Passive Testing - A Constrained Invariant Checking Approach

Behrouz Tork Ladani[1], Baptiste Alcalde[2] and Ana Cavalli[2]

[1] Department of Computer Engineering, University of Isfahan, Isfahan, Iran
ladani@eng.ui.ac.ir
[2] Institute National des Telecommunications GET-INT, Evry, France
{ana.cavalli, baptiste.alcalde}@int-evry.fr

Abstract. Passive testing of a network protocol is the process of detecting faults in the protocol implementation by passively observing its input/output behaviors (execution trace) without interrupting the normal network operations. In observing the trace, we can focus on the most expected relevant properties of the protocol specification by defining some invariants on the specification and checking them on the trace. While intuitive extraction of the invariants from the protocol requirements with respect to the control portion of the protocol system is relatively simple, taking the data portion into account is difficult. In this paper we propose algorithms for checking the correctness of given invariants on the specification and extracting the required constraints on the variables (data portion). Once we generate the constraints for a given invariant, we can check if the execution trace is confirmed by the specification with respect to the invariant and its constraints. We show the applicability of the algorithm on a case study: the simple connection protocol (SCP).

Keywords: *passive testing, invariants, invariant checking, constraint solving, SCP*

1 Introduction

Testing network protocol implementations, to assure that they work as their specification, is of high importance. Instead of conventional active testing, there are propositions to use passive testing for network protocol systems which means observing the input/output behavior of the implementation (i.e. execution trace) without interfering its normal behavior ([1],[2]). The naive approach to passive testing is to record the execution trace and try to find its faults by comparing it with the specification ([3], [4], [5]). Other approaches try to extract the critical properties of the specification in the form of some invariants and then try to observe them on the implementation ([6], [7]).

Most of passive testing methods are focused on the control parts of the system under test without considering the data parts, so it is sufficient for them to use finite state machines (FSMs) as the specification method. To take the data part of the protocols into account, extended finite state machines (EFSMs) are used to specify the system. EFSM uses parameterized input/output signals including variable parameters to encode data as well as predicates and actions to control the firing of the transitions by manipulating the relevant data. There are some methods proposed to perform passive testing using EFSM ([5], [8], [9]). These methods are based on exploring constraints of the variables and comparing the whole specification with the implementation regarding the constraints in a backward or forward manner (i.e. the naive approach).

In this paper we represent a method to perform passive testing based on the invariants on the EFSM. In our approach, first we have to extract the invariants intuitively from the protocol specification requirements regarding only the control portion of the protocols. After that, to take the data portion into account, we consider the invariant parameters as some variables. We present two algorithms for finding the corresponding constraints over the variables of the invariants automatically. The algorithms use the unification method [10] for checking the correctness of the given invariants over the EFSM and finding the constraints over its variables. Having the invariants and their corresponding constraints in hand, we can check the execution trace with the invariants using pattern matching methods.

It should be noted that finding suitable control-driven invariants, especially with the help of an expert is relatively simple; also there are some methods to extract a limited set of invariants from an EFSM automatically [7]. We use the notion of invariant introduced in [11] with little changes in the definitions to extract the control driven invariants intuitively.

The rest of the paper is organized as follows: in section 2 some preliminary concepts needed in the rest of the paper are described. In section 3 the notion of forward and backward invariants are described. In section 4, our algorithms for checking invariants on a given EFSM and extracting corresponding constraints are presented. Section 5 reports some experiments of the algorithms on the Simple Connection Protocol (SCP) to show the applicability of the method in detecting subtle errors in some given traces of the protocol. In section 6 we conclude the paper.

2 Preliminaries

2.1 Extended Finite State Machine

We use Extended Finite State Machines (EFSMs) to specify the network protocols.

Definition 1. An *Extended Finite State Machine (EFSM)* M is a 6-tuple M=(S, I, O, x, T, s_0) where S is a finite set of states, I and O are the finite sets of input and output parametric symbols respectively, x is a vector denoting a finite set of variables, T is the finite set of transitions, and s_0 is the initial state.

Each transition $t \in T$ is a tuple (s, s′, i, o, P, A) where s, $s' \in S$ are the initial and final states of the transition respectively, $i \in I$, $o \in O$ are the input and output symbols (possibly with parameters) respectively, P is the predicate (a Boolean expression), and A is the sequence of actions.

Definition 2. Let M=(S, I, O, x, T, s_0) be an EFSM, the sequence $i_1/o_1, \dots i_n/o_n$ is a *path* in M if for any $1 \le j \le n$, $i_j \in I$, $o_j \in O$, and there exist n transitions $t_1 \dots t_n \in T$ and n+1 states $s,s_1, \dots, s_{n-1},s' \in S$ such that $t_1 = (s,s_1,i_1, o_1)$, $t_n = (s_{n-1},s',i_n, o_n)$ and for any 1<j<n we have $t_j = (s_{j-1},s_j,i_j, o_j)$.

2.2 Substitution and unification

Our algorithms use the unification method, so we borrow some definitions from the context of logic programming.

Definition 3. A *substitution* θ is a set of bindings, each of the form V/T, such that V is a distinct variable and T is a term. θ is called a *renaming* if it maps each variable to a new fresh variable.

Applying a binding V/T to an expression E, replaces each free occurrence of V in E by T. Applying a substitution θ on an expression E denoted by Eθ applies all the bindings in θ to E simultaneously and independently.

Definition 4. Let $\theta = \{V_1/T_1, \dots, V_m/T_m\}$ and $\alpha = \{U_1/S_1, \dots, U_n/S_n\}$ be substitutions. The *composition* of θ and α, denoted by θoα is defined as:

$$\theta o \alpha = \{V_1 / T_1 \alpha , \dots, V_m / T_m \alpha \} \cup \{U_k / S_k \mid U_k \notin \{V_1 , \dots , V_m\}\}$$

Definition 5. A *unifier* of two simple expressions E and F is a substitution θ such that Eθ = Fθ. If two simple expressions have a unifier, they are said to be *unifiable*; we also say that E is unified with F by the unifier θ. A *most general unifier*, abbreviated as *mgu*, of two simple expressions E and F is a unifier θ that is more general than any unifier of E and F. As an example two expressions P(X,f(X)) and P(b,f(a)) are not unifiable, while the most general unifier of the expressions P(X,f(a)) and P(b, f(Y)) is {X/b, Y/a}.

2.3 Normalizing action sequences

We need in our algorithms to track the changes in the variable values made by the action sequences in each transition, so we define a special normal action sequence and present an algorithm to normalize a given action sequence. In normalizing an action sequence of a transition, a special renaming substitution is produced which we name the *normalizer substitution* of that action. The normalizer substitution is then used to propagate the changes of the variable values in a transition to predicates and actions of the successive transitions.

Definition 6. Let x be the set of variables in an EFSM, also let $A=(l_1:=r_1, \dots l_n:=r_n)$ be an action sequence of size n in a transition in the EFSM, in which $l_i \in x$ and r_i is an expression for $1 \leq i \leq n$. Also suppose that $R_k=\{V \in x \mid V$ is used in expressions $r_1, r_2, \dots, r_k\}$ for $1 \leq k \leq n$. A is a *normal action sequence* if $l_j \notin R_j$ for $1 \leq j \leq n$.

The algorithm depicted in figure 1 change a given action sequence to a normalized one and returns its corresponding normalizer substitution. The algorithm renames the new appearances of the variables whose values are changed in an action sequence. The normalizer substitution of the action is in fact the set of variable renaming substitutions performed in the above process.

3 The notion of invariants

In this section we represent the notion of invariants on EFSM as introduced in [11] with little changes in the definitions. An invariant represents a specific property (which should be always true) on an EFSM which is in fact a statement about causal relationships between input/output pairs in the EFSM.

```
INPUT: An action sequence A=($l_1$ := $r_1$, … $l_n$ := $r_n$) in the EFSM
        M= M=(S, I, O, x, Tr, $s_{in}$)
OUTPUT: A normalizer substitution$\theta$
SIDE EFFECT: Action sequence A is changed to a normal one
Begin
    $\theta := \varnothing$;
    for $i:=1$ to $n$ do begin
        R $:= \{ V \in x \mid V$ is used in expressions $r_1, \dots, r_i \}$
        if $l_i \in R$ then begin
            $l'_i :=$ new $V$;    /* a new variable name */
            $\theta := \theta \cup \{ l_i / l'_i \}$;
            $l_i := l'_i$;
            for $j := i+1$ to $n$ do begin
                $l_j := l_j\,\theta$;  $r_j := r_j\,\theta$;
            end;
        end;
    end;
End;
```

Fig. 1. Normalizing an action sequence

Regarding the way of expressing the temporal relationships of the input/outputs in an EFSM, two types of invariants are introduced. We call them forward and backward invariants. Note that to define the invariants we only consider the control parts of the protocol so we do not speak about the values of the variables in input or output parameters. In the next section we represent algorithms to find corresponding constraints on the variables of a given invariant that makes it correct on the EFSM.

3.1 Forward invariants

A *forward invariant* is used to express properties in the EFSM such as "*each time the implementation performs a specific execution trace like i_1/o_1, ..., i_{n-1}/o_{n-1} , i_n , the next observed output belongs to a specific set of output symbols*". Based on this definition, we can assume that a forward invariant contains three elements: A preamble I/O sequence, a preamble input and a test output set. Intuitively a forward invariant is correct if for all paths in the EFSM matching with the preamble I/O sequence, and followed by an I/O pair containing an input equal to the preamble input, then the corresponding output essentially belongs to the test output set.

Definition 7. Let M=(S, I, O, x, Tr, s_{in}) be an EFSM. We say that the F(PIO, PI, TOS) is a forward invariant for M if the following conditions are respected :

1. PIO is the preamble I/O sequence which is defined according to the following EBNF:

$$PIO ::= a/z, PIO \mid *, PIO \mid \varepsilon$$

In which $a \in I \cup \{?\}$, $z \in O \cup \{?\}$and ε is the null sequence.

2. $PI \in I$ is the preamble input and $TOS \subseteq O$ is the test output set.
3. Each time that the sequence PIO is matched with any path in the EFSM, and it is followed by any transition with input PI, then we get essentially an output belonging to TOS.

Note that we deal with the wildcard ? as the standard one in pattern matching, while modify the usual meaning of the symbol *. The symbol * replaces any sequence of input/outputs not containing any pair with input equal to PI.

3.2 Backward invariants

Using a *backward invariant* we can express more subtle properties such as "*each time a specific output is produced by the implementation, then we must have that a specific trace had been produced before*". So a backward invariant contains three elements: A preamble output set, a test input and a test I/O sequence. Intuitively a backward invariant is correct if any transition in the EFSM in which its output symbol belongs to the preamble output set, have an input equal to test input and essentially preceded by a path matching with the test I/O sequence.

Definition 8. Let $M=(S, I, O, x, Tr, s_{in})$ be an EFSM. We say that the B(TIO, TI, POS) is a backward invariant for M if the following conditions are respected :

1. TIO is the test I/O sequence which is defined according to the following EBNF:

$$TIO ::= a/z, TIO \mid *, TIO \mid \varepsilon$$

in which $a \in I \cup \{?\}$, $z \in O \cup \{?\}$ and ε is the null sequence.

2. $TI \in I \cup \{?\}$ is the test input and $POS \subseteq O$ is the test output set.
3. All transitions of M with an output symbol belonging to POS must essentially have an input symbol equal to TI and proceed by a path matching with TIO.

Let us remark that, in contrast with forward invariants (for the case of preamble input symbol), we do not force the test input symbol here to be an input action (it can be also the wildcard character "?"). Furthermore, our matching method is modified such that the symbol * replaces any sequence of input/outputs not containing any pair with input equal to TI.

4 Extracting invariant constraints

To use an invariant for passive testing, it is needed to assure at first about the correctness of the invariant on the specification. While checking invariants on a FSM simply returns a Boolean value showing the correctness or fail of the invariant, checking an invariant on an EFSM either returns simply a false Boolean value showing that the invariant is incorrect on the EFSM or returns a set of constraints on the variables of the invariant showing that the correctness of the invariant depends on the set of constraints. For passively testing the implementation, it is sufficient to match the execution trace with the invariant while its constraints regarding the value of the variables in the trace don't conflict. In this section we represent algorithms for checking forward and backward invariants on an EFSM and extracting their corresponding constraint set. The constraint extraction process is done once and off-line.

4.1 Forward invariant constraints

To check a given forward invariant on an EFSM, first we have to find the paths in the EFSM which are unifiable with the preamble part of the invariant. After that we should check if the invariant test set is reachable using all the unified paths or not, and if it is reachable, then what is the constraint set to make it true. The constraint set is in fact constructed during the unification of the preamble part with the paths in the EFSM.

Definition 9. Let $\rho = i_1/o_1,\ldots,i_n/o_n$ be an input sequence of size n and M=(S, I, O, x, Tr, s_{in}) be an EFSM, we define U_n as the set of *forward matchers* of ρ containing quadruples (s,θ,C,δ) in which s is a state belongs to S, θ and δ are substitutions and C is a set of constraints (conjoined predicates) which is constructed inductively as follows:

- The initial forward matcher set is equal to $U_0=S\times\{\emptyset\}\times\{\emptyset\}\times\{\emptyset\}$
- If $t=(s,s',a,z,P,A)\in Tr$ is a transition in M, and U_{j-1} contains a forward matcher quadruple (s,θ,C,δ) such that $(a/z)\delta$ is unifiable with (i/o), then U_j contains quadruples (s',θ',C',δ') in which $\theta'= \theta_o mgu((a/z)\delta, (i/o))$, $C'=C\cup(P\wedge normalized (A))\delta$, and δ' is the normalizer of A.

Using the above definition we can describe our algorithm for checking the correctness of a given forward invariant on an EFSM and extracting its necessary constraints.

Let F(PIO, PI, TOS) be a forward invariant in which PIO is of size n. Suppose that U_n is the forward matcher set of PIO. If U_n is empty then the invariant is incorrect, else we check that for any transition labeled by the input PI , we receive an output unifiable with one of the items in the TOS. If there is no possible transitions, then the invariant is incorrect, else for each forward matcher quadruple (s,θ,C,δ) in U_n , if C is empty then the invariant is true, else the invariant is true constraint to $C\theta$. The set of constraints $C\theta$ can be simplified using the existing constraint simplification algorithms. The algorithm is depicted more formally and detailed in figure 2.

The algorithm deals with invariants containing the wildcard character *. Also we consider that both i=? and o=? hold. We have used some auxiliary functions: *head(I)* returns the first i/o couple of the sequence I and *tail(I)* removes the first i/o couple from I. the Boolean function *path(s, s', i)* returns true if there exist a path $a_1/z_1, \ldots a_r/z_r$ from s to s' and for any $1 \leq j \leq r$ we have $a_j \neq i$. Also the function *simplified(C)* returns the simplified version of the constraint set C. In fact this function solves the constraints such that the most constraining predicates on a single variable are remained. We don't enter in the details of this function. There are some well known methods to do this in the literature [12].

4.2 Backward invariant constraints

To check a given backward invariant on an EFSM, first we have to find the set of transitions in the EFSM which have outputs unifiable with elements of the preamble output set in the invariant. After that, we should check whether the paths in the EFSM which are ended by the discussed outputs are unifiable with the test input and test I/O sequence in the invariant or not. And, if they are unifiable, what are the constraints on the variables of the invariant. The constraint set is in fact constructed during the unification of the test I/O sequence with the paths in the EFSM. We traverse the paths in the EFSM in a backward fashion to do the unification and extract the constraints.

Definition 10. Let $\rho = i_1/o_1,\ldots,i_n/o_n$ be an input sequence of size n and M=(S, I, O, x, Tr, s_{in}) be an EFSM, we define V_0 as the set of *backward matchers* of ρ containing quadruples (s,θ,C,δ) in which s is a state belongs to S, θ and δ are substitutions and C is a set of constraints (conjoined predicates) which is constructed inductively as follows:

```
Input: M=(S, I, O, x, Tr, $s_{in}$), I=F(PIO,PI,OTS)
Output: true/false or a set of constraints. Satisfaction of each constraint is sufficient to sat-
isfy the invariant.

Begin

 /* PIO Matching: Finding the paths in the EFSM which are unifiable with the PIO */
  $I' := PIO;\ U := S \times\{\varnothing\}\times\{\varnothing\}\times\{\varnothing\};$
  while $I' \neq \varepsilon$ and $U \neq \varnothing$ do begin
        $first = head(I');\ I' = tail(I');$
        if $first \neq *$ then begin  /* first = i/o */
           $T := Tr;\ U' := \varnothing;$
            while $T \neq \varnothing$ do begin
                  $choose\ t \in T;$   /* t=(s,s',a,z,P,A) */
                  $T := T-\{t\};$
                   if $(s,\theta,C,\delta) \in U$ and $unifiable((a/z)\delta,\ i/o)$ then begin
                        $\theta' := \theta_o mgu((a/z)\delta,\ i/o);\ C' := C \cup (P \wedge normalized(A))\delta;$
                        $\delta' := normalizer(A);\quad U' := U' \cup \{(s',\theta',C',\delta')\};$
                  end
            end
           $U = U';$
        end
        else begin   /* first= * */
              while $head(I') = *$ do $I' := tail(I');$
             $first := head(I');$   /* first=i/o */
             $U := \{ (s,\theta,C,\delta) ) \mid s \in S,\ \exists (s,\theta,C,\delta) \in U,\ p = path(s',s,i) \};$
         end
  end

 /* TOS checking: Checking if TOS is reachable using the unified path or not and if so
    what is its constraints*/
  if $U = \varnothing$ then return(false);
  else begin
        $tf :=$ false; $T := Tr;\ CS := \varnothing;$
        while $T \neq \varnothing$ do begin
            $choose\ t \in T;\quad T := T-\{t\};$   /* t=(s,s',a,z) */
            if $(s,\theta,C,\delta) \in U$ and $unifiable(a\delta,\ i_n)$ then begin
                  $\theta' := \theta_o mgu(a\delta,\ i_n);$
                  if $\nexists o \in O \bullet (z\delta)\theta' = o\theta'$ then return(false);
                  $tf :=$ true; $CS := CS \cup simplified(C \cup (P \wedge normalized(A))\delta);$
            end
         end
  end
  if not tf then return false;
  if $CS = \varnothing$ then return true else return (CS);
End
```

Fig. 2. Algorithm for checking forward invariants and finding its corresponding constraints

– The initial backward matcher set is equal to $V_n = S\times\{\varnothing\}\times\{\varnothing\}\times\{\varnothing\}$
– If t=(s, s', a, z, P, A)∈Tr is a transition in M, and V_{j+1} contains a backward matcher quadruple q=(s',θ,C,δ) such that (a/z)δ is unifiable with (i_j/o_j), then V_j contains quadruples q'=(s,θ',C',δ') in which θ'= $\theta_o$mgu((a/z)δ, (i_j/o_j)), δ' is the normalizer of A and C' = Cδ' ∪ (P ∧ normalized (A)). Delete the quadruple q from $V_{j+1.}$

Using the above definition we can describe our algorithm for checking the correctness of a given backward invariant on an EFSM and extracting its necessary constraints.
Let B(TIO, TI, POS) be a backward invariant. For all elements $o_m \in POS$ ($1 \leq m \leq |POS|$ in which $|POS|$ is the cardinality of POS) we concatenate the pair TI/o_m to the TIO to generate a set of input/output sequences like ρ_m = "TIO, TI/o_m". Let the size of ρ_m be n. For each ρ_m ($1 \leq m \leq |POS|$) we try to find its backward matcher set. If after constructing V_j ($0 < j \leq n$) in each iteration, V_j is empty or V_{j+1} is not empty, then the invariant is incorrect, else for each quadruple of the V_0, if C is empty, then the invariant is true without any condition, else the invariant is true constraint to simplified $C\theta$. The algorithm is depicted more formally and detailed in figure 3. Auxiliary functions used are the same as described for the forward invariant checking algorithm.

5 An example: SCP protocol

In this section we present the processing of the method we discussed in this paper for passively testing an implementation of the Simple Connection Protocol (SCP) to show the applicability of the method. SCP has the advantage of including most difficulties of passive testing in a protocol specification, and then is able to figure out the applicability of the algorithm on bigger protocols.

5.1 The Simple Connection Protocol

SCP allows us to connect an entity called upper layer to an entity called lower layer. The upper layer performs a dialogue with SCP to fix the quality of service desirable for the future connection. Once the negotiation is finished, SCP dialogues with the lower layer to ask for the establishment of a connection satisfying the previously negotiated quality of service. 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 con-

nection 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. Figure 4 shows the interactions of the SCP with its upper and lower layers.

```
Input: M=(S, I, O, x, Tr, $s_{in}$), I=B(TIO,TI,POS)
Output: true/false or a set of constraints. Satisfaction of each constraint is sufficient to
        satisfy the invariant.
Begin
  /* POS Matching: Finding states in the EFSM which are unifiable with elements of the POS */
  $T := Tr$;  $V := S \times\{\emptyset\}\times\{\emptyset\}\times\{\emptyset\}$;  error:=false;
  while $T \neq \emptyset$ and not error do begin
        choose $t \in T$;   /* t=(s,s',a,z,P,A) */
        $T := T-\{t\}$;
        if $\exists o \in POS \bullet unifiable((a/z)\delta, (TI/o))$ then begin
              $\theta := mgu(a/z, PI/o)$;          $\delta := normalizer(A)$;
              $C := P \wedge normalized(A)$;     $V := V \cup \{(s, \theta, C, \delta)\}$;
        end else error :=true;
  end;
  if $V=\emptyset$ then  error :=true;

 /* TIO matching: Checking if the TIO is matched with all paths ending to the states found in
    previous step or not and if so what is the constraint set   */
  $I' := reverse(TIO)$;
  while not empty(I') and not error do begin
        $V' := \emptyset$;     first :=head(I');     I':=tail(I');
         if $first \neq *$  then begin   /*  first = i/o */
              $T := Tr$;
              while $T \neq \emptyset$ do begin
                    choose $t \in T$;     /* t=(s,s',a,z,P,A) */
                    $T := T-\{t\}$;
                    if $(s', \theta, C, \delta) \in V$ and $unifiable((a/z)\delta, i/o)$ then begin
                          $\theta' := \theta_o mgu((a/z)\delta, (i/o))$;       $\delta' := normalizer(A)$;
                          $C' := C\delta' \cup (P \wedge normalized(A))$;    $V' := V' \cup \{(s, \theta', C', \delta')\}$;
                          $V = V-\{(s', \theta, C, \delta)\}$;
                    end  else error := True;
              end
         end  else begin     /*  first =*   */
                while head(I') = *   do  I' := tail'(I');     /* skip a seq. of  *'s */
                 first := head(I');     /* first = i/o */
                 $V' := \{ (s, \theta, C, \delta) \mid s \in S,\ \forall (s', \theta, C, \delta) \in V \bullet path(s, s', o)\}$;
              end
         if $V \neq \emptyset$ then error:= true  else $V := V'$;
  end;
  if error then return (false);
  $CS := \emptyset$;
  while $V \neq \emptyset$ do begin
      choose $v \in V$;        /*  v= $(s,\theta,C,\delta)$   */
      $V := V-\{v\}$;     $CS := CS \cup simplify(C)$;
  end;
  if $CS=\emptyset$ then return(true) else return(CS);
End;
```

Fig. 3. Algorithm for checking backward invariants and finding its corresponding constraints

5.2 Defining invariants

Let consider the EFSM specification of the Simple Connection Protocol depicted in figure 5. We suppose that the values of TryCount, ReqQos, FinQos, CONreq.qos, and accept.qos are defined in the interval [0;3]. Suppose that we want to passively test an implementation of the SCP regarding the following properties of the specification which are described using the invariants:

- I_1 = B (< refuse/connect(x) > , (refuse) , { CONcnf(-) }), means that SCP fail to connect the two layers (CONcnf(-)) only if the lower layer refused the connection twice before (refuse/connect(x), refuse/).
- I_2 = F (< CONreq(x)/connect(y) > , (accept(w)) , { CONcnf(+,z) }) , means that if SCP accepts to connect with the upper layer at his requested QoS (CONreq(x)/connect(y)) and the lower layer accept it at a given QoS, then a connection must be realized between the two layers.
- I_3 = F (<> , (accept(x)) , { CONcnf(+,y) }) , means that if the lower layer accept the connection (accept(x)), this connection must be realized (CONcnf(+,y)).
- I_4 = B (<> , (accept(x)) , { CONcnf(+,y) }) , means that a connection is realized (CONcnf(+,x)), only if the lower layer accepted it before (accept(y)).

Note that I_1 and I_4 are forward invariants while I_2 and I_3 are backward invariants. In definition of the above invariants we have used a control driven approach i.e. in this stage, parameters of the signals are not important so we have used some variables instead of them.

5.3 Finding invariant constraints

Now, we apply our method on the invariants to find their corresponding constraints. Table 1 shows the trace of the algorithms. For each forward (backward) invariant, the value of the intermediate forward (backward) matcher set i.e. U (i.e. V) and the ultimate constraint sets CS over the variables of the invariants have been shown. (See the algorithms in figures 2 and 3).

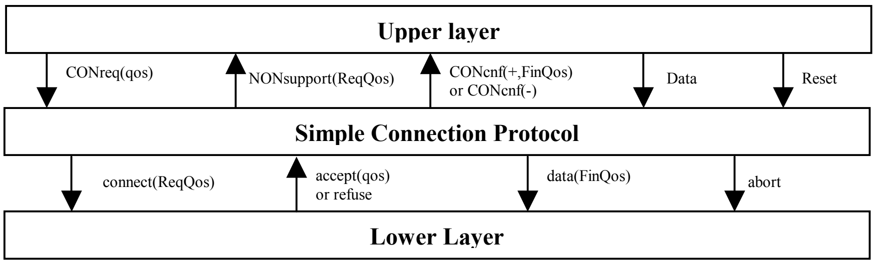
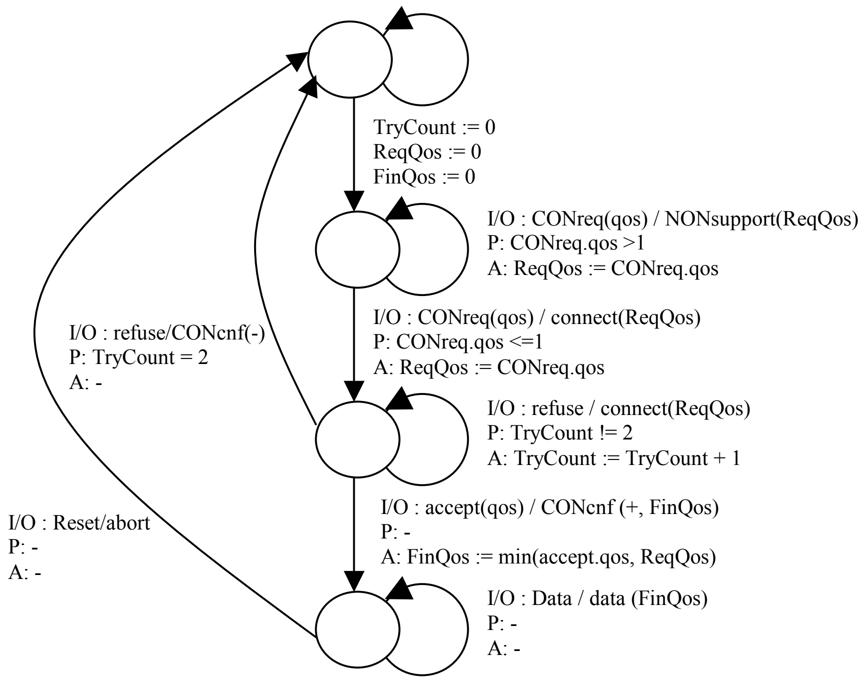


Fig. 4. Interactions of the SCP with its upper and lower layers


TryCount := 0
ReqQos := 0
FinQos := 0
I/O : CONreq(qos) / NONsupport(ReqQos)
P: CONreq.qos >1
A: ReqQos := CONreq.qos
I/O : CONreq(qos) / connect(ReqQos)
P: CONreq.qos <=1
A: ReqQos := CONreq.qos
I/O : refuse/CONcnf(-)
P: TryCount = 2
A: -
I/O : refuse / connect(ReqQos)
P: TryCount != 2
A: TryCount := TryCount + 1
I/O : accept(qos) / CONcnf (+, FinQos)
P: -
A: FinQos := min(accept.qos, ReqQos)
I/O : Reset/abort
P: -
A: -
I/O : Data / data (FinQos)
P: -
A: -
Fig. 5. EFSM specification of SCP

Applying the algorithms reveal that all the invariants are correct regarding the control part of the specification, but regarding the data part of the specification the invariants are true condition to some constraints which have been produced by the algorithms. For invariant I_1, there is no constraint over the variable of the invariant, so it should be matched by execution traces with any value for the variable x. For the other invariants, only such execution traces are matched with the invariants that the value of their input/output parameters does not cause any conflict with the corresponding constraints of the intended invariant.

5.4 Passive testing using the constrained invariants

Now suppose that the following execution traces are generated by a faulty implementation of the SCP:

- Trace_1 = CONreq(1) / connect(1) , refuse / CONcnf(-)
- Trace_2 = CONreq(1) / connect(0) , accept(1) / CONcnf(+, 0).

Table 1. Using algorithms to extract required constraints for the example invariants

Invariant	**U_1(or V_1)**	**U_2 (orV_2)**	**Constraint set (CS)**
I_1 **(Back-ward)**	$V_1 = \{ (s_3, \theta_1, C_1, \delta_1) \}$ $\theta_1 = \varnothing$ $C_1 = \{ \text{TryCount} = 2 \}$ $\delta_1 = \varnothing$	$V_2 = \{ (s_3, \theta_2, C_2, \delta_2) \}$ $\theta_2 = \{ \text{ReqQos/x} \}$ $\delta_2 = \{ \text{TryCount/y} \}$ $C_2=C_1\delta_2\cup$ {TryCount!=2,y=TryCount +1}= {y=2,TryCount!=2,y=TryCount +1} = { TryCount =1}	CS_I_1= $C_2\theta_2$= {TryCount=2}
I_2 **(For-ward)**	$U_1 = \{ (s_3, \theta_1, C_1, \delta_1) \}$ θ_1 = { CONreq.qos/x, ReqQos/y } C_1 = { CONreq.qos <=1, ReqQos = CONreq.qos } $\delta_1 = \varnothing$	$U_2 = \{ (s_4, \theta_2, C_2, \delta_2) \}$ $\theta_2=\theta_1$o{accept.qos/w, FinQos/z } ={CONreq.qos/x, ReqQos/y, accept.qos/w, FinQos/z } $C_2=C_1\cup${CONreq.qos= min(accept.qos, ReqQos) }δ_1 ={CONreq.qos <=1, ReqQos=CONreq.qos, FinQos=min(accept.qos, ReqQos)} $\delta_2 = \varnothing$	CS_I_2=$C_2\theta_2$= {x<=1, y=x, z=min(w, y)}
I_3 **(For-ward)**	$U_1 = \{ (s_3, \theta_1, C_1, \delta_1) \}$ θ_1 = { accept.qos/x, FinQos/y } C_1 = { y=min(x, ReqQos) } $\delta_1 = \varnothing$		CS_I_3=$C_1\theta_1$= {y=min(x, Re-qQos)}
I_4 **(Back-ward)**	$V_1 = \{ (s_4, \theta_1, C_1, \delta_1) \}$ θ_1 = { accept.qos/x, FinQos/y } C_1 = { y=min(x, ReqQos) } $\delta_1 = \varnothing$		CS_I_4=$C_1\theta_1$= {y=min(x, Re-qQos)}

We know that a transition error has occurred in the first trace because the specification forces two loops on state s_3 before eventual transition to s_1, corresponding to the three requests SCP must do before failing the connection. In this trace, the connection is said to be failed on first try. For the second trace, there is an output error because the first I/O couple should be CONreq(1)/connect(1). We can imagine that the trace comes from an implementation in which the action on the transition from s_2 to s_3 is ReqQos:=CONreq.qos –1 and then such a trace is produced. This error has for consequence to connect the upper and lower layers with a QoS equals to 0 when it could be (normally) equal to 1.

Tables 2 and 3 show the invariants used in the checking of the first and the second trace respectively. We try to identify the constraints with the values of the variables extracted from the traces:

- **Trace$_1$:** Since the analysis found CONcnf(-) in the trace and failed looking for the couple refuse/connect(x), then the trace is erroneous regarding the invariant

I_1. Note that the found error is control driven, so it is not needed to look at the constraints at all.

- **Trace$_2$:** There are three invariants which are candidate for this trace. Matching the trace with the invariants shows that there is not any control driven error, so we use constraints and the value of the variables in the trace to decide about the possibility of data driven errors:

CS-I_2 $\cup$ {x=1, y=0, w=1, z=0}={x<=1, y=x, z=min(w, y)} $\cup$ { x=1, y=0, w=1, z=0 }

= { 1=<1, 0=1, 0=min(1, 0) }

1=<1 is true, 0=1 is false and 0=min(1, 0) is true, so the invariant I_2 is false on Trace$_2$.

CS-I_3 $\cup$ {x=1, y=0}={y=min(x, ReqQos)}$\cup${x=1, y=0} = { 0=min(1, ReqQos) }

0 is the minimum of 1 and ReqQos only if ReqQos is equal to 0, so the invariant I_3 is true on the trace Trace$_2$ if ReqQos=0.

CS-I4 $\cup${x=1, y=0}={y=min(x, ReqQos)} $\cup$ { x=1, y=0 } = { 0=min(1, ReqQos) }

0 is the minimum of 1 and ReqQos only if ReqQos is equal to 0 so the invariant I_4 is true on the trace Trace$_2$ if ReqQos=0.

As we found an inconsistency in the checking of the invariant I_2 with the trace Trace$_2$ we conclude that the trace T_2 is false. Checking the other invariants on the trace is not necessary but figure here as an example of variable simplification.

Table 2. Using invariant I_1 to check the execution trace Trace$_1$

Trace$_1$	CONreq(1)	connect(1)	Refuse	CONcnf(-)
I_1	Refuse	connect(x)	Refuse	CONcnf(-)

Table 3. Using invariants I_2, I_3 and I_4 to check the execution trace Trace$_2$

Trace$_2$	CONreq(1)	connect(0)	Accept(1)	CONcnf(+,0)
I_2	CONreq(x)	connect(y)	Accept(w)	CONcnf(+,z)
I_3			Accept(x)	CONcnf(+,y)
I_4			Accept(x)	CONcnf(+,y)

6 Conclusions

Passive testing methods for network protocols can be classified into naïve and invariant based approaches. In the naïve approach the implementation trace which is recorded during the execution of the protocol under test is compared with total of the specification in a forward or backward manner. This is where, in the invariant based approach only critical properties of the specification (i.e. invariants) which are extracted by an expert are compared with the implementation trace. By using invariant based approach, not only a lot of extra processing is reduced, but also we can focus on the critical properties of the program under test.

Passive testing methods can be compared from another aspect. Some methods are limited to testing only control driven aspects of the implementation, i.e. the order of occurrences of the input/output signals, while other methods are capable of testing both control driven and data driven aspects of the implementation i.e. the values of the signal's parameters. For testing control driven aspects it is sufficient to use FSM for specification, while for data driven aspects it is neded to use EFSM.

In this paper we presented a new method for passive testing of both control driven and data driven aspects of the network protocols using an invariant based approach. The intended properties of the specification are expressed using some control driven invariants given by an expert. After that, using the given algorithms, the invariants are checked on the specification off-line. Also to take the data driven aspects into account, for the correct invariants, some constraints over the variables of the invariants are extracted. For passively testing the implementation traces, it is sufficient to compare, on-line, the trace with the invariants regarding the constraints using pattern matching. A trace is correct while it is matched with the invariant and the invariant's constraints are not conflicting regarding the values of the signal's parameters.

To show the applicability of the presented method, passive testing of the Simple Connection Protocol (SCP) using the presented method was illustrated.

References

1. R. Lai, “A survey of communication protocol testing”, Journal of Systems and Software 62(1): 21-46 (2002).
2. D. Lee, and M. Yannakakis, “Principles and methods of testing finite state machines---A survey”, Proc. IEEE 84, 8, (1996), 1089--1123.
3. 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

4. R. E. Miller and K. A. Arisha, "On Fault Location in Networks by Passive Testing", IPCCC 2000, Pheonix, AZ, Feb. 2000.
5. M. Tabourier and A. Cavalli, “Passive Testing and application to the GSM-MAP Protocol”, in Journal of Information and Software Technology 41(11) (15 Sept. 1999), Pages 813-821, Elsevier, 1999.
6. J. A. Arnedo, A. Cavalli and M. Nunez, “Fast Testing of Critical Properties through Passive Testing”, LNCS, vol. 2644/2003, Pages 295-310, Springer, 2003.
7. A. Cavalli, C. Gervy and S. Prokopenko, “New Approaches for Passive Testing Using an Extended Finite State Machine Specification”, in Journal of Information and Software Technology, 45:837-852, Elsevier, 2003.
8. D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin, "A Formal Approach for Passive Testing of Protocol Data Portions", Proc. ICNP'2002.
9. B. Alcalde, A. Cavalli, D. Khuu, D. Chen, D. Lee, “Network Protocol System Passive Testing for Fault Management - a Backward Checking Approach”, in the Proceedings of the 24th IFIP WG 6.1, International Conference on Formal Techniques for Networked and Distributed Systems, FORTE 2004, 27-30 September, 2004, Madrid, Spain.
10. F. Baader, W. Snyder, “Unification Theory, Handbook of Automated Reasoning”, Alan Robinson, Andrei Voronkov, eds., Vol. 1, Chapter 8, 446–533.
11. E.Bayse, A. Cavalli, M. Nunez and F. Zaidi, “A Passive Testing Approach based on Invariants: Application to the WAP”, To be published in journal of Computer Network, 2004.
12. K. Marriott and P. J. Stuckey, “Programming with Constraints: An Introduction”, Book, The MIT Press, 1998.