A Passive Testing Technique with Minimized On-line Processing For Fault Management of Network Protocols

Tae-Hyong Kim

Kumoh National Institute of Technology, Gumi, Korea

Summary

Passive testing of a network protocol, the technique of detecting faults of the protocol implementation by monitoring its inputs and outputs when it is operating in a real communication network, has drawn attention lately because it does not interfere with the protocol development process. This paper proposes an extended finite state machine (EFSM)-based passive testing technique with good fault-detecting capability at a minimized on-line processing cost. Before testing, the proposed technique expands the given EFSM model to an expanded EFSM (XEFSM) model and derives its homing tree. During the testing we can find faults only by tracing that homing tree and the XEFSM with a given event sequence before and after the homing is accomplished respectively. In order to evaluate the effectiveness of the proposed technique we showed a simple example with the simple connection protocol (SCP) and also developed a passive testing system for the open shortest path first (OSPF) protocol. In the experiment with an OSPF implementation having intentional errors, the proposed technique showed a top-level fault detecting capability with less on-line processing. Therefore the proposed technique can be a good solution to real-time passive testing of a network protocol with rapid packet exchanges.

Key words:

Network fault management, Passive testing, Expansion of an EFSM

1. Introduction

Development of a reliable protocol implementation is a major issue in the protocol engineering area and conformance testing was standardized to check if a protocol implementation conforms to the standard and the specification of that protocol [1]. Owing to the formal description techniques such as the specification and description language (SDL) [2], a lot of testing techniques with formal methods have been proposed for conformance testing [3,4,5]. However there have been few success stories in industry that such testing methods were used in the development of a commercial network product. That is mainly due to the cost of conformance testing because the prior occupation of market share is crucial to the success of a product in the network product industry.

Passive testing has drawn attention lately because it does not interfere with the protocol development process. It is performed on a network product operating in a real communication network only by monitoring the inputs and outputs of that product. Due to the lack of controllability, passive testing may not be completed in finite time and thus it can be used as a fault management technique of a network. There have been several techniques proposed for passive testing such as homing approach [6], invariant approach [7,8] and backward checking approach [9]. The current techniques usually use the extended finite state machine (EFSM) as a protocol model and check the consistency of the captured packets with that model. They have some common problems to solve. First, EFSM-based passive testing techniques face high processing complexity due to semantic analysis of the specification model. If a passive testing technique does not handle a packet fast enough, it may fail to perform passive testing in real time at a heavy network traffic condition. Second, obtaining greater fault-detecting capability requires more sophisticated testing techniques in general. That is to say, there is a trade-off relation between reducing the processing complexity for real-time passive testing and increasing fault coverage.

This paper proposes a passive testing technique trying to solve that problem by performing complex semantic analysis of the specification before testing and minimizing on-line processing work during the passive testing. For that purpose, the proposed technique performs the expansion of an EFSM model and generates the homing tree with fault-patterns before testing.

This paper is organized as follows. Section 2 introduces basic definitions and notations, and also the concept of fault patterns. Then the detailed proposed passive testing technique is explained with an example of the simple connection protocol (SCP) [9] in section 3. Section 4 shows an experimental result and evaluation with a popular routing protocol, the open shortest path first (OSPF) neighbor state machine [10]. Finally conclusions are drawn in section 5.

N bovtdsjqu!sfdfjwfe!po!N bsdi!36-!3118/!
N bovtdsjqu!sfwjtfe!po!N bsdi!39-!3118/!

2. Preliminary Work

This section describes the basic definitions and notations required to explain the proposed technique and also introduces the concept of fault patterns as a generalization of invariants [7].

2.1 Basic definitions

As the specification model of a protocol, a normal form event-based EFSM (NF-EEFSM) is defined as follows.

Definition 1. An NF-EEFSM M is the 5-tuple $(S, s_0, \Sigma, \tilde{v}, T)$ where S is the finite set of logical states, $s_0(\in S)$ is the initial state, Σ is the finite set of events the element of which $\varepsilon(\in\Sigma)$ is denoted by $\sigma e(\tilde{p})$, where $\sigma\in\{?,!\}$, e is a message name, and $\tilde{p}$ is the finite set of event parameters, $\tilde{v}$ is the finite set of variables, T is the finite set of transitions, where the label of a transition $t(\in T)$ is denoted by the 5-tuple $(s_s, s_f, \varepsilon, P(\tilde{v},\tilde{p}), A(\tilde{v},\tilde{p}))$ in which s_s and s_f are the start and the final state of t respectively, and $P(\tilde{v},\tilde{p})$ and $A(\tilde{v},\tilde{p}))$ are the predicate and the action of t respectively.

Note that $\Sigma=\Sigma_I\cap\Sigma_O$, where Σ_I and Σ_O are the finite sets of input and output events respectively such that $?e(\tilde{p})\notin\Sigma_O$ and $!e(\tilde{p})\notin\Sigma_I$. For the implementation of the proposed technique we normalize $P(\tilde{v},\tilde{p})$ and $A(\tilde{v},\tilde{p})$ as follows. $P(\tilde{v},\tilde{p})$ is a disjunctive normal form of simple predicates π_i denoted by $\boldsymbol{a_i}\tilde{v} + \boldsymbol{b_i}\tilde{p} + c_i \sim 0$, where $\sim\in\{<, >, \leq, \geq, =, \neq\}$, $\boldsymbol{a_i}$ and $\boldsymbol{b_i}$ are integer matrices whose sizes are $1\times|\tilde{v}|$ and $1\times|\tilde{p}|$ respectively, and c_i is an integer constant. $A(\tilde{v},\tilde{p})$ is a set of linear assignment equations, $\tilde{v} \leftarrow \boldsymbol{a_j}\tilde{v} + \boldsymbol{b_j}\tilde{p} + \boldsymbol{c_i}$, where $\boldsymbol{a_j}$, $\boldsymbol{b_j}$, and $\boldsymbol{c_j}$ are integer matrices whose sizes are $|\tilde{v}|\times|\tilde{v}|$, $|\tilde{v}|\times|\tilde{p}|$, and $|\tilde{v}|\times 1$ respectively. In the specification of a real network protocol, nonlinear equations may be used in the predicate or action such as the absolute value function or the set operations. Such nonlinear equations can be transformed to the equivalent linear equations by using the domain propagation technique [11].

The following assumptions on the NF-EEFSM are also used to reduce the problem size when handling an NF-EEFSM.

Assumption 1. An NF-EEFSM is deterministic and strongly connected [3].

Assumption 2. Each loop within an NF-EEFSM is either an unconditional loop where each iteration of the loop generates the same global control state subspace (type 1), a conditional loop where the number of iterations of the loop is not bounded above and each iteration of the loop generates the same global control subspace (type 2), or a conditional loop where the number of iterations of the loop is bounded above (type 3).

In assumption 1, a transition t where there exists a simple predicate π_i in $P(\tilde{v},\tilde{p})$ such that $\boldsymbol{a_i}\neq\boldsymbol{0}$ and $\boldsymbol{b_i}=\boldsymbol{0}$ is called a conditional transition, and an unconditional transition otherwise. Additionally, a loop if all transitions constructing that loop are unconditional transitions is called an unconditional loop, and a conditional loop otherwise.

2.2 Generalization of invariants: Fault Patterns

The invariant approach [7] checks a captured packet trace if a part of that trace violates the invariants generated from the specification model. However, the classification of invariants, forwards and backward, was done with the logical considerations on the invariants and thus that classification is not complete. That is to say, there are faulty message sequences that cannot be detected easily with the existing invariant approach. This paper, instead of such invariants, uses fault patterns for packet traces and introduces a simple notation for representing fault patterns.

Definition 2. The set of fault patterns of an NF-EEFSM M, Φ_M is defined by $\Phi_M = \{\varphi \mid \varphi$ is a packet trace including inputs and outputs that cannot be observed from $M\}$, where a fault pattern φ is represented by $\Xi_i(\varepsilon_i)$ in which Ξ is the concatenation generator of events such that $\Xi_{i=1\ldots n}(\varepsilon_i) = \varepsilon_1\ @\ \varepsilon_2\ @\ \ldots\ @\ \varepsilon_n$ where @ is the concatenation operator.

In the notation of fault patterns, ε_i can be represented as an element of $\wp(\{\sigma_i e_i \mid \sigma_i e_i \in \Sigma\})\cap\wp(\{\sigma_i \underline{e_i} \mid \sigma_i e_i \in \Sigma\}) - \{\varnothing\}$, where $\wp(\cdot)$ is the powerset operator, $\sigma_i\underline{e_i}$ is the negative event of $\sigma_i e_i$ denoting all events $\sigma_j e_j \in \Sigma$ such that $\sigma_j = \sigma_i$ and $e_j \neq e_i$. $\{\sigma e\}$ and $\{\sigma\underline{e}\}$ can be represented as σe and $\sigma\underline{e}$ respectively for simplicity.

Example 1. Let the fault pattern set of an NF-EEFSM M, $\Phi_M = \{?a!x, ?\underline{a}!\{y,z\}, ?b!\underline{x}, ?\underline{a}!\underline{z}?b!y, ?\underline{a}!\underline{z}?\underline{b}!y, ?c!\underline{z}?\{\underline{b},\underline{c}\}!y\}$. An observation of any element of Φ_M from an implementation of M indicates the fault of that implementation at any time. $?\underline{a}!\{y,z\}$ and $?b!\underline{x}$ correspond to backward and forward invariants respectively. $?\underline{a}!\underline{z}?b!y$ and $?\underline{a}!\underline{z}?\underline{b}!y$ are clearly differently each other but their difference may not be represented distinctly with the invariant classification. In addition, $?c!\underline{z}?\{\underline{b},\underline{c}\}!y$ cannot be represented by an invariant.

The negative event representation in fault patterns can be extended to a sequence of events or parameters in an event. $\underline{?a!x?b}!y$ represents all event sequences that ends with the event $!y$ and the preceding event sequence of that $!y$ is not $?a!x?b$, namely $\underline{?a!x?b}!y = \{?\underline{b}!y, !\underline{x}?\underline{b}!y, ?\underline{a}!\underline{x}?\underline{b}!y\}$. $?a(1)!\underline{x(1)}$ and $?a(1)!x(\underline{1})$ represent all output events that is not $x(1)$ and the output event x whose parameter is not 1 respectively, following the event $?a$.

3. The Proposed Passive Testing Technique

The goal of this paper is to develop a passive testing technique with good fault-detecting capability and minimized on-line processing for real-time passive testing of a network product. To obtain that goal, we first expand the NF-EEFSM model of a protocol to an equivalent FSM in order to transfer the complex semantic analysis of that machine to off-line work before testing. And the most of homing and pattern matching work is also moved to off-line world by generating the homing tree of that expanded NF-EEFSM. If that homing tree is complete, it can possess the maximal fault-detecting power.

3.1 Overview of the proposed technique

The working flow of the proposed passive testing technique is illustrated in figure 1.

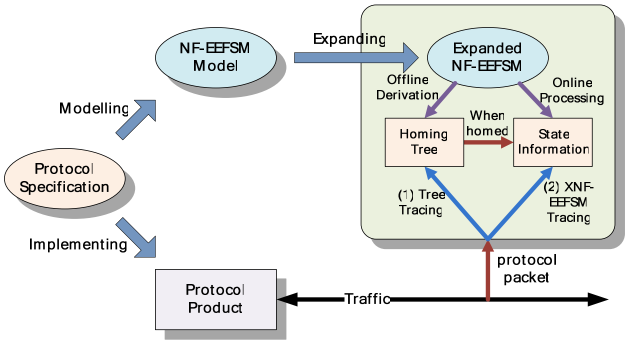


Fig. 1 Flow diagram of the proposed technique

First, from the specification of the target protocol, its NF-EEFSM model is generated. Then that NF-EEFSM is transformed to an equivalent FSM where all transitions are unconditional, which is called an expanded NF-EEFSM (XNF-EEFSM). That expansion will be explained in detail at subsection 3.2.

Instead of executing the homing procedure on-line during the testing, we generate the adaptive homing tree of that XNF-EEFSM before testing. During the testing, when the tester captures a packet to or from the target protocol product, it has only to trace that homing tree with that packet. That may take much smaller processing time than the existing on-line processing. How to generate the homing tree and how to use it for homing and fault detection will be described in subsection 3.3.

When the homing is accomplished, the current logical state of the XNF-EEFSM and the current values of all control variables are identified. Hence the tester can validate the correctness of the protocol product only by tracing that XNF-EEFSM with that information with the subsequent protocol packets. Note that the proposed technique has a tracing-level processing complexity during the testing. The overall procedure of the proposed technique is represented in algorithm 1.

Algorithm 1. The proposed passive testing technique

```
• Inputs: the specification of the protocol $M_0$, the event
  sequence observed at the target product $\Xi_i(\varepsilon_i)$ $(1 \le i \le n)$
• Output: either 'ErrorDetected' or 'NoError'
• Variable: the result of homing $Result$
Step.1: Construct the NF-EEFSM $M_1$ from $M_0$
Step.2: XNF-EEFSM $M \leftarrow$Call Algorithm 2 $(M_1)$;
Step.3: Homing tree $H_M \leftarrow$Call Algorithm 3 $(M)$;
Step.4: $Result \leftarrow$Call Algorithm 4 $(H_M, \Xi_i(\varepsilon_i))$;
 switch ($Result$)
   case 'ErrorDetected': return 'ErrorDetected';
   case $Homed(i,s)$:
     for $j \leftarrow 1$ to $n$
       $\sigma_k e_k \leftarrow nextEvent(M, s, \varepsilon_i)$;
       $s \leftarrow nextState(M, s, \varepsilon_i)$;
       if $\sigma_k e_k = \sigma_{j+1} e_{j+1}$ then continue;
       else return 'ErrorDetected'; endif
     endfor
     return 'NoError';
 endswitch
```

3.2 Expansion of an NF-EEFSM

The expansion procedure of an NF-EEFSM is based on the expansion method of an NF-EFSM shown in [3]. First, some notation and functions are introduced to explain the expansion algorithm of this paper. Δ denotes the domain constructed from all control variables in $\tilde{v}$ and Λ denotes the domain constructed from all parameters in $\tilde{p}$ of the input events. The subset of Δ allowed at a state will be called the *domain* of the state. The term *precondition* of a transition t_i, denoted P_i, is used to mean the predicate $P(\tilde{v},\tilde{p})$ of t_i. The functions $R_\Delta(\cdot)$: $P(\tilde{v},\tilde{p}) \rightarrow \wp(\Delta)$ and $R_\Lambda(\cdot)$: $P(\tilde{v},\tilde{p}) \rightarrow \wp(\Lambda)$ transform a predicate in the DNF form to the subdomains of Δ and Λ that satisfy that predicate respectively. Their inverse functions $R_\Delta^{-1}(\cdot)$: $\wp(\Delta) \rightarrow P(\tilde{v},\tilde{p})$ and $R_\Lambda^{-1}(\cdot)$: $\wp(\Lambda) \rightarrow P(\tilde{v},\tilde{p})$ generate the predicates in the DNF form that determine the input subdomains of Δ and Λ. The term *postcondition* of a transition t_i, denoted by $Q_i(\cdot)$: $\wp(\Delta) \times \wp(\Lambda) \rightarrow \wp(\Delta)$, is the function that derives the domain in Δ, according to the action $A(\tilde{v},\tilde{p})$ of t_i, given two subdomains of Δ and Λ. The inverse functions $Q_{\Delta i}^{-1}(\cdot)$: $\wp(\Delta) \rightarrow \wp(\Delta)$ and $Q_{\Lambda i}^{-1}(\cdot)$: $\wp(\Delta) \rightarrow \wp(\Lambda)$ derive the domains in Δ and Λ respectively, according to the inverse action $A^{-1}(\tilde{v},\tilde{p})$ of t_i, given a subdomain of Δ. $d(\cdot)$:$S \rightarrow \wp(\Delta)$ is the function generating the domain of a state in Δ, and $s_s(\cdot)$:$T \rightarrow S$ and $s_f(\cdot)$:$T \rightarrow S$ are the starting state and final state functions of a transition respectively. The expansion algorithm of an NF-EEFSM is as follow under the assumption that all the postcondition functions and their inverse functions can be evaluated symbolically in any domain considered.

Algorithm 2. The expansion of an NF-EEFSM
- Input: an NF-EEFSM M_1
- Output: the equivalent XNF-EEFSM M

Step.1: Partition the domain of a state s in M_1 that has at least two conditional transitions originating from it as follows: Let the conditional transitions $t_1, t_2, \ldots, t_n$, $(n\geq 2)$ originating from state s have preconditions $P_1, P_2, \ldots, P_n$ respectively. Each subdomain, $\{\delta^s{}_X \mid X\subseteq\{1, \ldots, n\}\wedge X\neq\varnothing\}$ is given by $\delta^s{}_X = R_\Delta((\wedge_{i\in X}P_i)\wedge(\wedge_{i\notin X}\neg P_i))$. If the final non-empty disjoint subdomains are $\delta^s{}_1, \ldots, \delta^s{}_m$ $(m\leq 2n\text{-}1)$, split the state s to $s_1, \ldots, s_m$ whose domains are $\delta^s{}_1, \ldots, \delta^s{}_m$ respectively. If this is the first iteration, repeat this step for all the states from which there are outgoing conditional transitions. After the first iteration, priority is given to states that are not within any type 3 loop, if there exist such states; otherwise, the state to be split is selected among states that are within type 3 loops.

Step2: Rearrange transitions related to the split states. If a state s is split into $n(\geq 2)$ states, $s_1, \ldots, s_n$, remove each transition t going from or to the state s. Then, for each removed transition t_a going from the state s to a state $s_f(\neq s)$, make n temporary transitions going from s_i $(1\leq i\leq n)$ to s_f whose labels are the same as that of the removed transition. For each removed transition t_b going to the state s from a state $s_s(\neq s)$, make n temporary transitions going from s_s to s_i $(1\leq i\leq n)$ whose labels are the same as that of the removed transition. For each removed transition t_c going from and to the same state s, make n^2 temporary transitions going from each s_i $(1\leq i\leq n)$ to each s_j $(1\leq j\leq n)$ whose labels are the same as that of the removed transition.

Step 3: For each temporary transition t_i, make it permanent or discard it depending on the following cases:
- Case A. If $d(s_s(t_i))\cap R_\Delta(P_i)=\varnothing$ or $Q_i(d(s_s(t_i)),R_\Lambda(P_i)) \cap d(s_f(t_i))=\varnothing$, discard t_i.
- Case B. If $d(s_s(t_i))\subseteq R_\Delta(P_i)$ and $Q_i(d(s_s(t_i)),R_\Lambda(P_i)) \subseteq d(s_f(t_i))$, make t_i unconditional.
- Case C. If $d(s_s(t_i))\subseteq R_\Delta(P_i)$ and $Q_i(d(s_s(t_i)),R_\Lambda(P_i))$ □$d(s_f(t_i))$ and $Q_i(d(s_s(t_i)),R_\Lambda(P_i))\cap d(s_f(t_i))\neq\varnothing$: if $d(s_s(t_i)) \subseteq R_\Delta(P_i')$ then make t_i unconditional; otherwise, make t_i conditional with the predicate P_i', where $P_i' = d(s_s(t_i)) \cap Q_i^{-1}(d(s_f(t_i)))$.

Step 4: If a transition t_i which was determined to be permanent at step 3 has the event $\sigma_i e_i$ carrying a parameter $p_i(\in\tilde{p})$ such that $R'_\Lambda(p_i)\subset\Lambda$ where $R'_\Lambda(p_i)$ is the domain of p_i which can be allowed in Λ, update the predicate of t_i to $R_\Lambda^{-1}(R'_\Lambda(p_i))$.

Step 5: If the initial state is split, determine which the new initial state is now among those split states. Remove all states that cannot be reached from the initial state. If there are no conditional transitions, terminate; otherwise, return to step 1.

3.3 The homing tree for fast fault checking

The proposed technique uses a homing tree customized for performing both homing and fault detection. That homing tree is based on a tree representing an adaptive homing sequence [12] and has additional fault detection information. The algorithm to generate the homing tree from an XNF-EEFSM is as follows.

Algorithm 3. Generation of the homing tree
- Input: an XNF-EEFSM M
- Output: the homing tree H_M of M
- Variables: the logical state set for a node of the homing tree S_t (the initial value: S), the node set of the homing tree candidate for the extension CS (the initial value: S), an extended event set □

Step.1: For each event $\varepsilon_i = \sigma_i e_i$ $(\in\Sigma)$ in M, and transitions t_j $(1\leq j\leq n)$ which have that same event ε_i carrying parameter p_j, partition Λ to subdomains $\delta^s{}_{i,k}$ $(1\leq k\leq m_i)$ such that $\delta^s{}_{i,k} = ((\cap_{j\in X} R'_\Lambda(p_j) \cap (\cap_{i\notin X} R'_\Lambda(p_j)^C)$, $X\subseteq\{1, \ldots, n\}$ and $X\neq\varnothing$, where $R'_\Lambda(p_j)^C = \Lambda - R'_\Lambda(p_j)$. Then, for each split subdomain $\delta^s{}_{i,k}$, add the event $\sigma_i e_i(\delta^s{}_{i,k})$ to □.

Step.2: Get a node ν in CS and assign the label of that node to S_t. For each event ε_i in □ of a given type, input and output by turns, create a leaf l_i originating from ν, attach the label ε_i on l_i, and create an unlabeled node ν_i at the open end of l_i.

Step.3: Put the label on each node ν_i generated at step 2 as follows. For the leaf l_i terminating at ν_i, if its label event ε_i can occur at some states in the label state set S_t of the originating node ν according to M, the set of possible present states $S_f(\subseteq S, \neq\varnothing)$ after executing that event ε_i become the label of node ν_i. If the event ε_i cannot occur at any state in S_t, attach the label 'Error' on node ν_i. If the label of node ν_i appears for the first time and the number of states in S_f is greater than 2, add ν_i to CS.

Step.4: Remove the node ν from CS. If CS is empty, terminate; otherwise, go to step 2.

Extension of the events to the scope of parameters enables the fault detection for a packet trace with faulty parameters. Terminal leaves of the homing tree are of one type among the followings: the homed node whose label has single state (type 1), the error node with the label 'Error' (type 2), and the recurrent node with the same label as an existing node (type 3). When we trace the homing tree with a captured packet trace, if the tracing stops at a terminal node, we can see the result directly from its label. Note that fault patterns can be obtained from each error node of the homing tree. The final fault patterns can be derived by rearranging fault patterns with consideration of their inclusion relations. The algorithm for homing and error detection with the homing tree is as follows.

Algorithm 4. Error detection with the homing tree

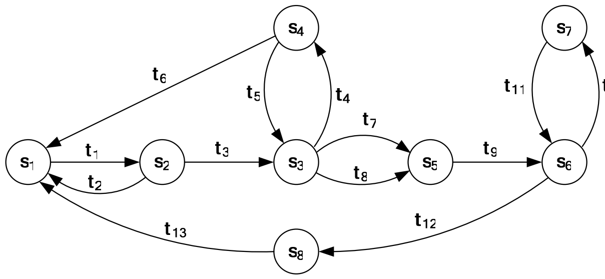
```
• Inputs: a homing tree $H_M$, an event sequence $\Xi_i(\varepsilon_i)$ ($1 \le i \le n$)
• Output: either 'ErrorDetected' or $Homed(traceLocation, currentState)$
• Variable: the variable to store the location of a node in
  the homing tree $n_j$ (the initial value: the head of $H_M$)
begin
  $i \leftarrow 0$, $j \leftarrow 1$; /* initialization */
  TOP: do
    $i \leftarrow i+1$; /* get the next event */
    $n_{j+1} \leftarrow trace(H_M, n_j, \varepsilon_i)$;
    $j \leftarrow j+1$; /* move to the next node */
  while ($n_j$ is not a terminal node);
  switch ($label(n_j)$)
    case state $s$: return $Homed(i,s)$;
    case 'Error': return 'ErrorDetected';
    otherwise:
      $n_{j+1} \leftarrow search(H_M, label(n_j))$;
      goto TOP; /* repeat the tracing */
  endswitch
end
```

In algorithm 4, the function *trace*(H_M, n_j, ε_i) derives the next node of n_j by tracing the homing tree H_M with the event ε_i, and with the function *search*(H_M, *label*(n_j)) we obtain the exiting node which has the same label as the node n_j in H_M.

3.4 An example: simple connection protocol (SCP)

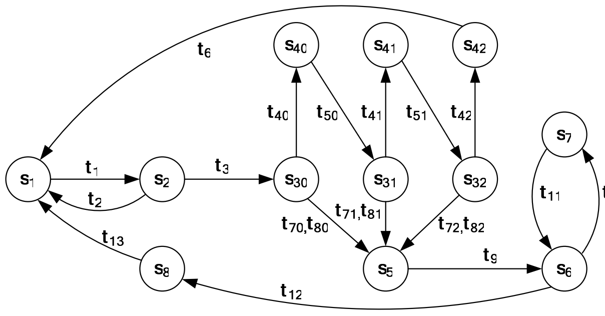
In order to demonstrate the working of the proposed technique, the SCP is used which has been often used as an example in the literature on the passive testing [9]. The SCP makes a the quality of service (QoS) configuration request for the connection to the lower layer upon the request of the upper layer and informs the upper layer of the acceptance result of that request from the lower layer. Figure 2 shows the NF-EEFSM of the SCP which has 8 states and 13 transitions. Note that six transitions have nonempty predicates but only two are conditions among them.

By algorithm 2, the XNF-EEFSM of the SCP was generated from that NF-EEFSM as shown in Figure 3, where the transition labels which have not been updated were omitted for simplicity. It has 12 states and 17 transitions. Then, the homing tree of the SCP was generated from that XNF-EEFSM by algorithm 3, which is shown in Figure 4. It has 11 terminal nodes which include 8 homed nodes and 3 error nodes. Therefore, the homing is accomplished by tracing 2.125 packets on average in case of the SCP. From the homing tree, two fault patterns were obtained as follows: (1) ?refuse !<u>{CONcnf, connect}</u> (2) ?refuse !connect ?<u>{accept, refuse}</u>.



t_1=(s_1, s_2, ?CONreq(qos), {}, {TryCount:=0, ReqQos:=qos, FinQos:=0})
t_2=(s_2, s_1, !NONsupport(ReqQos), {ReqQos>1}, {})
t_3=(s_2, s_3, !connect(ReqQos), {ReqQos<=1}, {})
t_4=(s_3, s_4, ?refuse, {}, {})
t_5=(s_4, s_3, !connect(ReqQos), {TryCount!=2}, {TryCount:=TryCount+1})
t_6=(s_4, s_1, !CONcnf(-), {TryCount=2}, {})
t_7=(s_3, s_5, ?accept(qos), {qos<=ReqQos}, {FinQos:=qos})
t_8=(s_3, s_5, ?accept(qos), {qos>ReqQos}, {FinQos:=ReqQos})
t_9=(s_5, s_6, !CONcnf(+,FinQos), {}, {})
t_{10}=(s_6, s_7, ?Data, {}, {}), t_{11}=(s_7, s_6, !data(FinQos), {}, {}),
t_{12}=(s_6, s_8, ?Reset, {}, {}), t_{13}=(s_8, s_1, !abort, {}, {}),

Fig. 2 The NF-EEFSM model of the SCP



t_{40}=(s_{30}, s_{40}, ?refuse, {}, {}), t_{41}=(s_{31}, s_{41}, ?refuse, {}, {}), t_{42}=(s_{32}, s_{42}, ?refuse, {}, {})
t_{50}=(s_{40}, s_{31}, !connect(ReqQos), {}, {TryCount:=TryCount+1})
t_{51}=(s_{41}, s_{32}, !connect(ReqQos), {}, {TryCount:=TryCount+1})
t_{70}=(s_{30}, s_5, ?accept(qos), {qos<=ReqQos}, {FinQos:=qos})
t_{71}=(s_{31}, s_5, ?accept(qos), {qos<=ReqQos}, {FinQos:=qos})
t_{72}=(s_{32}, s_5, ?accept(qos), {qos<=ReqQos}, {FinQos:=qos})
t_{80}=(s_{30}, s_5, ?accept(qos), {qos>ReqQos}, {FinQos:=ReqQos})
t_{81}=(s_{31}, s_5, ?accept(qos), {qos>ReqQos}, {FinQos:=ReqQos})
t_{82}=(s_{32}, s_5, ?accept(qos), {qos>ReqQos}, {FinQos:=ReqQos})

Fig. 3 The XNF-EEFSM of the SCP

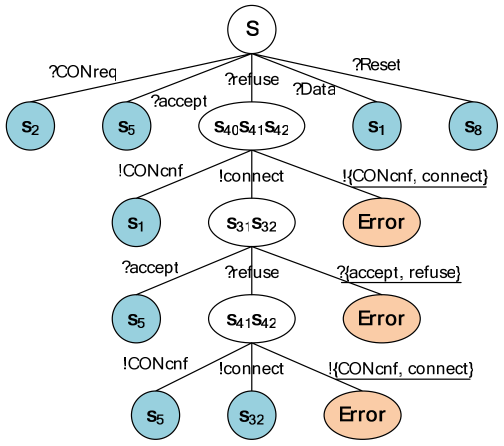


Fig. 4 The homing tree of the SCP XNF-EEFSM

In order to estimate the fault detecting capability of the proposed technique, we performed a simple passive testing on the SCP with the event sequence "?CONreq(1) !connect(1) ?refuse !CONcnf(-)" which has been used for that purpose in [6]. Given the event '?CONreq(1)', the homing is accomplished and the current state is identified as s_2 in the proposed technique. Now we have only to trace the XNF-EEFSM of the SCP from s_2. The next event '!connect(1)' is allowed in that XNF-EEFSM and the current state is updated to s_{30}. Then the next event '?refuse' is also allowed and s_{40} is now the current state. The final event '!CONcnf(-)' is not allowed at s_{40} and we can detect a fault. Neither the existing homing approach nor the original invariant approach can detect a fault with this event sequence [6]. The backward checking and the constrained invariant checking approaches can detect a fault but they may require considerable amount of on-line processing.

4. Experimental Results

For the validation of efficacy of the proposed passive testing technique, we applied it to a real routing protocol, the OSPF protocol in a real network environment. In this experiment, we construct a testing system and environment with the proposed technique, which will be explained with the result and evaluation in this section.

4.1 The OSPF neighbor state machine

We used the OSPF neighbor state machine that maintains connections between two neighboring OSPF routers and exchanges link state information [10]. First we construct the NF-EEFSM model of the OSPF neighbor state machine. The OSPF often uses set operators for checking the routing information it receives. Instead of an expression with a set operator, several equivalent linear expressions with a finite array are used in the NF-EEFSM. Figure 5 shows the simplified NF-EEFSM of the OSPF neighbor state machine. The complete NF-EEFSM has 8 logical states and 90 transitions.

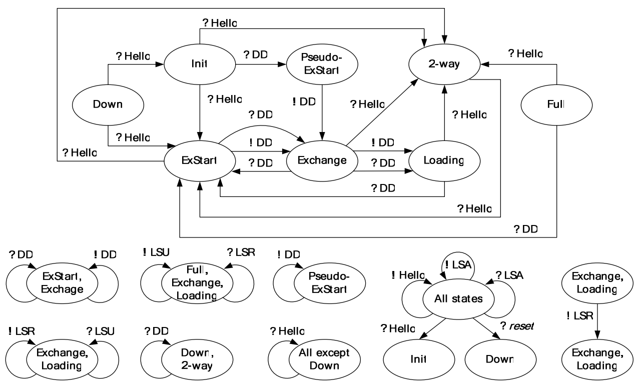


Fig. 5 The simplified NF-EEFSM of the OSPF neighbor state machine

There are 60 transitions with nonempty predicates but only 16 transitions are conditional in that NF-EEFSM. We transformed it to the equivalent XNF-EEFSM with algorithm 2. The XNF-EEFSM of the OSPF neighbor state machine has 21 states and 245 transitions. Then, its homing tree was generated by algorithm 3. The homing tree of the OSPF XNF-EEFSM has 117 terminal nodes including 88 homed nodes and 29 error nodes. Accordingly the homing of the OSPF XNF-EEFSM can be accomplished after tracing 3.625 packets on average. From that homing tree, we also obtained 21 fault patterns.

4.2 Testing system and environment

The structure of the passive tester for the OSPF protocol we implemented with the proposed technique is depicted in Figure 6.

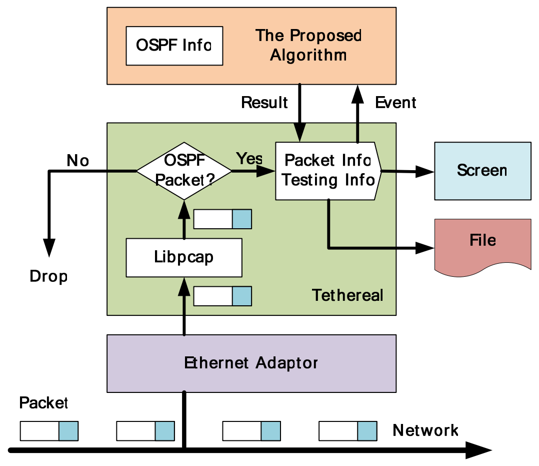


Fig. 6 The structure of the implemented passive tester

The passive tester was implemented in a laptop computer with Linux OS. The *libpcap* library [13] and the open-source tool *tetherial* [14] were used in our passive testing tool so as to capture and decode protocol packets from the network. Packets arrived at the Ethernet adapter of the tester are copied to the internal buffer by the libpcap libaray and OSPF packets are picked up by the filters of the tethereal tool. From OSPF packets, the corresponding events are created and given to the proposed algorithm which was programmed. The proposed algorithm part has the OSPF information required for our passive testing such as the OSPF XNF-EEFSM and its homing tree. Given an event, it runs the algorithms and sends the result to the tethereal. We can see the information of the processing can be seen with the output part based on the tethereal. If an error is detected, the tester displays that error and initializes itself for resuming the testing.

Figure 7 shows the environment for our experiments. We constructed a experimental network with two OSPF routers, our passive tester, and an OSPF software router, the target product, based on the open-source router emulator *Zebra* [15]. In order to produce intentional errors we slightly modified the message output part of the Zebra *ospfd* tool. With that modification, two types of errors, the message type error and the message parameter error, are generated randomly at a given rate. That error generation information is saved in a file for later checking.

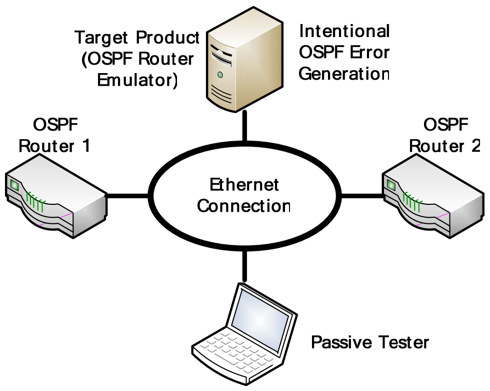
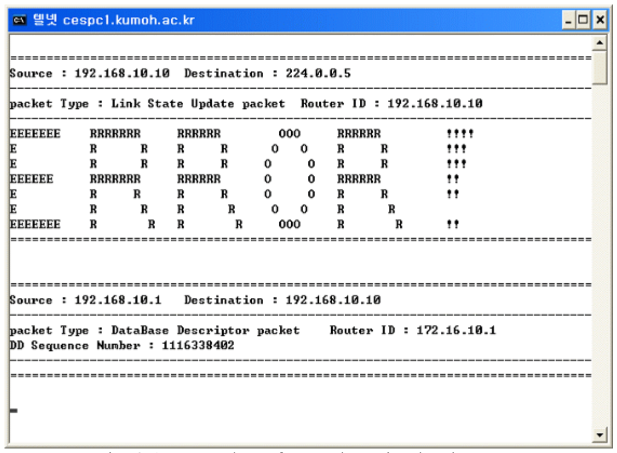


Fig. 7 The experiment environment for passive testing

4.3 The result and evaluation

In our experiments, the tester detected all single errors of the target product which were generated intentionally except a few cases. In case of a series of errors, the subsequent errors after the first one may not be detected. Figure 8 shows a part of testing log file which displays error detection message by the tester.



```
================================================================================
Source : 192.168.10.10  Destination : 224.0.0.5
--------------------------------------------------------------------------------
packet Type : Link State Update packet  Router ID : 192.168.10.10
--------------------------------------------------------------------------------
EEEEEEE    RRRRRRR     RRRRRR        000     RRRRRR         !!!!
E          R      R    R      R     0   0    R     R        !!!
E          R      R    R      R    0     0   R     R        !!!
EEEEEE     RRRRRRR     RRRRRR      0     0   RRRRRR         !!
E          R     R     R      R    0     0   R     R        !!
E          R      R    R       R    0   0    R      R
EEEEEEE    R       R   R        R    000     R       R      !!
================================================================================


================================================================================
Source : 192.168.10.1   Destination : 192.168.10.10
--------------------------------------------------------------------------------
packet Type : DataBase Descriptor packet    Router ID : 172.16.10.1
DD Sequence Number : 1116338402
--------------------------------------------------------------------------------
================================================================================
```

Fig. 8 A screenshot of error detection by the tester

The tester did not detect a few errors and sometimes it reports errors wrongly for the correct packet. We found that that problem is due to the imperfect capture capability of the libpcap library. It may miss a packet if the packet exchange is very fast. According to [16], the libpcap library shows unsatisfactory capturing power when a packet rate is high and the packet size is small. The *PF_RING* socket may be used to reduce this problem [16].

Now we evaluate the competence of the proposed technique by comparison with the existing techniques in several aspects. As for the fault coverage, since the proposed technique uses a kind of homing approach, it can detect every faults of the target product after it accomplishes the homing. Even before homing, it can detect the event parameter inconsistency with the homing tree extended to the parameter domain. Hence it has better fault detecting capability than the original homing approach. We guess it has the same fault coverage as the backward checking approach which performs an exhaustive check to find a fault. The constrained invariant checking approach also has very good fault coverage but that depends on the invariants it derives, which may not be complete. The most distinctive merit of the proposed technique is that it has lower on-line processing cost than the existing techniques. It has the trace-level processing complexity with the homing tree and the XNF-EEFSM of the target protocol. Therefore it is suitable for real-time passive testing in a heavy network traffic situation. The cost it should pay for such advantages is a large amount of off-line processing before testing. However that can be automated and does not affect the passive testing itself. Table 1 summarizes the comparison between the passive testing techniques.

Table 1 Comparison between the passive testing techniques

	Homing	*Invariant original**	*Invariant constrained**	*Backward checking*	*Proposed*
Fault coverage	good	moderate	very good	very good	very good
On-line cost	high	moderate	high	very high	low
Off-line cost	low	high	very high	low	very high

* Depends on the number of invariants used

5. Conclusions

Passive testing is very useful for fault management of a network product which has not been tested sufficiently during the development process. Moreover it does not interfere with the target product in a network because it tests the product only by observing packets going to and from that product. There have been several studies to obtain a good passive testing technique with high fault detecting capability. Such a technique, however, may require a lot of processing work during the testing, which may affect real-time passive testing at a heavy network traffic condition.

This paper proposed a passive testing technique based on an EFSM model with minimized on-line processing. It moves a considerable amount of on-line processing work to the off-line world. First, it transforms the NF-EFSM model of the target protocol to the equivalent XNF-EEFSM which is a kind of FSM where all transitions are executable at each state without complex semantic analysis. Then it also derives the homing tree of that XNF-EEFSM before testing. During the testing, we can perform homing and fault detection only by tracing that tree with a given event sequence. After the homing is accomplished, we have only to trace the XNF-EEFSM with that event sequence for the fault management of the target product. An example with the SCP and an experiment with the OSPF neighbor state machine show that the proposed technique has as good fault coverage as the backward checking or the constrained invariant checking approaches which performs a thorough check for fault detection. As the proposed technique has lower on-line processing cost than the existing techniques, it can be a good solution to real-time passive testing of a network protocol product with rapid packet exchanges.

We are planning to apply the proposed technique to other popular network protocol such as TCP [17] and to make a precise comparison with the existing approaches. We are also interested in handling the false error detection due to missed packets and locating the faults for fault correction.

Acknowledgment

This work was supported by the Korea Research Foundation Grant Funded by the Korean Government (MOEHRD) (D00568).

References

[1] ISO, "OSI Conformance Testing Methodology and Framework", IS-9646, 1991

[2] ITU, "Specification and Description Language", ITU-T Recommendation Z.100, 2000

[3] Hierons, R. M., Kim, T.-H., and Ural, H., "On the Testability of SDL Specifications", Computer Networks, To Appear, 2004

[4] Hierons, R. M., and Ural, H., "UIO Sequence Based Checking Sequences For Distributed Test Architectures", Information and Software Technology, Vol.45, 2003, pp.793-803

[5] M. Uyar, M. Fecko, A. Duale, P. Amer, A, Sethi, "Experience in Developing and Testing Network Protocol Software Using FDTs", Information and Software Technology, Vol.45, 2003, pp.815-835

[6] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu, X. Yin, "A Formal Approach for Passive Testing of Protocol Data Portions", Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP 2002), 2002.

[7] A. Cavalli, C. Gervy, S. Prokopenko, "New Approaches for Passive Testing Using an Extended Finite State Machine Specification", Information and Software Technology, Vol.45, 2003, pp.837-852

[8] Behrouz Tork Ladani, Baptiste Alcalde, Ana R. Cavalli, "Passive Testing - A Constrained Invariant Checking Approach", TESTCOM 2005, Lecture Note in Computer Sciences, Vol.3502, pp.9-22, 2005.

[9] B. Alcalde, A. Cavalli, D. Chen, D. Khnu, D. Lee, "Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach", FORTE 2004, Lecture Note in Computer Sciences, Vol.3235, pp.150-166, 2004.

[10] IETF Network Working Group, "OSPF version 2", RFC 2328, Internet Society, 1998.

[11] J. Dick, A. Faive, "Automating the generation and sequencing of test cases from model-based specifications", Proc. of the first International Symposium on Formal Methods in Europe (FME'93), Odense, Denmark, April 19-23, 1993, pp.268-284.

[12] Zvi Kohavi, "Switching and Finite Automata Theory", 2nd Edition, McGraw-Hill, Inc., 1978.

[13] Libpcap project, libpcap library, Ver. 0.8.1, http://www.tcpdump.org, 2003.

[14] Ethereal project, tethereal, Ver.0.9.14, http://www.ethereal.com, 2004.

[15] IP Infusion Inc., GNU Zerba, Ver.0.94, http://www.zebra.org, 2004.

[16] Luca Deri, "Improving Passive packet Capture: Beyond Device Polling", http://citeseer.ist.psu.edu/ 695645.html, 2004.

[17] Information Sciences Inst., Univ. of Southern California, Transmission Control Protocol, RFC 793, 1981.

Tae-Hyong Kim received the B.S. and M.S. degrees, from Yonsei University in 1992 and 1995, respectively, and a Ph.D. degree in electrical and electronic engineering from the same university in 2001. He was a postdoctoral fellow at the School of Information Technology and Engineering (SITE) at the University of Ottawa from 2001 to 2002. He is currently an assistant professor in the School of Computer and Software Engineering (SCSE) at the Kumoh National Institute of Technology (KIT) in Korea. His current research interests include software and protocol specification, verification and testing techniques, communication protocols, and next generation mobile networks. He is a member of the SDL Forum Society.