault location in networks by passive testing. In: IEEE International Performance, Computing and Communications Conference (IPCCC00), pp. 281–287 (2000)
16. Miller, R., Arisha, K.: Fault identification in networks by passive testing. In: 34th Annual Simulation Symposium, pp. 277–284 (2001)
17. Miller, R., Arisha, K.: Fault identification in networks using a cfsm model by passive testing. Technical report, UMIACS (2001)
18. Miller, R., Arisha, K.: Fault coverage in networks by passive testing. In: International Conference on Internet Computing, pp. 413–419 (2001)
19. Moore, R.: Interval Analysis. Prentice-Hall Inc, Englewood Cliffs, N.J (1966)
20. Spielman, D., Teng, S.: Smoothed analysis: Why the simplex algorithm usually takes polynomial time. Journal of the ACM 51, 385–463 (2004)
21. Tabourier, M., Cavalli, A.: Passive testing and application to the gsm-map protocol. Information and Software Technology 41, 813–821 (1999)
22. Wu, J., Zhao, Y., Yin, X.: From active to passive: Progress in testing of internet routing protocols. In: IFIP FORTE01, pp. 101–118 (2001)
23. Zhao, Y., Yin, X., Wu, J.: Online test system, an application of passive testing in routing protocols test. In: 9th IEEE International Conference on Networks, pp. 190–195 (2001)