Network Protocol System Passive Testing for Fault Management - a Backward Checking Approach

Baptiste Alcalde[1], Ana Cavalli[1], Dongluo Chen[2], Davy Khuu[1], and David Lee[3]

[1] Institut National des Télécommunications GET-INT, Evry, France
{baptiste.alcalde, ana.cavalli, davy.khuu}@int-evry.fr
[2] Department of Computer Science, Tsinghua University, Beijing, China
chdl@csnet1.cs.tsinghua.edu.cn
[3] Bell Labs Research, Lucent Technologies lee@research.bell-labs.com

Abstract. Passive testing has proved to be a powerful technique for protocol system fault detection by observing its input/output behaviors yet without interrupting its normal operations. To improve the fault detection capabilities we propose a backward checking method that analyzes in a backward fashion the input/output trace from passive testing and its past. It effectively checks both the control and data portion of a protocol system, compliments the forward checking approaches, and detects more errors. We present our algorithm, study its termination and complexity, and report experiment results on the protocol SCP.

1 Introduction

Passive testing is an activity of detecting faults in a system under test by observing its input/output behaviors without interfering its normal operations. The usual approach of passive testing consists in recording the trace produced by the implementation under test and trying to find a fault by comparing this trace with the specification ([4], [6], [7]). Other approaches explore relevant properties required for a correct implementation, and then check on the implementation traces of the systems under test ([1], [2]). Most of the work on passive testing are based on finite state machines (FSMs) and they are focused on the control part of the tested systems without taking into account data parts. To cope with protocol data portions, Extended Finite State Machines (EFSMs) are used to model the systems, which include parameters and variables to encode data. In [7] a first approach to perform passive testing on EFSMs was proposed. An algorithm based on constraints on variables was developed and applied to GSM-MAP protocol. However, this algorithm cannot detect transfer errors. In [5], an algorithm based on variable determination with the constraints on variables was presented. This algorithm allows to trace the variables values as well as the system state, however, every transfer errors still cannot be detected.
To overcome this limitation, we propose a new approach based on backward tracing. This algorithm is strongly inspired by this presented in [5], but processes

the trace backward in order to further narrow down the possible configurations for the beginning of the trace and to continue the exploration in the past of the trace with the help of the specification. This algorithm contains two phases. It first follows a given trace backward, from the current configuration to a set of starting ones, according to the specification. The goal is to find the possible starting configurations of the trace, which leads to the current configuration. Then it analyses the past of this set of starting configurations, also in a backward manner, seeking for end configurations, that is to say configurations in which the variables are determined. When such configurations are reached, we can take a decision on the validity of the studied path.
This new algorithm is applied to Simple Connection Protocol (SCP) that allows to connect two entities after a negotiation of the quality of service required for the connection. Even it is a simple protocol it presents a number of key characteristics of real communication protocols. The testing results are compared to the passive testing algorithm in [5].
The rest of the paper is organized as follows. Section 2 describes the basic concepts used in the paper. Section 3 contains preliminary algorithms for processing transition back tracing. The section 4 presents the main backward tracing algorithm. In section 5 the issues related to the termination and complexity of the main algorithms are discussed. Section 6 reports the experiments of the algorithm on the Simple Connection Protocol.

2 Preliminaries

We first introduce basic concepts needed and then present an overview of our algorithm.

2.1 Extended Finite State Machine

We use Extended Finite State Machine (EFSM) to model the protocol systems.

Definition 1. *An Extended Finite State Machine M is a 6-tuple $M = < S, s_0, I, O, \vec{x}, T >$ where S is a finite set of states, s_0 is the initial state, I is a finite set of input symbols (eventually with parameters), O is a finite set of output symbols (eventually with parameters), $\vec{x}$ is a vector denoting a finite set of variables, and T is a finite set of transitions. A transition t is a 6-tuple $t = < s_i, s_f, i, o, P, A >$ where s_i and s_f are the initial and final state of the transition, i and o are the input and the output, P is the predicate (a boolean expression), and A is the ordered set (sequence) of actions.*

Definition 2. *An events trace is a sequence of I/O pairs.*

In this paper we consider that the traces can start at any moment of the implementation execution.

Given a trace from the implementation under test and a specification, the algorithm will detect the three types of error that can occur in an EFSM.

Definition 3. *The three types of error are:*

1. ***the output errors*** *: when the output of a transition in the implementation differs from the output of the corresponding transition in the specification.*
2. ***the transfer errors*** *: when the ending state of a transition in the implementation differs from the ending state of the corresponding transition in the specification.*
3. ***the mixed errors*** *: a mix between the two errors defined above.*

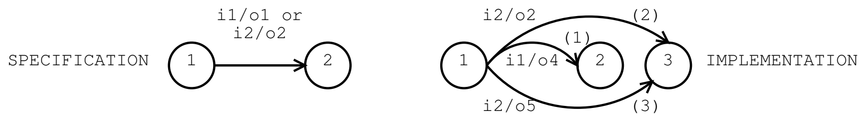


Fig. 1. Output(1), transfer(2), and mixed(3) errors

2.2 Candidate Configuration Set

The backward checking algorithm processes in two phases as shown in figure 2. The first step consists in following the trace w backward, from the end to the beginning, according to the specification. The goal is to arrive to the set X of possible start configurations of w. In order to keep information we use configurations named Candidate Configuration Set (CCS) inspired from [5].

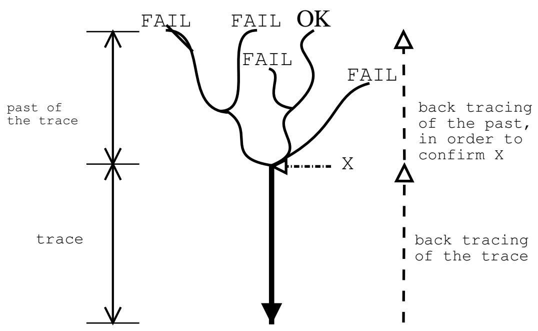


Fig. 2. Overview of Backward Checking

Definition 4. *Let M be an EFSM. A Candidate Configuration Set (CCS), is a 3-tuple $(e, R, Assert)$ where e is a state of M, R is an environment (that is to say that each variable v has a constraint $R(v)$), and $Assert$ is an assertion (Boolean expression).*

The second step is the backward checking of the trace past. This step consists in confirming at least one of the departure configurations extracted from the back tracing of a trace. It means we must verify that the variable values or constraints are compliant with the specification. We need to trace the transitions from their end states to their start states until we reach a validation criteria. We need to confirm the variables ranges. However, often there is only a subset of variables that we can confirm, and we call these variables determinant of a trace.

Definition 5. *A variable v is a determinant of a trace t if v must be necessarily validated before validating t.*

In order to keep information about determinants, we define a new structure for the past of the trace: the Extended Candidate Configuration Set (also called Extended Configuration).

Definition 6. *Let M be an EFSM. An Extended Candidate Configuration Set (ECCS) is a 4-tuple $(e, R, Assert, D)$, where e is a state of M, R is an environment, $Assert$ is an assertion, and D is a set of determinant variables.*

Between the two steps we check the determinant variable set as follows : every variable whose interval in a configuration of X - the set of possible start configurations of the trace that is included in its specified domain - must be added to the determinant variables set to be checked.

3 Preliminary Algorithms

In the following section, we present the preliminary algorithms that will be used in the main algorithm. We begin with the inverse actions algorithm, and then consistency checking and the transition back tracing algorithms.
What we do is checking backward the trace and then exploring its past as shown in Fig 2, determining the variables. In order to perform this checking on the whole trace and its past we need a process that checks a transition backward. The algorithms presented in this section make it possible.

3.1 The Inversed Actions

A main difficulty is the application of the inverse action A^{-1}. The inverse actions will be processed in a reversed order. Hence the following normal ordered actions $\{a_1, \ldots, a_n\}$ will be processed in an order: $\{a_n, \ldots, a_1\}$.

Each inverse action depends on the type of the corresponding normal action. There are three types of actions:

1. $w \longleftarrow constant$
2. $w \longleftarrow f(u, v, \ldots)$ where w is not a variable of f
3. $w \longleftarrow f(w, u, v, \ldots)$ where w is also a variable of f

These three types of actions are assignations : they overwrite the value of the left variable w with the value of the right component. We note that the value of w is modified by an action, but the other variables after action keep the value they had before the action and that only the value of the variable w will be modified by back tracing a transition. Except for this, every type of action must be inverted:

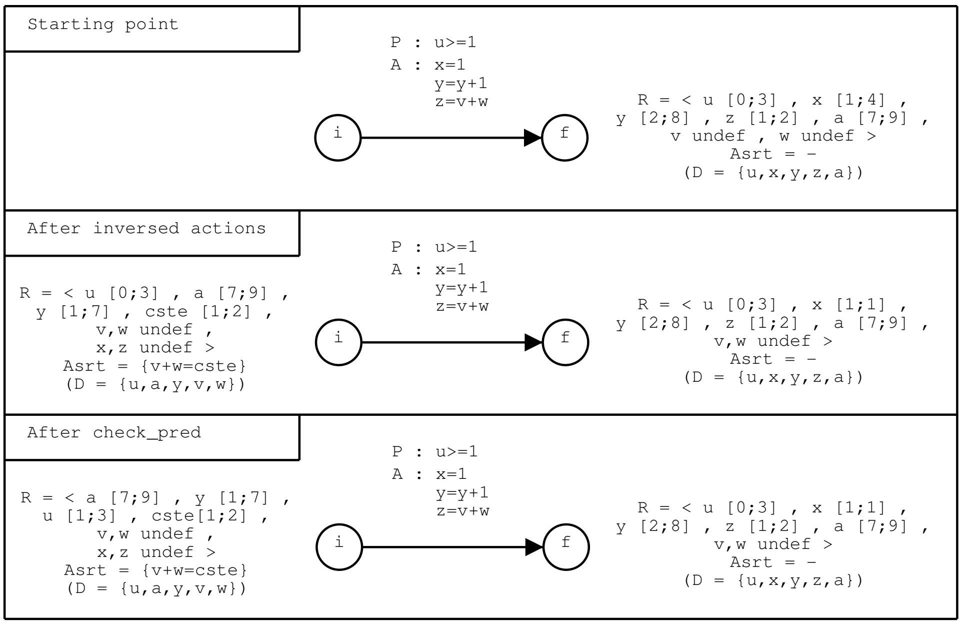
1. Action of type 1. The value of w after the action is a *constant*. This gives us a first occasion of detecting an error. If the *constant* does not conform the current constraint then we are on an invalid path. Otherwise, we replace every occurrence of w with the *constant* and refine the constraints of other variables. However, it is impossible to know the value of w before the action; indeed, actions simply overwrite the former value of w. After the action back tracing the value of w is UNDEFINED;
2. Action of type 2. We could take that $R(w)$ is equal to $R(f(u, v, \ldots))$ but we can be more precise: it is $R(w) \cap R(f(u, v, \ldots))$. In order to keep as much information as possible, every occurrence of w will be replaced by $f(u, v, \ldots)$. However, the value of w before action remains UNDEFINED;
3. Action type 3. This action brings no new constraint refinement on the variable w (on the left side of the assignment) after the action (left member) but it gives a constraint on the variable w (on the right side of the assignment) before the action. Consequently, every occurrence of w will be replaced by $f^{-1}(w)$.

3.2 Final Checking Phase

The check_consistency process is from [5] and is able to detect inconsistency in the definition of the variables by refining the intervals of variables and its constraints.
There are no big differences between the transition back tracing algorithms for the trace and for its past, and we ignore in the trace algorithm what can happen to the set of determinants before the action. Indeed, in the trace we do not determine variables; we can only refine their values, and we invalid the trace if the constraints are not consistent. For the trace we must check the output before processing the inverse actions. After processing every action we can determine the variables involved in the input if its constraint is consistent with what we found. Otherwise, we invalid the transition.
On the other hand, we must check that the variable values that we found are consistent with the predicates. Otherwise, the path is invalid. Therefore, in the checking we must determine if a transition is valid or not. We need a process called check_pred for the past of the trace to modify the set of determinants. In the case of back tracing, we just need to add the predicates to the set of assertions and process check_consistency - no specific operations are needed.
The pseudo code for back tracing of the trace and of its past, followed by the check_pred and check_consistency algorithms are presented in the appendix.

3.3 Example

We show now an example of this process. Consider the common steps of the trace and past cases, a transition without input/output, and we include the variable set D into parentheses.



4 Main Algorithms

We are ready to present our main algorithm of backward checking.

4.1 Backward Checking of a Trace

The backward checking for a whole trace can be derived from the algorithm for back tracing a transition (Back_trace_transition):

- *trace*: The observed trace. *gettail*(*trace*) removes and returns the last i/o pair from *trace*.
- X: Set of starting extended configurations from back trace of an event trace. Each configuration is a 4-tuple $(e, R, Assert, D)$
- X': Current set of extended configurations
- V: Set of known extended configurations
- c': A new configuration
- : Returns TRUE if the sequence is correct, and FALSE otherwise

```
1.  $V \longleftarrow X$
2.  while $X \neq \emptyset$ & $i/o := gettail(trace)$ do
3.      $X' \longleftarrow \emptyset$
4.      for each configuration $c \in X$ do
5.          for each transition $t$ where $t.end\_state = c.state$ and $t.i/o = i/o$ do
6.              $c' \longleftarrow$ Back_trace_transition$(t, c)$
7.              $X' \longleftarrow X' \cup \{c'\}$
8.              $V \longleftarrow V \cup X'$
9.      $X \longleftarrow X'$
10. return FALSE
```

4.2 Backward Checking of the Past of an Event Trace

The backward checking algorithm applied to the past of a trace consists of a breadth-first search in the past of the configurations, which are extracted from the back tracing of a trace due to the fact that one cannot use a variable value before it is assigned. In order to validate a trace, we only need to find a path binding a set of assignments or predicates to one of the configurations extracted from back tracing. We now proceed to the main algorithm. We first define the operations $\sqcap$ and $\setminus$ on the Extended Candidate Configuration Sets (ECCS) that will be used for pruning the exploration tree of the past. Then we study the path convergence and discuss the algorithm termination, the correctness and the complexity.

The $\sqcap$ Operation. It is an intersection between two configurations:

Definition 7. *Let be three configurations $c_1 = (e, R_1, Assert_1, D)$, $c_2 = (e, R_2, Assert_2, D)$, and $c = (e, R, Assert, D)$. We define the intersection operator $\sqcap$ as follows. If $c = c_1 \sqcap c_2$, then :*

1. *for each variable v, $R(v) = R_1(v) \cap R_2(v)$ where $\cap$ is the intervals intersection operator*
2. *$Assert = Assert_1 \wedge Assert_2$ where $\wedge$ is the boolean "and" operator*

Remark on $\sqcap$. The configuration states and the variable sets, which are not validated yet, are the same. If they are not, the "intersection" equals to NULL.

The $\setminus$ Operation. It is a privation. Given two configurations c_1 and c_2, the result of $c_1 \setminus c_2$ is a couple (c_a, c_b). We obtain c_a by removing c_2 from c_1, but only in case of each variable is restricted to the intersection of the intervals c_1 and c_2, respectively. c_b is the rest of c_1.

Definition 8. *Given four configurations $c_1 = (e, R_1, Assert_1, D)$, $c_2 = (e, R_2, Assert_2, D)$, $c_a = (e, R_a, Assert_a, D)$ and $c_b = (e, R_b, Assert_b, D)$, we define the privation operator $\setminus$ as follows. If $(c_a, c_b) = c_1 \setminus c_2$, then :*

1. *for c_a :*
   (a) *for each variable v, we have got : $R_a(v) = R_1(v) \cap R_2(v)$ where $\cap$ is the intervals intersection operator*

   (b) *$Assert_a = Assert_1 \wedge \overline{Assert_2}$, where $\wedge$ is the boolean "and" operator*
2. *for c_b :*
   (a) *$R_b = R_1$*
   (b) *$Assert_b = Assert_1 \wedge (\bigvee_{i=0}^{|V|-1} (v_i \notin R_2(v_i)))$ where $\wedge$ is the boolean "and" operator, and $\vee$ is the boolean "or" operator (be careful of priorities of parenthesis)*

Remark on $\backslash$. If $Assert_2$ equals to $\emptyset$, then c_a equals to NULL. Indeed $Assert_2$ means we have to keep all of the values that R_2 allows, yet on the contrary $\overline{Assert_2}$ implies that we must delete all of them.

General remark. The operations $\sqcap$ and $\backslash$ may return configurations, which are inconsistent. For example, the result of $c_1 \backslash c_1$ is not consistent. Moreover, some results may need to be refined. Indeed when two assertions are concatenated the constraints intervals of each variable may have to be changed. So we should use the **Check_consistency** procedure that has already been presented. For now, we consider that the results of $\sqcap$ and $\backslash$ are automatically checked and transformed by **Check_consistency**.

Examples. Consider the configurations $c_1 = (e, < x = [0;5], y = [0;3] >, \_, \{x\})$ (where _ means no assertion) and $c_2 = (e, < x = [0;2], y = [-1;1] >, \{x > y\}, \{x\})$, and three configurations c_i, c_a and c_b, which are defined as following :

– $c_i = c_1 \sqcap c_2$
– $(c_a, c_b) = c_1 \backslash c_2$

We first determinate c_i. R_i is defined as the intersection of R_1 and R_2, and $Assert_i$ is $Assert_1 \wedge Assert_2$. Then we have:
$c_i = (e, < x = [0;2], y = [0;1], \{x > y\}, \{x\})$.

Determinating c_a and c_b is a little bit more complicated. R_a is the intersection of R_1 and R_2, and $Assert_a$ is $Assert_1 \wedge \overline{Assert_2}$. Then we have :
$c_a = (e, < x = [0;2], y = [0;1] >, \{x \leq y\}, \{x\})$.

At last for c_b, we have the following properties. R_b equals R_1, and we must add $x < 0 \vee x > 2$ and $y < 0 \vee y > 1$ to $Assert_b$. Then we have :
$c_b = (e, < x = [0;5], y = [0;3] >, \{(x < 0 \vee x > 2) \wedge (y < 0 \vee y > 1)\}, \{x\})$.

Note that the two last configurations c_a and c_b are not refined as it was defined in [5]. If we apply the **Check_consistency** procedure, we obtain :
$c_a = (e, < x = [0;1], y = [0;1] >, \{x \leq y\}, \{x\})$ and $c_b = (e, < x = [3;5], y = [2;3] >, \_, \{x\})$.

Path Convergence. Consider a step r of our algorithm. If we find a configuration c that we have already found earlier, in a previous step or earlier in the step r, we have got a "path convergence" phenomena.

Definition 9. *Two paths P_1 and P_2 are convergent (in the past!) if they lead to the same configuration c.*

Consequently both P_1 and P_2 have the same past. So we will obtain the same information if we explore the common past from P_1 or from P_2. Consider that we have first followed P_1. When we find that P_2 converges toward c, we do not continue the exploration: we prune P_2 at c. The pruning enables us to deal with the infinite exploration paths.
Unfortunately extended configurations make convergences hard to be detected; they are non-empty intersections of configurations. We proceed as follows. Given three configurations c, c_1 and c_2, let c be equal to $c_1 \sqcap c_2$. Suppose that c_2 has been found before c_1. Then we have the following:

– c =NULL. c_1 and c_2 are independent and the respective pasts of c_1 and c_2 must be explored;
– ($c \neq$NULL) $\wedge$ ($c = c_1$). c_1 is included in c_2 and we must delete c_1;
– ($c \neq$NULL) $\wedge$ ($c \neq c_1$). c_2 is included in c_1 and we must substitute c_1 by $c_1 \backslash c_2$

The algorithm **Check_redundancy**, that will be described later, deals with the convergence cases.

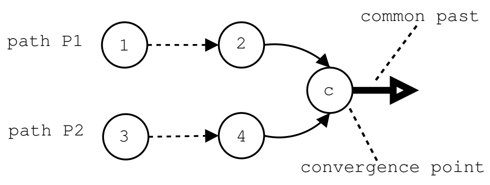


Fig. 3. Example of Path Convergence

Algorithm of Backward Checking of the Past of a Trace. The Backward_checking_past algorithm backward traces the past of a trace in order to validate it. The input is the set of starting extended configurations, which we extracted from the trace back tracing.

Note that if the start configuration is invalid (not reachable from the initial configuration set) then we have to explore backward all the configurations to tell whether there is no valid path from the initial configuration set. However, if it is indeed valid, finding a valid path is enough. In most cases of passive testing, the traces do not contain faults and it is desirable to use a heuristic method to find a valid path. We now present such a procedure.

In order to guide the heuristic search, we have figure out the end configurations. A configuration set c is an end configuration set if it satisfies one of the following conditions:

1. $c \cap c\_init \neq \emptyset$ where c_init is the initial configuration set of the machine
2. $c.D = \emptyset$
3. c is contained in another configuration set that has been explored

The second criteria is valid, since $c.state$ is reachable from the initial state of the machine, and there must be a valid path from the initial configuration.

We now present a heuristic search. We assign a weight for each configuration-transition pair $< c, t >$. Since we want to trace back to the initial configuration or reduce $c.D$, we increase the weight of such pairs. A priority queue Q contains all the configuration-transition pairs to be explored, sorted by their weights. The pair with the highest weight is placed in the head of Q and will be selected first.

The weight wgt of a configuration-transition pair $< c, t >$ with an initial value 0 can be incremented by the following rules :

1. if $t.start\_state = c\_init.state$, $wgt += w1$
2. if $t.start\_state$ has not been explored, $wgt += w2$
3. if $t.action$ defines k variables in $c.D$, $wgt += w3 * k$

The first two rules guide the search towards the initial state of the specification while the third one is to reduce the set of determinants. It is important to remark that we don't need to reach the initial state itself, and that a transition determining every variables left in the set of determinants is enough to conclude on the correctness of the explored path. This explains the importance of the third rule (we can note that the initial state is a particular case of it as it is supposed to determine every variables).

The values of $w1, w2$, and $w3$ can be given after practical usage.

The following is the procedure where

- Q: Set of configuration-transition pairs to be explored
- V: Set of already-explored Extended Configurations
- : Returns TRUE if the trace is correct, and FALSE otherwise.

```
1.  initialize $Q, V$
2.  **while** $Q \neq \emptyset$ **do**
3.      .take the first item $< c, t >$ from $Q$
4.      .build a new configuration $c'$ :
        $c' \longleftarrow$ **Back_past_transition**$(t, c)$
5.      .**if** $c' == NULL$
6.      .   .**goto** 2
7.      .**if** $c'.D = \emptyset$ **do**
8.      .   .**return** TRUE
9.      .$c'$ =**Check_redundancy**$(c', V)$
10.     .**If** $c' \neq \emptyset$ **do**
11.     .   .$V \longleftarrow V \cup c'$
12.     .   .**for** each transition $t$ where
            $t.end\_state = c'.state$ **do**
13.     .   .   .calculate the weight of
                $< c', t >$
14.     .   .   .insert $< c', t >$ into $Q$ by its
                weight
15. **return** FALSE
```

In the worst case, this algorithm will explore backward all the possible configurations. When Q becomes empty no valid path is possible from the trace information from the passive testing and "FALSE" is returned - there are faults in the protocol system under test.

5 Algorithm Termination and Complexity

In the first part of the algorithm (backtracking of the trace) there is no problem of termination because we follow the trace, so this step finishes when the trace finishes. The problem we had and we solved is in the second part of the algorithm (in the past of the trace). We present these problems in the following subsection.

5.1 Loop Termination

There are two problems that we must solve : infinite paths, and infinite number of paths. These problems are often caused by loops.
A first infinite path case occurs when a path infinitely often reachs a configuration. This problem is solved thanks to the detection of path convergence (see 4.2), and ECCS operations that prevents exploring more than once in a configuration. A second case occurs when a variable is infinitely increased or decreased. In this case the loop is limited by the upper or lower bound of the interval of definition of the variable.
There are two cases when we have an infinite number of paths. First, a configuration has an infinite number of parents. Secondly there is an infinite path from which several paths start. But if the configurations number is finite, then a configuration can not have an infinite number of parents.
We proved the termination of the algorithm, and we present in the next subsection a study of the algorithm complexity.

5.2 Complexity

In the first part of the algorithm (trace) the complexity depends on the trace. We have :
Proposition 1. Suppose that the observed event trace is of length l, then the complexity of the first part of the presented algorithm is proportional to l.

For the second part (past of the trace) the complexity depends on the number of possible configurations. A configuration includes a state number, interval of definition of variables, and a list of determinant variables. The complexity of the second part of the algorithm is :
Proposition 2. Let n_s be the number of states in the EFSM of the specification, $|R(x_i)|$ the number of values the variable x_i can take (in the interval of definition), and n the number of variables, then there is in $\mathrm{O}(n_s(\prod_i |R(x_i)|)(2^n - 1))$ possible configurations.

We must balance this complexity with the power of the algorithm. The worst case of this algorithm is the case where there is an error because we must check every path of the past. When there is no error our algorithm gives a sure answer (in constrast with former algorithms) at the first correct path we meet (that is supposed to be fast using the heuristic). Anyway, the backward checking - if we consider only the trace analysis - is an improvement of former algorithm, and has the same complexity.

6 Experiments on SCP Protocol

We now report the application of our algorithms on the Simple Connection Protocol (SCP). SCP is a very interesting protocol for test purpose because it includes most possible difficulties for passive testing in a small specification. Therefore, it can show the efficiency of the algorithm on bigger real protocols. We first describe this protocol and then report the experiments of the algorithm from [5] and our new algorithm.

6.1 The Simple Connection Protocol

SCP allows us to connect an entity called *upper layer* to an entity called *lower layer* (Fig 4). The upper layer performs a dialogue with SCP to fix the quality of service desirable for the future connection. Once this negotiation finished, SCP dialogues with the lower layer to ask for the establishment of a connection satisfying the quality of service previously negotiated. The lower layer accepts or refuses this connection request. If it accepts the connection, SCP informs the upper layer that connection was established and the upper layer can start to transit data towards the lower layer via SCP. Once the transmission of the data finished, the upper layer sends a message to close the connection. On the other hand, if the lower layer refuses the connection, the system allows SCP to make three requests before informing the upper layer that the connection attempts all failed. If the upper layer wishes again to be connected to the lower layer, it is necessary to restart the QoS negotiation with SCP from beginning. Every variable is defined in the interval $[0;3]$. An EFSM specification of SCP is in the figure 5.

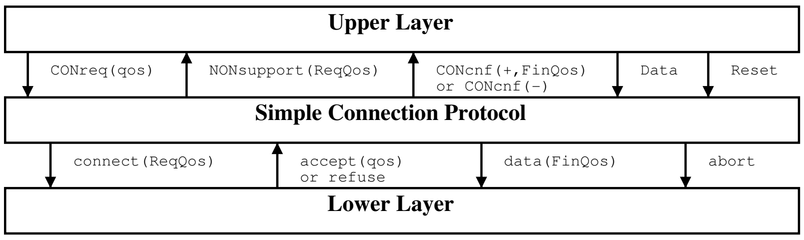


Fig. 4. Simple Connection Protocol : Layers

6.2 Experiments of the Two Algorithms

Consider a false implementation of SCP, that has been used in [2] : the predicate of the transition $\textcircled{s3}\longrightarrow\textcircled{s1}$ is replaced by $TryCount = 0$. The figures 6, 7 and 8 show the executions of the first algorithm and of the backward

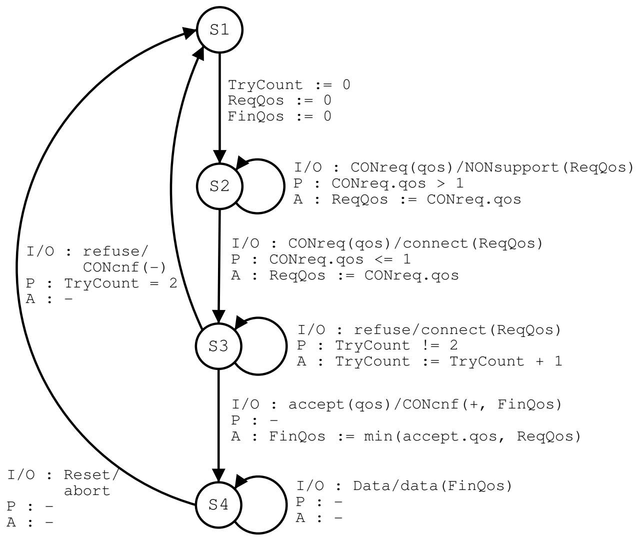


Fig. 5. Simple Connection Protocol : EFMS specification

checking algorithm (trace and past) on the trace **CONreq(1)/ connect(1), refuse/CONcnf(-)**.
The figure 6 shows that the error is not detected by the algorithm presented in [5]. The trace is left ”possible” for it.
The figure 8 shows the execution of backward checking algorithm in the past. We obtain the configuration $(S2, < TryCount = 2; ReqQos = 1; FinQos = [0;3]; CONreq.qos = 1 >, \_, \{TryCount; ReqQos; CONreq.qos\})$ from the back tracing of the trace (Fig. 7) and we continue in the past. After the first step of the while loop, X is empty because the transition $\textcircled{s2} \longrightarrow \textcircled{s2}$ leads to a contradiction between $CONreq.qos$ value (=1) and the predicate $CONreq.qos > 1$, and the transition $\textcircled{s1} \longrightarrow \textcircled{s2}$ is also invalid due to a contradiction between $ReqQos$ value (=1) and the action $ReqQos = 0$. Then there is no more configuration to backtrack and the algorithm terminates, returning FALSE - there are faults in the protocol implementation.

7 Conclusion

Apparently, passive testing is a promising method for protocol fault management, as it allows to test without disturbing the normal operation of a protocol system or service. In this paper, we present a new backward checking algorithm. It detects output and transfer errors in an implementation by observing and analyzing its event traces. A major difficulty of passive testing is its analysis for faults. Our approach provides a backward trace analysis that is efficient and also a compliment to the forward analysis in [5], and can uncover more faults.

step	event	configurations
0	-	$(S_i;\ < TC = [0;3],\ RQ = [0;3],\ FQ = [0;3],\ Crq.qos = [0;3],\ acc.qos = [0;3] >,\ \_)$ (for each i state number)
1	CONreq(1) / connect(1)	$(S_3;\ < TC = [0;3],\ RQ = 1,\ FQ = [0;3],\ Crq.qos = 1,\ acc.qos = [0;3] >,\ \_)$
2	refuse / CONcnf(-)	$(S_1;\ < TC = 2,\ RQ = 1,\ FQ = [0;3],\ Crq.qos = 1,\ acc.qos = [0;3] >,\ \_)$

Fig. 6. Execution of the First Algorithm

step	event	configurations
0	-	$(S_i;\ < TC = [0;3],\ RQ = [0;3],\ FQ = [0;3],\ Crq.qos = [0;3],\ acc.qos = [0;3] >,\ \_)$ (for each i state number)
1	refuse / CONcnf(-)	$(S_3;\ < TC = 2,\ RQ = [0;3],\ FQ = [0;3],\ Crq.qos = [0;3],\ acc.qos = [0;3] >,\ \_)$
2	CONreq(1) / connect(1)	$(S_2;\ < TC = 2,\ RQ = 1,\ FQ = [0;3],\ Crq.qos = 1,\ acc.qos = [0;3] >,\ \_)$

Fig. 7. Back Tracing the Trace

step				
0	current conf.	$(S_2,\ < TC = 2;\ RQ = 1;\ FQ = [0;3];\ Crq.qos = 1 >,\ \_\ ,\ \{TC;RQ;Crq.qos\})$		
	seen conf.	$(S_2,\ < TC = 2;\ RQ = 1;\ FQ = [0;3];\ Crq.qos = 1 >,\ \_\ ,\ \{TC;RQ;Crq.qos\})$		
	transition $\textcircled{s_2}\longleftarrow\textcircled{s_2}$	back tracing	next config.	$\emptyset$
	transition $\textcircled{s_2}\longleftarrow\textcircled{s_1}$	back tracing	next config.	$\emptyset$
	validation	there is no more configuration : return FALSE		

Fig. 8. Back Tracing the Past of the Trace

Passive testing is a formal approach for network protocol system monitoring and measurement where Internet protocols such as OSPF and BGP were monitored for fault detection [3]. Formal method will continue to exhibit its power in network protocol system fault management in a wider range of applications and protocol layers.

8 Appendix

Back_trace_transition(t,c) Algorithm
This algorithm is used for backtracing a transition during the trace processing.

- : returns c' if $c' \xrightarrow{t} c$ is possible, NULL if not.

```
1.  if ($output.v \notin c.R(v)$) do
2.    .return NULL
3.  else
4.    .$c' = clone(c)$
5.    .$c'.R(v) = Def(v)$
6.    .replace every occurrence of $v$ in $c'.Asrt$ by
           $output.v$
7.  inverse list of actions
8.  foreach action $a$ do
9.    .if action $a$ is : $w \longleftarrow constante$ then
10.   .  .if $c'.R(w) \cap constante = \emptyset$ then
11.   .  .  .return incorrect trace
12.   .  .else
13.   .  .  .$c'.R(w) = Def(w)$
14.   .  .  .replace every occurrence of $w$ in $c'.Asrt$ by $constante$
15.   .if action $a$ is : $w \longleftarrow f(\vec{x})$ then
16.   .  .replace every occurrence of $w$ by
           $f(\vec{x})$ in $c'.Asrt$
17.   .  .if $w \in \vec{x}$ then
18.   .  .  .$c'.R(w) = R(f^{-1}(\vec{x}))$
19.   .  .else
20.   .  .  .$c'.Asrt = c'.Asrt \wedge$
           $(\underline{w} \leq f(\vec{x}) \leq \overline{w})$
21.   .  .  .$c'.R(w) = Def(w)$
22. foreach predicate $p$ do
23.   .normalize $p$
24.   .$c'.Asrt = c'.Asrt \wedge p$
25. if ($input.v \notin c'.R(v)$) do
26.   .return NULL
27. else
28.   .$c'.R(v) = Def(v)$
29.   .replace every occurrence of $v$ by
         $input.v$ in $c'.Asrt$
30. check_consistency($c'$)
31. return $c'$
```

Back_past_transition(t,c) Algorithm
This algorithm is used for backtracing a transition during the past trace processing.

```
1.  $c' = clone(c)$
2.  inverse list of actions
3.  foreach action $a$ do
4.    .if action $a$ is : $w \longleftarrow constante$ then
5.    .  .if $c'.R(w) \cap constante = \emptyset$ then
6.    .  .  .return incorrect trace
7.    .  .else
8.    .  .  .$c'.R(w) = Def(w)$
9.    .  .  .replace every occurrence of $w$ in $c'.Asrt$ by $constante$
10.   .  .  .$D = D - w$ //w is validated
11.   .if action $a$ is : $w \longleftarrow f(\vec{x})$ then
12.   .  .replace every occurrence of $w$ by
           $f(\vec{x})$ in $c'.Asrt$
13.   .  .if $w \in \vec{x}$ then
14.   .  .  .$c'.R(w) = R(f^{-1}(\vec{x}))$
15.   .  .else
16.   .  .  .$D = D - w$
17.   .  .  .$c'.Asrt = c'.Asrt \wedge$
           $(\underline{w} - cst \leq f(\vec{x}) \leq \overline{w} - cst)$
18.   .  .  .$c'.R(w) = Def(w)$
19.   .  .$D = D \cup \vec{y}$
20. check_consistency(c')
21. check_pred(p,c')
22. return $c'$
```

Check_pred(P, c) Algorithm

```
1.  for each predicate $v = f(\vec{x}) \in P$ do
2.    .if $(c.R(v) \cap c.R(f(\vec{x})) \subseteq c.R(v))$ then
3.    .  .$c.D = c.D - v$ // $v$ is validated
4.    .  .$c.R(v) = c.R(v) \cap c.R(f(\vec{x}))$
5.    .else return FALSE
6.  return TRUE
```

Check_consistency(c) Algorithm
The following algorithm derives from the one presented in [5]. It tests configurations consistency, refines their constraints and delete all unused assertions. It returns the processed configuration if the initial one is consistent, or NULL if it is not.
Variable assignment Rule **(R)**: for each variable range if we have a set of non empty intervals from the processing of the conjunctive terms then the new variable range consists of an interval whose lower (upper) bound is the minimum (maximum) of all the interval lower (upper) bounds.

- c : configuration that must be refined
- c' : copy of c. Note $c' = (e, R, Assert, D)$
- return the refinment of c, or NULL
- S : a new set of intervals
- At : a new assertion

```
1.  $c' \longleftarrow c$
2.  transform $c'.Assert$ in DNF
3.  $S \longleftarrow \emptyset$
4.  $At \longleftarrow \emptyset$
5.  for each conjunctive term $D_t$ of $c'.Assert$ do
6.    $dt\_true \longleftarrow$ TRUE
7.    $refine \longleftarrow$ TRUE
8.    while $refine$ = TRUE do
9.      $refine \longleftarrow$ FALSE
10.     $R_l \longleftarrow c'.R$
11.     $R'_l \longleftarrow \emptyset$
12.     for each predicate $p$ of $D_t$ do
13.       normalize $p$
14.       if $\sum_i (a_i R_l(x_i)) \subseteq R(\sim Z)$ do
                  /*$p$ is TRUE*/
15.         remove $p$ from $D_t$
16.         go to 12
17.       if $\sum_i (a_i R_l(x_i)) \cap R(\sim Z) = \emptyset$ then
18.         $dt\_true \longleftarrow$ FALSE
19.         go to 28
20.       for each $x_j$, $j = 1, \ldots, k$ do
21.         $R'_l(x_j) \longleftarrow \frac{R(\sim Z) - \sum_{i \neq j}(a_i R(x_i))}{a_j} \cap R_l(x_j)$
22.         if $R'_l(x_j)$ = NULL do
23.           $dt\_true \longleftarrow$ FALSE
24.           go to 35
25.         if $R'_l(x_j) \subset R_l(x_j)$ do
26.           $refine \longleftarrow$ TRUE
27.           $R_l(x_j) \longleftarrow R'_l(x_j)$
28.   if $dt\_true$ = FALSE then
29.     remove $D_t$ from $c'.Assert$
30.   else
31.     for each variable $v$ do
32.       $At \longleftarrow At \wedge (v \in R_l(v))$
33.       $S(v) \longleftarrow$ combination of $S(v)$ and
                $R_l(v)$,
          according to R
34. if $|S| = 0$ do
35.   return NULL
36. else
37.   $c'.R \longleftarrow S$
38.   $c'.Assert \longleftarrow c'.Assert \wedge At$
39.   return $c'$
```

Check_redundancy(c, V) Algorithm
The following algorithm aims to deal with convergence cases, in order to solve the infinite loops problem.

- c : configuration to be checked
- V : set of already-seen configurations

- X : set of configurations from redundancy check
- X' : intermediate set of configurations

```
1.  $X \longleftarrow \{c\}$
2.  **for** each configuration $c_V \in V$ **do**
3.    .$X' \longleftarrow \emptyset$
4.    .**for** each configuration $c_i \in X$ **do**
5.    .  .$c'_i \longleftarrow c_i \sqcap c_V$
6.    .  .**if** $c'_i$ =NULL **do**
7.    .  .$X' \longleftarrow X' \cup \{c_i\}$
8.    .  .  .**goto** 4
9.    .  .**else if** ($c'_i$ ≠NULL)&($c'_i = c_i$) **do**
10.   .  .  .**goto** 4
11.   .  .**else if** ($c'_i$ ≠NULL)&($c'_i \neq c_i$) **do**
12.   .  .  .$(c_i^a, c_i^b) \longleftarrow c_i \backslash c'_i$
13.   .  .  .**if** $c_i^a$ ≠NULL **do**
14.   .  .  .  .$X' \longleftarrow X' \cup \{c_i^a\}$
15.   .  .  .**if** $c_i^b$ ≠NULL **do**
16.   .  .  .  .$X' \longleftarrow X' \cup \{c_i^b\}$
17.   .$X \longleftarrow X'$
18. **return** $X$
```

References

1. J.A. Arnedo, A. Cavalli, M. Núñez, *Fast Testing of Critical Properties through Passive Testing*, Lecture Notes on Computer Science, vol. 2644/2003, pages 295-310, Springer, 2003.
2. A. Cavalli, C. Gervy, S. Prokopenko, *New approaches for passive testing using an Extended Finite State Machine specification*, in Information and Software Technology 45(12) (15 sept. 2003), pages 837-852, Elsevier.
3. R. Hao, D. Lee, and J. Ma, *Fault Management for Networks with Link-State Routing Protocols* Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS), April 2004.
4. D. Lee, A.N. Netravali, K. Sabnani, B. Sugla, A. John, *Passive testing and applications to network management*, IEEE International Conference on Network Protocols, ICNP'97, pages 113-122. IEEE Computer Society Press, 1997.
5. D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu and X. Yin, *A formal approach for passive testing of protocol data portions*, Proceedings of the IEEE International Conference on Network Protocols, ICNP'02, 2002.
6. R.E. Miller, and K.A. Arisha, *On fault location in networks by passive testing*, Technical Report #4044, Departement of Computer Science, University of Maryland, College Park, August 1999.
7. M. Tabourier and A. Cavalli, *Passive testing and application to the GSM-MAP protocol*, in Information and Software Technology 41(11) (15 sept. 1999), pages 813-821, Elsevier, 1999.