A Formal Approach for Passive Testing of Protocol Data Portions

David Lee[†],Dongluo Chen[‡], Ruibing Hao[†], Raymond E. Miller[⋆], Jianping Wu[‡] and Xia Yin[‡]
[†]Bell Labs Research China, Lucent Technologies, Beijing, China
[‡]Department of Computer Science, Tsinghua University, Beijing, China
[⋆]Department of Computer Science, University of Maryland, Maryland, USA
[†]{lee,rhao}@research.bell-labs.com
[‡]{chdl,jianping,yxia}@csnet1.cs.tsinghua.edu.cn
[⋆]miller@cs.umd.edu

Abstract

Passive testing is a process of detecting faults in a system under test by passively observing its input/output behaviors only without interrupting its normal operations, and proves to be a promising technique for network fault management. We study passive testing of data portions of network protocols and present two algorithms, using an Event-driven Extended Finite State Machine model. Experimental results on the Internet routing protocol OSPF are reported.

1. Introduction

Passive testing is a process of detecting faults in a system under test by passively observing its input/output behaviors only without interrupting its normal operations. It collects input/output information from the "in-process" system without disturbing the normal network operations by applying active testing messages.

Passive testing is useful for network fault management. As a tester collects input/output messages from an implementation under test it checks if the message sequence conforms to the specification. In [4], the authors reviewed the concept of passive testing and developed algorithms for finite state machine (FSM) and nondeterministic finite state machine (NFSM) models. In [6] [7], the authors specified the network as a communicating finite state machine (CFSM) where fault detection and location were studied. Passive testing is a feasible approach for distributed systems such as routing protocols [11] while active testing is not always feasible, since it is difficult to interrupt a router in operation.

The published works on passive testing are on control portion testing, yet little effort has been devoted to the data portion testing. Specifically, a protocol data portion contains variables to encode states, which can be modeled as an Extended Finite State Machine(EFSM). In [10] a simple algorithm on EFSM was developed and applied to the GSM-MAP protocol, and the test coverage of passive testing was demonstrated. The algorithm records the values of variables, and discards them whenever ambiguity occurs. Yet no convincing arguments were given concerning how the faults were detected.

In this paper, we passively test the variable values of protocols and propose an efficient algorithm to trace the variable values as well as the system state. Variables are critical in some protocols since they can determine the external behavior of the system. On the other hand, it is known to be difficult to test variable values [5]. Our algorithms can deal with all kinds of operations on variable values associated with transitions and provide efficient variable value determination for the data portion fault detection.

The paper is organized as follows. In Section 2, an Event-driven EFSM model is presented where the predicates and actions are defined in BNF form. A simple algorithm is introduced in Section 3.1, and then modifications are discussed with examples. A more powerful algorithm is proposed in Section 3.2 and illustrated in Section 3.3. Fault detection capabilities of passive testing algorithms are also discussed. In Section 4, a model of the OSPF neighbor state machine is described and the experimental results with different algorithms are reported. We conclude our paper in Section 5.

2. A Model

Network protocols contain variables. Although an FSM can usually represent the control portion of a communication protocol, it is not powerful enough to properly model the data portion, the variables associated with a protocol system. We introduce an Event-driven Extended Finite

Proceedings of the 10 th IEEE International Conference on Network Protocols (ICNP'02)
1092-1648/02 $17.00 © 2002 IEEE

State Machine model to express the data portion so as to effectively determine how the variables affect state transitions and how they are modified during transitions. In passive testing, only observable behavior of the network devices can be captured. Each packet/message from one peer to its counterpart is called an event. An event is either an input message to the device under test or an output message from it, but not both. The fields in the message are taken as parameters of this event. The predicate of a transition is a logic (Boolean) expression. It is composed of a set of variables, denoted by a vector $\vec{x} = (x_1, \ldots, x_k)$, and the parameters of the current event.

In active testing, the tester sends a message to the implementation under test (IUT), and then waits for a response. The messages exchanged between the tester and IUT can be viewed as a sequence of stimulus/response pairs: $i_1/o_1, i_2/o_2, i_3/o_3, \ldots$. But in passive testing, all the packets are captured individually by the tester from the network without any knowledge of the causal relations between the packets. Therefore, we use an Event-driven EFSM to model the protocol system under test. In this model, we handle one event at each time, either an input event or an output event. If the original protocol specification has a transition like $s_1 \xrightarrow{i/o} s_2$, we will convert it into $s_1 \xrightarrow{i} s_1' \xrightarrow{o} s_2$. This model is quite flexible. It is suitable for the passive testing of routing protocols such as BGP and OSPF, and it also works well for other protocols such as TCP.

Definition 1 (Event-driven Extended Finite State Machine) *An Event-driven Extended Finite State Machine(EEFSM) is a quintuple $M =< S, s_0, \Sigma, \vec{x}, T >$, where*

1. *$S = \{s_0, s_1, ..., s_{n-1}\}$ is a finite set of states.*
2. *$s_0 \in S$ is the initial state.*
3. *Σ is a finite set of events. For $e(\vec{y}) \in \Sigma$, e is the event name, and $\vec{y} = (y_1, y_2, ..., y_r)$ is a finite set of parameters of the event.*
4. *$\vec{x}$ is a vector denoting a finite set of variables.*
5. *T is a finite set of transitions. For $t \in T$, $t =< s, s', e(\vec{y}), P(\vec{x}, \vec{y}), A(\vec{x}, \vec{y}) >$ is a transition where s and s' are the start and end state of the transition, $P(\vec{x}, \vec{y})$ is a predicate of the variables and the input event parameters, and $A(\vec{x}, \vec{y})$ is an action which is a set of assignment statements, updating $\vec{x}$ as a function of $\vec{x}$ and $\vec{y}$.*

In communication protocols, most variables are integer-valued variables. The predicates are logic expressions on these integer variables. A transition is executed if and only if its predicate has a value TRUE with the current variables and input parameters values. The actions in transitions are composed of assignments to the variables. The definitions of predicate and action in BNF form are in Appendix A .

3. Passive Testing Algorithms

In [4] the passive testing process contains two phases. The first phase is to identify the current state, which is called passive homing. The second is the fault detection phase where all the events are traced to find a difference between the specification and the implementation. When we turn to the EEFSM model, the homing phase has to find the values for the variables as well as the current state. We use the term "state homing" when the current state is identified. Whereas "variable homing" means all the variable values are determined by the tester. According to the EEFSM model, variable values are set in Actions. In passive testing, the network is in operation when the passive testing process starts. There is no guarantee that "state homing" or "variable homing" can be achieved. In this work, we combine the two phases, since the homing phase and the fault-detection phase use the same procedure. We introduce two passive testing algorithms with the EEFSM model. Algorithm 1 is simple and similar to the one in [10]. After discussing its deficiencies, we present Algorithm 2, which is much more efficient in detecting faults. And we also give a criteria to evaluate the fault detection capability of different passive testing algorithms.

3.1. Algorithm 1

A straightforward approach is to record and then keep track of a variable value when it is known. We use *unknown* when its value has not been determined by the tester yet. The current status of a machine is denoted as:

- The current possible state set: S_c
- The current possible variable value vector : $\vec{x}$.

Initially $S_c = \{s_0, s_1, ..., s_{n-1}\}$ and all the variable values in $\vec{x} = (x_1, ..., x_k)$ are *unknown*.

In this algorithm, the predicate of a transition must be evaluated to guard the transition. Since there may be variables with *unknown* value in the predicate, there may be three possible results of the evaluation.

1. TRUE. The logic expression is evaluated to be true.
2. FALSE. The logic expression is evaluated to be false.
3. POSSIBLE. If no definite result is given.

When a predicate is evaluated to be FALSE, this transition definitely cannot be executed. Otherwise the predicate is not false (either TRUE or POSSIBLE) and the transition could be executed. When an event occurs, there can be one or more transitions, which could possibly be executable. When we say a transition is possible, we mean the start state of the transition is in S_c and the predicate of the transition is evaluated to be not false. Each possible transition will create a possible next state and a possible new vector value $\vec{x'}$ from the corresponding actions. If there is more

than one possible transition, the final values of the vector $\vec{x'}$ is determined according to the following two rules:

R1 - If the candidate transitions give the same value to a variable, then this variable is assigned this value.

R2 - If the candidate transitions assign different values to a variable, then the variable is labeled as unknown after this event.

If a transition is possible, the end state of this transition is added to the next possible state set. Along with the variable assignments, we obtain the next possible system status after the current event.

Algorithm 1
Input : An EEFSM of the IUT
Output : faults detected
$Q1$: the current possible state set
$Q2$: the next possible state set

begin
1. $Q1 := \{s_0, s_1, ..., s_{n-1}\}$;
2. $\forall x_i \in \vec{x}$, $x_i := unknown$;
3. **while** (GetNextEvent(e))
4. $Q2 := \phi$;
5. **for** each transition t that (t.event=e) $\wedge$ (t.start_state$\in Q1$)
6. construct a new variable value set $\vec{x_t} := \vec{x}$;
7. **if** Eval(t.predicate) = *false* under $\vec{x_t}$
8. **then** delete $\vec{x_t}$;
9. **else** execute the actions;
10. add t.end_state to $Q2$;
11. $\vec{x} := \cup_t \vec{x_t}$ according to *R1* and *R2*;
12. **if** $Q2$ is empty
13. **then return** "fault detected";
14. **else** $Q1 := Q2$
end

We analyze the time complexity of Algorithm 1. In our definition of predicate and action in Appendix A, the predicate of a transition consists of one or more simple predicates and the action part of a transition is a set of assignments. It takes a unit time to evaluate a simple predicate or to do a simple assignment. Suppose that there are a total of m transitions in the EEFSM model. Assume that $|a|_{max}$ is the maximal number of assignments in the action part. Similarly denote the maximal number of simple predicates in the predicate by $|p|_{max}$.

Proposition 1 *If $Q2$ becomes empty at the lth event, it takes a maximal time $O(ml(|p|_{max} + |a|_{max}))$ for Algorithm 1 to detect a fault of an implementation machine.*

Algorithm 1 is simple. Yet it has some deficiencies. In what follows we analyze these deficiencies with some examples.

1. Algorithm 1 does not use the implicit information in the predicates. The predicates are only used to guard the executability of the transitions. In fact, useful information can be obtained from predicates. Consider the example in Fig. 1. Using Algorithm 1, the value of u remains *unknown* after the transition fires. However, this is the only possible transition from S1. We know that $u = 1$ after this transition. We call the constraints in the predicates the "implicit" information, which could be very useful for passive homing as well as fault detection.

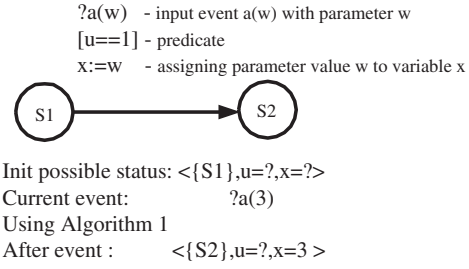
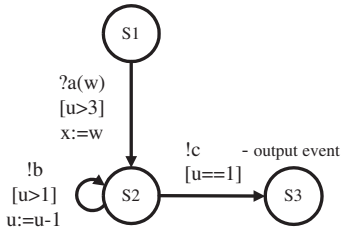


Figure 1. Predicates in transitions

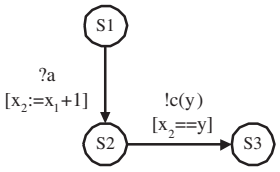
2. Because Algorithm 1 does not keep a record of the constraints on the variables values, it may allow certain transitions to happen that are actually not possible. The constraints on variables values could contribute significantly to fault detection. See the example in Fig.2. With the initial state S1 and u unknown, after input event $?a(3)$ occurs, Algorithm 1 takes that the transition from S2 to S3 is also possible. Actually after $?a(3)$ occurs, $u > 3$ must be satisfied. If we keep this information, then the transition with event $!c$ from S2 to S3 cannot be executed because the predicate evaluates to be FALSE. So there should not be an input event $?a(3)$ followed immediately by an output event $!c$. If an implementation contains such a fault, it will go undetected by Algorithm 1.

3. Algorithm 1 determines variable values in a simple and direct way without considering any information about the relations between variables. See the example in Fig.3. When the transition from S1 to S2 fires, x_2 gets its value from x_1+1. Then x_2 is output in the transition $!c(y = 5)$ through y. Even if we have used the implicit information, the variables values we can obtain after these transitions are $< x_1 =?, x_2 = 5 >$. But if we know the relation between x_1 and x_2, we can tell that $x_1 = 4$. Note that there are many assignments in protocol specification between variables. Discarding the information of the relations between variables reduces the capability to discover variables values and to detect faults.



init possible status: <{S1},u=?, x=?>
Using Algorithm1
after event a(3) : <{S2},u=?, x=3>
then if the !c event occurs just after ?a(3),
the algorithm can not detect this fault.

Figure 2. Inequalities in protocol specification



init configuration set: <{S1}, x_1=?, x_2=?>
Using Algorithm 1
after event ?a and !c(5): <{S3}, x_1=?, x_2=5>
If we know the relation between x_2 and x_1,
we know that x_1 must be 4.

Figure 3. Relations between variables

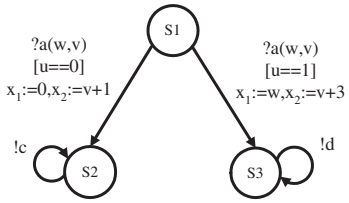
4. In Algorithm 1, only one possible variable value set is kept. If several candidate transitions give one variable different values, the value of this variable stays unknown. This method is very simple, but it could be ineffective. See the example in Fig.4. With Algorithm 1, no variable value can be determined. But if we associate a variable value set with each possible state, more information can be extracted. In this example, when event $?a(4,7)$ occurs, we trace the two transitions. The current possible configurations are $< S2, x_1 = 0, x_2 = 8 >$ and $< S3, x_1 = 4, x_2 = 10 >$. Afterwards if event $!d$ happens, we can delete the first one and know the current configuration as $< S3, x_1 = 4, x_2 = 10 >$. Here the events $!c$ and $!d$ are characterizing sequences that can distinguish S2 from S3. The length of characteristic sequences is greater than 1 in general [5] .

3.2. Algorithm 2

Considering all these and other deficiencies of Algorithm 1, we present Algorithm 2, which is much more effective for fault detection and yet more complex.

1. Use the Interval formalism [8] to denote integer variable values. The value of v is denoted as $R(v)$. $R(v) = [a, b]|a \leq v \leq b$.

   As a special case, a precise value is represented as $R(v) = [a, a]|v = a$, and we say that v is *decided*. A set of operations on integer variable values denoted by an interval is given in Appendix B.



init status set : <{S1},u=?, x_1=?, x_2=?>
The events are ?a(4,7) and !d
Using Algorithm 1
the final status: <{S3},u=?, x_1=?, x_2=?>

Figure 4. Characterizing sequences

2. Use an assertion to record the constraints on integer variables.

   Definition 2 (Assertion) *An assertion, $assert(\vec{x})$, is a predicate that is observed to be true at the current state.*

   The assertion, $assert(\vec{x})$, records the constraints on variables. Constraints can be obtained from either predicates or actions. When a transition is fired, the predicate of this transition is assumed to be true and is added to $assert(\vec{x})$. Assignments with undecided variables on the right side should also be included into $assert(\vec{x})$ as well as predicates. For example, in Fig.3, when the event $?a$ occurs, the clause $(x_2 := x_1 + 1)$ redefined x_2. So all the clauses in $assert(\vec{x})$ with x_2 should be deleted. And then the predicate $(x_2 = x_1 + 1)$ should be put into $assert(\vec{x})$. When the output event $!c(5)$ occurs, it reveals that $x_2 = 5$. Every x_2 in $assert(\vec{x})$ is replaced by 5 thus $x_1 = 4$ is obtained.

3. Use Candidate Configuration Set to represent the possible statuses of the protocol.

   Definition 3 (Candidate Configuration Set) *A Candidate Configuration Set (CCS) c is a triple $< s, R(\vec{x}), assert(\vec{x})$, where*

   - *s is the state*
   - *$R(\vec{x})$ is the variable set whose values are represented by Interval*
   - *$assert(\vec{x})$ is an assertion on $\vec{x}$.*

We use a CCS to represent the possible statuses of a machine. A CCS not only indicates the current state of the machine, but also records the constraints on the variable vector $\vec{x}$ in $R(\vec{x})$ and $assert(\vec{x})$.

For convenience, $assert(\vec{x})$ is represented in disjunctive normal form (DNF), which consists of conjunctive terms, connected by "$\vee$"(or) operators. Each conjunctive term can be a simple predicate or simple predicates connected by "$\wedge$"(and) operators.

In Algorithm 2 we use two CCS queues Q1, Q2, where Q1 records the current possible CCS set and Q2 records the next possible CCS set. Upon an event e, we find out all the eligible CCS in Q1 to fire transitions. The next transition to fire from each CCS must be consistent with those constraints in the current CCS. That is, before we could declare that a transition is executable, the predicate of this transition has to be checked to assure that it is consistent with the $R(\vec{x})$ and $assert(\vec{x})$ of the current CCS. If a transition passes the consistency check, a successor CCS is generated.

There may be multiple CCSs at a moment before homing. Initially, there are n possible CCSs, where n is the number of states in the machine, and all the variable values are *unknown*. When we start to trace the events, some CCSs, which could not fire transitions, are removed. However, there may be CCSs with the same state but different $R(\vec{x})$ and $assert(\vec{x})$, and the number of CCSs may increase. If the number of CCSs exceeds a threshold N, there should be a procedure to combine the similar ones. How to determine N? Experiments showed that the number of situations is not very likely to be greater than $3n$, and as a practical solution we set $N = 3n$.

Algorithm 2
Input : An EEFSM of the IUT
Output : faults detected
$Q1$: the current possible CCS set
$Q2$: the next possible CCS set
$|Q2|$: the number of elements in $Q2$

begin
1. $Q1 := \{ccs_0, ccs_1, ..., ccs_{n-1}\}$,
   $ccs_i :=< s_i, \vec{x} := unknown, assert(\vec{x}) := \phi >$;
2. **while** (GetNextEvent(e))
3. $Q2 := \phi$;
4. **for** each CCS $c \in Q1$
5. **for** each $t \in T$, (t.event=e)$\wedge$(t.start_state=c.state)
6. substitute the variables and parameters in t.predicate with values if they are decided;
7. **if** t.predicate evaluates false, **goto** 5;
8. construct a new CCS $c_t = c$, add t.predicate to $c_t.assert(\vec{x})$;
9. **check_consistency**(c_t)$^{[Proc1]}$;
10. **if** there are inconsistencies in c_t **then** delete c_t, **goto** 5;
11. **do_actions**(c_t, t.**action**)$^{[Proc2]}$;
12. c_t.state:=t.end_state, $Q2 := Q2 \cup \{c_t\}$.
13. **if** $Q2$ is empty **then return** "fault detected";
14. **if** $|Q2| > N$ **then** combine similar CCSs ;
15. $Q1 := Q2$
end

In Algorithm 2, from step 8 to step 10, we check the satisfiability of t.predicate by contradiction. According to the definition of $assert(\vec{x})$, $c_t.assert(\vec{x})$ is always true for $c_t.R(\vec{x})$. In step 8, we assume t.predicate is true. Then the conjunction of t.predicate and $c_t.assert(\vec{x})$ should also be true. We check the satisfiability of this conjunction in the **check_consistency** procedure. If the conjunction is found unsatisfiable, it means that our assumption about t.predicate is incorrect and transition t cannot be fired. If the procedure finds the conjunction is unsatisfiable, we declare that t.predicate and $c_t.assert(\vec{x})$ are inconsistent.

3.2.1 Consistency Check and Interval Refinement

The purpose of the **check_consistency** procedure is to refine the interval of each variable in addition to the satisfiability checking. If the interval of some variable becomes empty, the execution sequence to this point must contain a transition whose predicate was FALSE(yet with the value POSSIBLE during the execution), and we know that there exists inconsistency among the constraints, and the current configuration is not possible.

Before we delve into the details of consistency check, we first introduce two concepts: normalized simple predicate and Interval refinement.

To find inconsistency between t.predicate and $c_t.assert(\vec{x})$, we first need to transform their conjunction to DNF form. Each conjunctive term in this DNF is a simple predicate or simple predicates connected by "$\wedge$" operator. Each simple predicate can be transformed into a normalized form as follows.

Definition 4 (Normalized Simple Predicate) *A normalized simple predicate is in the form of* $a_1x_1 + a_2x_2 + ... + a_kx_k \sim Z$ *, where:*

- $\sim \in \{<, >, \leq, \geq, =, \neq\}$
- $x_1, x_2, ..., x_k$ *are variables*
- $a_1, a_2, ..., a_k, Z$ *are integer constants.*

A normalized simple predicate can be rewritten in the form of an interval equation, such as $\sum_i a_i R(x_i) = R(\sim Z)$, where $R(\sim Z)$ denotes the interval of $\sim Z$.

To normalize a simple predicate, we first substitute all the variables whose values have been decided. Then move the constant items to the right and the variables to the left side of the operator. For example, $7x_2 - x_3 < 9 + x_1$

with $x_3 = 2$ is normalized to $(-1)x_1 + 7x_2 < 11$. This normalized simple predicate can also be rewritten as $(-1)R(x_1) + 7R(x_2) = (-\infty, 10]$.

Definition 5 (Interval Refinement) *For an interval $R(x_i)$ in an interval equation $\sum_i a_i R(x_i) = R(\sim Z)$, $R(x_i)'$, the refinement of $R(x_i)$, can be defined as:* $R(x_i)' = ((R(\sim Z) - \sum_{j \neq i} a_j R(x_j))/a_i) \cap R(x_i)$.

Proc1 shows the procedure of checking the consistency of a candidate configuration set CCS. It also performs interval refinement at the same time.

Proc1 check_consistency(c)
Input : a CCS c
Output : FALSE if there are inconsistencies in c, else TRUE

begin
1. Transform $c.assert(\vec{x})$ to DNF;
2. *flag := FALSE*;
3. **for** each conjunctive term D_l in $c.assert(\vec{x})$
4. construct a new variable set $R_l(\vec{x}) := c.R(\vec{x})$;
5. **for** each simple predicate p in D_l
6. normalize p;
7. **if** $\sum_i a_i R_l(x_i) \cap R(\sim Z) = \phi$
then delete current $R_l(\vec{x})$ and D_l, **goto** 3;
8. **if** $\sum_i a_i R_l(x_i) \subseteq R(\sim Z)$
then delete p from D_l, **goto** 5;
9. **for** each x_i
$R'_l(x_i) := ((R(\sim Z) - \sum_{j \neq i} a_j R_l(x_j))/a_i) \cap R_l(x_i)$
if $R'_l(x_i) = \phi$
then delete current $R_l(\vec{x})$ and D_l, **goto** 3;
10. **if** $R'_l(\vec{x}) \subset R_l(\vec{x})$
then $R_l(\vec{x}) := R'_l(\vec{x})$, **goto** 5;
else *flag := TRUE*;
11. **if** (*flag*)
then combine all the $R_l(\vec{x})$ generated in step 3
to $R(\vec{x})$ according to *R3*, $c.R(\vec{x}) := R(\vec{x})$;
12. **return** *flag*
end

In Proc1, conjunctive terms are processed separately (line 3) and the results are combined together (line 11). For each conjunctive term, every simple predicate is checked and the intervals of variables are refined iteratively. Whenever an interval changes its value, Proc1 goes back to line 5. The stopping criterion is: no variable changes its interval. Since each iteration reduces the interval strictly monotonically and the interval boundaries are integers, the iteration stops in finitely many steps.

A flag is used in Proc1 to indicate that at least one of the conjunctive terms is satisfied. If the value of this flag is false after the loop, it means the entire conjunctive terms are unsatisfiable and there is inconsistency in $c.assert(\vec{x})$. Otherwise we report no inconsistency found and update $R(\vec{x})$ by combining all the generated $R_l(\vec{x})$ according to the following rule:

R3 - if there are several candidate intervals for a variable after the refinement, then the lower bound of this variable takes the minimal of all the candidate lower bounds, and the upper bound takes the maximal of all the candidate upper bounds.

The check_consistency procedure gives a general method to refine the intervals of variables. Heuristic procedures can be used to simplify the calculation. For example, at the beginning of processing a conjunctive term, we can pick those equations in the form of "$x == constant$" and use the constant value to replace x in the conjunctive term. Such methods can help to reduce the number of iterations.

The computation time of this procedure depends on the number of the conjunctive terms and the interval of variables. In the worst case, the number of iterations equals the length of the variable's interval, $\sum_j |R(x_j)|$ where $|R(x_j)|$ denotes the length of x_j's interval. In most protocol specifications, the predicates are quite simple. We have analyzed some protocol specifications, such as OSPF, TCP, PPP and BGP, the **check_consistency** procedure can usually be completed in no more than 3 iterations.

3.2.2 Executing Actions

If a transition is executed, the variables may be modified by the assignments in the action. In passive testing, to trace the variable values we follow the assignments in the specification. Proc2 contains a procedure for executing the assignments.

Assignments can be classified into three types. We have to pay attention to assignments in the form of $w := f(w, u, v, ...)$. Inverse function $f^{-1}(w)$ should take the place of w in $assert(\vec{x})$.

Proc2 do_action(c_t, t.action)
Input : c_t - current CCS ,
t.action - the action part of current transition
Output : updated c_t

begin
1. **for** each clause in t.action
2. substitute the known variables on the right side with their values;
3. **case** this clause is of the form $w := constant$
4. $c_t.R(w) := constant$;
5. delete all the clauses in $c_t.assert(\vec{x})$ containing w;
6. **break**;
7. **case** this clause is of the form $w := f(w, u, v, ...)$
8. calculate $c_t.R(w)$;
9. replace all the w in $c_t.assert(\vec{x})$ with $f^{-1}(w)$;

10. **break**;
11. **case** this clause is of the form $w := f(u, v, ...)$
/*the parameters of f do not contain w*/
12. calculate $c_t.R(w)$;
13. delete all the clauses in $c_t.assert(\vec{x})$ containing w;
14. add clause $w = f(u, v, ...)$ to $c_t.assert(\vec{x})$;
15. **break**;
16. **return** c_t

end

Now we analyze the complexity of Algorithm 2. Assume that m is the total number of transitions in the EEFSM, $|a|_{max}$ is the maximum number of assignments in an action; $|p|_{max}$ and $|assert|_{max}$ are the maximum number of simple predicates in a predicate and in an assertion respectively. Suppose r is the worst case number of iterations for assertions, which equals to $\sum_j |R(x_j)|$. k is the number of variables in $\vec{x}$, and n_i is the number of CCSs in Q1 before processing the $(i+1)$th event.

Suppose that we detect a fault at the lth event. There are n_i CCSs in Q1 before we process the $(i+1)$th event. At most $m \cdot n_i$ transitions are possible in this step. When a transition fires Algorithm 2 not only evaluates the predicate of this transition but also checks the assertion to find inconsistencies. The **check_consistency** procedure takes time of $r \cdot k \cdot |assert|_{max}$ to converge.

Proposition 2 *If $Q2$ becomes empty at the lth event, it takes time $O(m(|a|_{max} + |p|_{max} + r \cdot k \cdot |assert|_{max}) \sum_{i=0}^{l-1} n_i)$ for Algorithm 2 to detect a fault from an implementation machine.*

3.2.3 Fault Detection Capability of Passive Testing Algorithms

In active testing for protocol control portions, the faults are generally divided into output faults and next-state faults. In passive testing of protocol data portions with Event-driven EFSM, we classify faults into control-message faults and event-parameter faults. If a current event cannot be executed from the current state, it is a control-message fault. If the event can be fired while its parameters are inconsistent with the variable values, we call it an event-parameter fault.

Because the initial configuration of a machine is unknown when passive testing begins, not all the faults can be detected by passive testing. A passive testing algorithm is *correct* if it does not report nonexistent faults. A passive testing algorithm is *complete* if for any execution sequence such that the specification and the IUT produce observable different event sequences, the algorithm can detect the faults. It can be shown that the completeness of passive testing is undecidable [3]. A good passive testing algorithm must be correct and should be powerful enough, if not complete. "Homing" is a critical criteria to evaluate the power of different passive testing algorithms. If an algorithm can identify the current configuration of IUT, it can trace not only the state but also the precise variables values, which means more possibility to detect faults.

Algorithm 2 could detect both control-message faults and data-portion faults. Algorithm 2 is correct because it would not report a fault when there is none. Algorithm 2 preserves all the possible current configurations in each step. So if the implementation is correct, its configuration should be contained in one of the configuration sets built by Algorithm 2.

3.3. An example

In this section, we use an example EEFSM shown in Figure 5 to illustrate the execution of Algorithm 2. In this example EEFSM, each state will ignore any unspecified input and stay unchanged. The following table illustrates the execution of Algorithm 2 (including Proc1 and Proc2) step by step upon the observed event sequence $?c(8)\ !a(11)\ ?a(6)\ !d(11)$.

Table 1. Execution steps of Algorithm 2

#	*Event*	*set of CCS*
0		<S1,-,ϕ> <S2,-,ϕ> <S3,-,ϕ> <S4,-,ϕ> <S5,-,ϕ> <S6,-,ϕ> <S7,-,ϕ>
1	$?c(8)$	<S1,-,ϕ> <S2,-,ϕ> <S4, {R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]}, {u>v & seq=u+v & up=seq+8}> <S4, {R(u)=[0,7], R(v)=[0,7], R(seq)=[5,19], R(up)=[13,27]}, {u≤v & seq=19-u-v & up=seq+8}>
2	$!a(11)$	<S3, {R(v)=11, R(ctr)=0}, ϕ>, <S5, {R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19}, {u>v & 11=u+v}>, <S5, {R(u)=[1,7], R(v)=[1,7], R(seq)=11, R(up)=19}, {u≤v & 8=u+v}>
3	$?a(6)$	<S3, {R(v)=11, R(ctr)=0}, ϕ>, <S2, {R(seq)=11, R(up)=19, R(u)=6}, {v ≠ 6}>, <S6, {R(u)=2, R(v)=6, R(seq)=11, R(up)=19}, ϕ>
4	$!d(11)$	<S7, {R(u)=2, R(v)=6, R(seq)=11, R(up)=19, R(ctr)=0}, ϕ>

Initially, there are 7 CCSs. Each CCS corresponds to a state in the EEFSM and has an empty set of asserts. The initial interval of each variable is set according to its declaration and denoted by '-' in the table.

According to Algorithm 2, from the initial set of CCSs, upon input event $?c(8)$, CCS <S1,-,ϕ>, <S4,-,ϕ>, <S5,-,ϕ>, <S6,-,ϕ>, <S7,-,ϕ> will generate the same successor <S1,-,ϕ>, while CCS <S2,-,ϕ> will remain unchanged. CCS <S3,-,ϕ> will generate the following two successor CCSs, <S4, {R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]}, {u>v & seq=u+v & up=seq+8}> and <S4, {R(u)=[0,7], R(v)=[0,7], R(seq)=[5,19], R(up)=[13,27]}, {u≤v, seq=19-u-v, up=seq+8}>. The CCS set in step 2, 3 and 4 can be calculated following the same procedure. We can see from Table 1 that after 4 steps only one CCS is left and all the variable values are also decided.

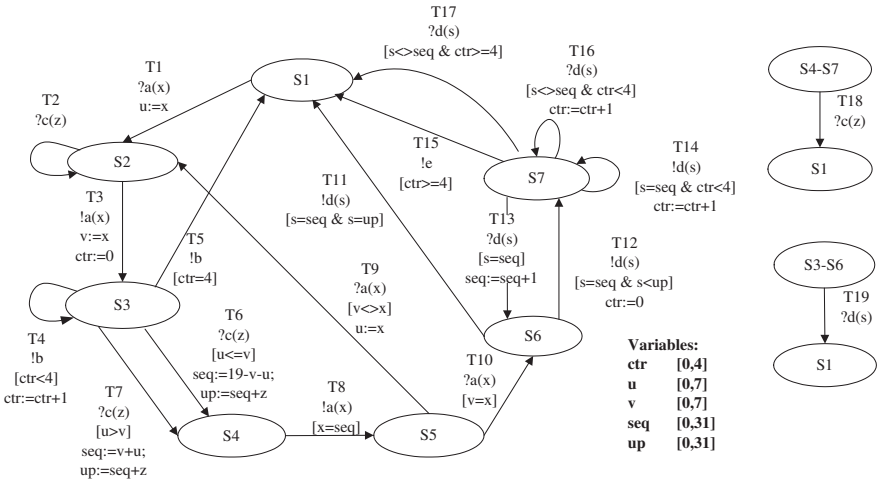


Figure 5. An Example EEFSM

Table 2. Consistency check and interval refinement

#	R(u)+R(-v)=[1,+∞)	R(u)+R(v)=[11,11]	Intervals
0	N/A	N/A	R(u)=[1,7] R(v)=[0,6]
1	R(u)=([1, +∞) -[-6,0]) ∩ [1,7]=[1,7] R(v)=(([1, +∞) -[1,7])/-1) ∩ [0,6]=[0,6]	R(u)=([11,11]-[0,6]) ∩ [1,7]=[5,7] R(v)=([11,11]-[5,7]) ∩ [0,6]=[4,6]	R(u)=[5,7] R(v)=[4,6]
2	R(u)=([1, +∞) -[-6,-4]) ∩ [5,7]=[5,7] R(v)=(([1, +∞) -[5,7])/-1) ∩ [4,6]=[4,6]	R(u)=([11,11]-[4,6]) ∩ [5,7]=[5,7] R(v)=([11,11]-[5,7]) ∩ [4,6]=[4,6]	R(u)=[5,7] R(v)=[4,6]

Now we show in detail how the successor CCS <S5, {R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19}, {u>v & 11=u+v}> in step 2 is generated from the CCS <S4, {R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]}, {u>v & seq=u+v & up=seq+8}> in step 1. First the current state in the CCS <S4, {R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]}, {u>v & seq=u+v & up=seq+8}> is S4. Upon the input event $!a(11)$, the transition T8 can be fired only when the predicate [x=seq] is evaluated not false. Assume this transition can be fired, we add [11=seq] to the assertion set of the current CCS (the current value of parameter x is 11) and start to check its consistency using Proc1. The assertion under check is {u>v & seq=u+v & up=seq+8 & seq=11}. We can easily decide that the value of seq is 11 and up is 19. So the assertion is simplified to {u>v & u+v=11}.

Table 2 shows how Proc1 check the consistency of the assertion and refine the intervals of variables u and v. We can see from Table 2, after 2 iterations, the intervals of variables u and v stay unchanged and the refinement is finished. Upon completion, no inconsistency is found and the variables are refined to {R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19}. The end state of the transition T8 is S5, which results in the successor CCS <S5, {R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19}, {u>v & 11=u+v}> in step 2.

4. Experimental Results

OSPF [9] is a widely used routing protocol in the Internet based on the Open Short Path First algorithm. The OSPF neighbor state machine is to maintain connections between two OSPF neighboring routers and exchange Link State Advertisements (LSA). Variables, such as sequence numbers, are used to record the current status of the connections.

Formally, an EEFSM can be transformed into an NFSM by extracting the variables from the specification. Thus the NFSM has no predicates, no actions and no input parameters. The OSPF NFSM model is presented in Appendix C. This NFSM model is like a mesh, revealing little information about the neighbor state machine. Note that the transformed NFSM is not equivalent to the original EEFSM.

We implemented three passive testing algorithms to compare their fault detection capabilities. They are the Algorithm 1 and Algorithm 2 presented in this paper and the NFSM algorithm presented in [4]. We applied these algorithms in the OSPF neighbor state machine testing.

The experiments are carried out in an experimental environment as shown in Fig 6. In the experimental network, we used *Socrates* [2] to generate link state information. *Socrates* is a software tool developed at Bell Labs. It can simulate an OSPF network and exchange OSPF link state information with routers. OSPF packets are captured from

the conversation between a Cisco router and the Router Under Test (RUT). Then the OSPF packets are decoded and sent to the Tester.

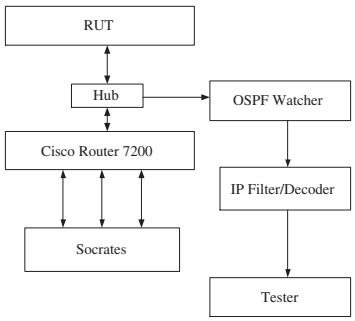


Figure 6. Experiment environment

Routers are high-speed devices. In order to capture all the OSPF packets in a conversation, we developed an OSPF Watcher to record all the related packets in a database. The IP filter and the Decoder analyze these captured OSPF packets and send useful event messages to the Tester. This test architecture can work in a real network and detect faults online while the RUT is connected in a network in operation.

We use the capability of *state homing* and *variable homing* to denote the efficiency of the passive testing algorithms. With the NFSM model, state homing is not obtained in all the 25 cases. There are 8 cases out of the 25 experiments that Algorithm 1 completed the homing phase, and 14 cases out of 25 that Algorithm 2 completed the homing phase. The experimental results are shown in Table 3.

Table 3. Experimental result

Algorithm	State Homing	Variable homing
NFSM algorithm	0/25	–
Algorithm 1	8/25	8/25
Algorithm 2	14/25	14/25

From the statistics, we can see that the NFSM approach is not a good fit for the passive testing of the behavior of OSPF. Passive testing algorithms using EEFSM model are much better than NFSM model. Algorithm 2 uses Interval to record the possible variable value regions and uses assertion to record the relations among variables. In this way it has a better knowledge of the running state machine compared to Algorithm 1. Obviously, Algorithm 2 is more powerful than Algorithm 1.

In the experiments, among the homing cases, Algorithm 2 takes an average of 4.4 steps to finish state homing, and takes an average of 11 steps to finish variable homing. The number of possible CCSs in Q1 never exceeds 10. This algorithm can detect faults in the OSPF neighbor state machine effectively.

5. Conclusion and future work

In a unit testing, a protocol system can be isolated and tested by actively applying inputs to reveal faults from the outputs in response. However, for a system in operation in a networked or integrated environment, often we could only passively observe the system behaviors to detect faults, and that naturally leads to the passive testing research and development activities.

In this paper, several passive testing algorithms on the EEFSM model and their applications to the OSPF neighbor state machines are presented. The deficiencies of existing algorithms are studied, and our algorithms prove to be much more effective in tracing the state and the variables values for detecting faults. We use symbolic logic methods to deal with the predicates and use assertions to record the relations among variables. The idea behind our approach is to refine the valid variable value sets using as much information as possible.

Fault location is an important step after fault detection so that detected faults can be identified and corrected. Yet little progress has been made.

Acknowledgement

We thank Xiaoliang Wei from Tsinghua University for implementing part of the algorithms and experimenting on OSPF.

References

[1] http://blrc.china.bell-labs.com/groups/ngi/projects/passive/.

[2] R. Hao, D. Lee, R. K. Sinha, and D. Vlah. Testing ip routing protocols - from probabilistic algorithms to a software tool. *Proceeding of FORTE-2000*, pages 249–264, October 2000.

[3] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.

[4] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. *Proceedings of IEEE International Conference on Network Protocols*, pages 113–122, October 1997.

[5] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proc. of the IEEE*, 84:1090–1123, Aug 1996.

[6] R. Miller. Passive testing of networks using a cfsm specification. *Proceedings of the IEEE International Performance, Computing and Communications Conference*, pages 111–116, February 1998.

[7] R. Miller and K. A. Arisha. On fault location in networks by passive testing. *Proceedings of the IEEE International Performance, Computing and Communications Conference*, pages 281–287, February 2000.

[8] R. E. Moore. *Methods and Applications of Interval Analysis*. Society for Industrial & Applied Mathematics, Philadelphia, 1979.

[9] J. Moy. *RFC 2328 OSPF Version 2.* 1998.
[10] M. Tabourier and A. Cavalli. Passive testing and application to the gsm-map protocol. *Information and Software Technology*, 41:813–821, September 1999.
[11] J. Wu, Y. Zhao, and X. Yin. From active to passive: Progress in testing of internet routing protocols. *Proceeding of FORTE-2001*, pages 101–116, August 2001.

A BNF of Predicate and Action

1. Integer expression is the atomic part of a predicate and an action.

   INTEGER_EXPRESSION ::=
   INTEGER
   | INTEGER_VARIBLE
   | EVENT_PARAMETER
   | (INTEGER * INTEGER_VARIBLE)
   | (INTEGER_EXPRESSION + INTEGER_EXPRESSION)
   | (INTEGER_EXPRESSION - INTEGER_EXPRESSION)

2. The predicates are logic expressions. A Simple predicate is defined as follow.

   p ::=
   TRUE
   | FALSE
   | (INTEGER_EXPRESSION==INTEGER_EXPRESSION)
   | (INTEGER_EXPRESSION>INTEGER_EXPRESSION)
   | (INTEGER_EXPRESSION<INTEGER_EXPRESSION)
   | (INTEGER_EXPRESSION$\geq$INTEGER_EXPRESSION)
   | (INTEGER_EXPRESSION$\leq$INTEGER_EXPRESSION)
   | (INTEGER_EXPRESSION$\neq$INTEGER_EXPRESSION)

   A predicate is defined recursively.

   P ::= p | (P $\wedge$ P) | (P $\vee$ P)

3. Each action is a set of assignments.

   ACTION ::= ASSIGNMENT; | ASSIGNMENT; ACTION | NULL

   ASSIGNMENT ::= VARIABLE:=INTEGER_EXPRESSION

B Operations on Integer Variable Values Denoted by Interval

Arithmetic operations on integer variable values can be found from [8]. Table 4 defines the evaluation of logic expressions on integer variables denoted by Interval. When a logic expression is evaluated, if the variables satisfy the condition, the corresponding result is returned. Otherwise the logic expression is evaluated to POSSIBLE.

Note: UA, LA are the upper and lower bounds respectively, so as UB, LB.

C. The Specification of the OSPF Neighbor State Machine

1. The EEFSM model

Table 4. Evaluation of logic expressions

Logic expression	result	Condition
A==B	TRUE	UA=LA=UB=LB
	FALSE	LA>UB or LB>UA
A$\geq$B	TRUE	LA$\geq$UB
	FALSE	UA<LB
A>B	TRUE	LA>UB
	FALSE	UA$\leq$LB
A$\leq$B	TRUE	UA$\leq$LB
	FALSE	LA>UB
A<B	TRUE	UA<LB
	FALSE	LA$\geq$UB
A$\neq$B	TRUE	LA>UB or LB>UA
	FALSE	UA=LA=UB=LB

There are a total of 84 transitions in the EEFSM model of the OSPF neighbor state machine in our experiment. The full version of the EEFSM specification can be found from [1] . One of the transitions is shown below. This transition is from state Exchange to state Loading. The event that triggered this transition is a Data Description packet.

NAME : T18
I_STATE : Exchange
EVENT : DD_rcv(RouterID, FlagI, FlagM, DDSeqNum)
F_STATE : Loading
PREDICATE : FlagI==0 & my_MS==1 & my_M+FlagM==0 & DDSeqNum==my_seq & my_DDsent==1
ACTION : my_NRID:=RouterID; my_seq:=DDSeqNum+1; ne_M:=FlagM; my_DDsent:=0;

2. The NDFSM model

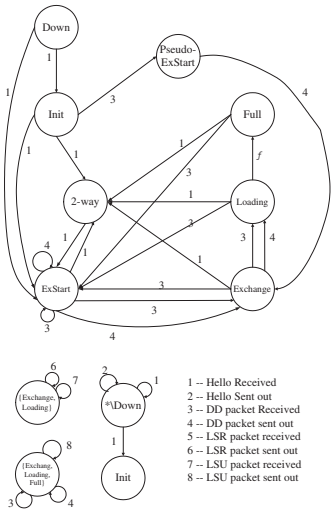


Figure 7. NDFSM model