New approaches for passive testing using an Extended Finite State Machine specification

Ana Cavalli*, Caroline Gervy, Svetlana Prokopenko

Institut National des Télécommunications, 9 rue Charles Fourier, Evry cedex 91011, France

Abstract

This paper presents two new approaches for passive testing using an Extended Finite State Machine (EFSM) specification. The state of the art of passive testing shows us that all the methods for detection of errors based on EFSMs try to match the trace to the specification. Indeed, one searches a state succession in the specification machine that is able to generate the trace observed on the implementation. Using this approach, processing is performed on the specification and the trace remains in the background since no operation is applied to it. This made us realise that focusing our efforts on the trace could be beneficial and has given us two approaches presented in this paper that extract information from the specification and then work on the trace. Thus, they take a different direction than the previous methods. We first present an approach to test traces by using invariants resulting from the specification. We formally define these invariants and we see how to extract them. We also discuss their ability to detect errors appearing in the implementation. This approach is able to test the data flow, but not in a very satisfactory way. This is the reason for a second approach seeking to apply a set of constraints to the trace. We develop in detail its principles. Both approaches are applied to a Simple Connection Protocol (SCP) and the results of preliminary experiments are presented.

Keywords: Simple connection protocol; Passive testing; Extended Finite State Machine

1. Introduction

Passive testing involves examining the input/output behavior of an implementation without predefining the input. The first step is to observe the input/output behavior in order to derive a trace. Once this tracing information is available, we determine whether it conforms to the specification associated with the implementation under test. If this is the case, then no conclusions can be made concerning the validity of the implementation. In fact, another trace could find errors that are not revealed by the current trace. If errors of conformance are found then we can conclude that the implementation does not conform.

The advantage of passive testing over active testing is its capability of working over the network and with the implementation under operation. Active testing disturbs the network and operation and could provoke the crash of the service. On the contrary passive testing only observes and does not intervene. Another advantage of this type of testing is that it can be used to detect unwanted intrusions in the network that have modified the input/output behavior. These modifications can be detected but passive testing cannot yet be effectively used since the error detection algorithms need to be improved and made more efficient.

In the last five years, numerous research works have been devoted to passive testing. Most of these works were dedicated to passive testing based on Finite State Machines (FSMs). For example, in Ref. [1] some techniques to detect errors in deterministic and nondeterministic FSMs are proposed. On the other hand, Refs. [3,4,10] explore the domain of deterministic Communicating FSMs focalising their work in error detection and localisation. Nevertheless, few works have been dedicated to EFSMs because the introduction of variables raises a lot of unresolved problems. However, the work presented in Ref. [2] proposed several solutions for errors detection. Taking into account this research and those mentioned above, we continue this work to define new approaches for errors detection. This paper presents these new approaches, their advantages and limitations.

While studying the state of the art of passive testing, we realised that all the methods for error detection in the context
of EFSMs try to match the trace to the specification. Indeed, they search a state succession in the specification machine that is able to generate the trace observed in the implementation. If the succession is found, it is impossible to conclude that the conformity of the implementation holds. On the contrary, if there is no succession of states, we know that the implementation contains an error and we can say that it does not conform to its specification.

Using these approaches, the specification is processed but the trace remains in the background since no operation is applied to it. Indeed, these methods extract the necessary information from the trace and then launch various processes on the specification machine. This phenomenon convinced us that efforts needed to be made that focused more on the trace. The two approaches presented in this paper extract information from the specification and then process the trace. Thus, the work here goes in a different direction than that of the previous methods.

In this paper, we propose the first approach which is to test the trace using invariants resulting from the specification. We define these invariants and we show how both to extract them and use them in the best way possible. Finally, in this paper we extract information from the specification and then launch various processes on the trace. We develop in detail its principles and the various stages necessary to its application.

The paper is organised as follows. Section 2 provides the definition of the concept of EFSM. Section 3 presents the new approaches for passive testing. Section 4 presents the preliminary results of the application of the two approaches to a Simple Connection Protocol (SCP) [11]. Finally, Section 5 gives the conclusions and perspectives of this research work.

2. Preliminaries

An extended finite state machine (EFSM) is a 6-tuple \( M = (I, O, S, \bar{x}, T, s_0) \) where \( I, O, S, \bar{x} \) and \( T \) are finite sets of input symbols, output symbols, states, variables and transitions, respectively, and \( s_0 \) is an initial state. Each transition \( t \) of the set \( T \) is a 6-tuple: \( t = (s_t, s'_t, e_t, o_t, P_t, A_t) \) where \( s_t, s'_t, e_t \) and \( o_t \) are the start (current) state, end (next) state, input and output, respectively. \( P_t(\bar{x}) \) is a predicate on the current variable values and \( A_t(\bar{x}) \) defines an action on variables values.

Initially, the machine is at an initial state \( s_0 \in S \) with initial variable values \( \bar{x}_0 \). Suppose that the machine is at the state \( s_t \) with the current variable values \( \bar{x} \). Upon an input \( e_t \), if \( \bar{x} \) is valid for \( P_t \), i.e. \( P_t(\bar{x}) = \text{TRUE} \), then the machine follows the transition \( t \), outputs \( o_t \), changes the current variable values by action \( \bar{x} := A_t(\bar{x}) \), and moves to the state \( s'_t \).

For each state \( s \in S \) and input \( e \in I \), let \( t_i = (s, s'_i, e, o_i, P_i, A_i), 1 \leq i \leq r \) denotes the transitions with the start state \( s \) and input \( e \). In a deterministic EFSM the sets of valid variable values of these \( r \) predicates are mutually disjoint, i.e. \( X_i \cap X_j = \emptyset, 1 \leq i \neq j \leq r \). Otherwise, the machine is nondeterministic.

As usually, by trace of an EFSM we mean a sequence of adjacent transitions of an EFSM.

In this paper, we consider only normalised EFSMs, which are composed of an unique module, and whose actions do not include complex instructions (WHILE, IF-THEN-ELSE, ...). An EFSM can always be transformed into a normalised EFSM [5].

3. Passive testing of EFSMs

3.1. Problems to be solved

An EFSM is composed of two different streams:

- **Control flow** that describes the way messages are to be sent and received. It can be considered as a simple FSM including states and transitions. At the beginning, the FSM is at the initial state. An input message constrains the FSM to produce one or several output messages and to move from the current to the next state. This procedure is called a transition.

- **Data flow** specifies other functions (for instance, the quality of service) that have an influence on the parameter values associated to messages. These functions can be described with the help of an FSM.

The main difficulty is the processing of variables issued from data flow. In active testing, this problem has been studied [6–9] and even if the major obstacle remains the generation of executable test sequences, the values of variables are known. For passive testing, the execution is not a problem because we do not interfere with the implementation. But, we do not know the variable values and the machine can be in any of the \( n \) states at the beginning of the trace. It is to solve these problems that we propose the following approaches.

3.2. The first approach: Use of invariants

3.2.1. Principle

We extract a set of properties from the EFSM specification, then we study the trace resulting from the implementation to determine whether it validates this set of properties. The extracted properties have to hold at every moment, we call them invariants.

This new approach is composed of two steps:

- the step of extraction of the invariants resulting from the EFSM,
- the step of checking the validity of the trace using the invariants extracted before.
Our first idea was to find general invariants describing some characteristic behaviors of the studied specification. Thus, we wanted to describe links of cause for purpose between the various events using temporal logic. This idea was abandoned because the trace length introduces some problems. Indeed, we consider an invariant expressing the fact that the event $e_2$ must occur after the event $e_1$. However, by checking the trace we detect $e_1$ but the trace stops before $e_2$ occurs. Thus, we can conclude wrongly that the implementation is erroneous. This is why we were interested in more detailed invariants based, for instance, on the successions of input/output symbols. We will call them input/output invariants. They allow us to describe properties such as ‘after the input symbol $x$, we must have the output symbol $y’ or ‘before the output symbol $z$, we must have the input symbol $v’.

3.2.2. Definition of input/output invariants

An input/output invariant is composed of two distinct parts: the preamble and the test. The preamble is the part that needs to be common with the trace to launch the test. We define several types of invariants.

Output invariants allow us to express properties such as ‘after such symbol(s), we must have such output symbol’. To use them, it is necessary to read the trace from left to right and to search for the pattern, which is used as preamble before the test is launched.

Input invariants allow us to express properties such as ‘before such symbol(s), we must have such input symbol’. To use them, it is necessary to read the trace from right to left and to search for the pattern, which is used as preamble before launching the test.

Succession invariants allow us to express properties more subtle than those of the input and output invariants. Indeed, some specifications use loops allowing, for example, to make several connection attempts before being definitively rejected by the system (e.g. we must restart the connection procedure from the beginning). The succession invariants will enable us to detect if an implementation does not authorise several attempts before a final rejection.

**Example 1.** There are several types of invariants resulting from the SCP specification (see Section 4.1).

- The following sequence is an output invariant of length 1
  \[ CONreq(qos) / NONsupport(reqQos) \]

  preamble \hspace{1cm} text

- The following sequence is an output invariant of length 2
  \[ CONreq(qos)/connect(reqQos)(accept(qos)) \]

  preamble \hspace{1cm} text

Thus, if we find in the trace the pattern \( CONreq(qos)/connect(reqQos))(accept(qos)/YYY) \) then we must check that the pattern \( YYY \) is equal to \( CONcnf(+/FinQos) \). If so, we must continue to check the trace, if not, it means that an error has been found. In this case, the process is stopped and the error notified.

3.2.3. Control and data flows

The use of invariants does not raise a problem for the control flow but we must be careful with the data flow. Indeed, we have control invariants (e.g. no variable appears in the messages of these invariants) and data invariants (e.g. including parameterised messages).

The problem with the parameterised messages is that we have to check that the parameter values are compatible with their respective definition fields. However, we do not try to determine the variable values, instead, we try to check that they are compatible with this elementary property that is the checking of the definition field. For this, we will assign to each variable $v$ of the system a $val(v)$ set which contains all the possible values for $v$ (we propose to use intervals to define $val(v)$ more briefly). Thus, during the trace analysis, we will have to check that the value presented in the trace validates the variable definition field that is associated with it.

This checking is possible if we know the signature (e.g. the set of parameters) of each message of the specification. Indeed, if the same message admits several signatures, we will not be able to know which parameter we are considering and, consequently, we will not be able to check if it validates its definition field.
Example 2. Consider a message $M_1$ with different parameters.

- $M_1(\text{param}_1, \text{param}_2, \text{param}_3)$ and $M_1(\text{param}_1, \text{param}_3)$. We have to distinguish between messages with parameters 1, 2, 3 and with parameters 1, 3. In this case there is easy because the number of parameters is different
- $M_1(\text{param}_1, \text{param}_2, \text{param}_3)$ and $M_1(\text{param}_1, \text{param}_4, \text{param}_5)$. It is impossible to distinguish $\text{param}_2$ from $\text{param}_3$ (and $\text{param}_3$ from $\text{param}_5$) if they have the same type.

3.2.4. Invariant length

Before extracting the invariants from a specification, it is necessary to answer the following question: ‘What should be the length of the extracted invariants?’ Indeed, it is not very interesting to look for long invariants because the longer the invariant, the less it is able to produce a test. To illustrate this phenomenon, we provide, in the following, the probability to produce a test for various invariant lengths. We will note $l_i$ the trace length, $I$ the set of the input symbols and $O$ the set of the output symbols.

\[
\begin{align*}
\text{Length} & = 1 \rightarrow l_i/|I| \\
\text{Length} & = 2 \rightarrow (l_i - 1)(|I|^2|O|) \\
\text{Length} & = 3 \rightarrow (l_i - 2)(|I|^3|O|^2) \\
& \quad \quad \cdots \\
\text{Length} & = k \rightarrow (l_i - k + 1)(|I|^k|O|^{k-1})
\end{align*}
\]

Case 1 and case 2 give an idea of the number of tests carried out depending on the length $k$ of an invariant (Tables 1 and 2).

Thus, an invariant of length 1 can give place to 600 times more tests than an invariant of length 3 in case 1 and 160,000 times in case 2. Therefore, the interest in searching for invariants of length 3 or more is very low. Thus, we search systematically for invariants of length 1 and 2; but it will be possible to extract longer invariants if it is necessary.

3.2.5. Invariant extraction

To extract the output invariants of length 1, it is enough to work on the set of input symbols $I$ and to look to see if a symbol $e \in I$ is not always followed by the same symbol $o \in O$.

To obtain output invariants of length 2, we use triplets of the form $(\text{input } 1, \text{output } 1, \text{input } 2)$. Of course, we will only deal with the ‘feasible’ triplets, e.g. the triplets which correspond to a succession of transitions in the specification and not to the Cartesian product of $I \times O \times I$.

Following are presented the steps of this process:

- First we need to associate with each feasible triplet $(e_t, o_t, e_k)$ a set $Out(e_t, o_t, e_k)$ containing all the output symbols corresponding to it. Thus, if there is a transition $t_1$ having an input symbol $e_t$ and an output symbol $o_t$ which is followed by a transition $t_2$ having an input symbol $e_k$ and an output $o_k$ then we add $o_k$ to $Out(e_t, o_t, e_k)$.
- Next, for each feasible triplet $(e_t, o_t, e_k)$ we check to see if $|Out(e_t, o_t, e_k)| = 1$. If it is the case, then we have detected an output invariant of length 2, else we pass to the next triplet.

Some adaptations allow us to extract the input invariants. For that, we have to deal with the set $O$ for the invariants of length 1 and with the feasible triplets of the form $(\text{output } 1, \text{input } 2, \text{output } 2)$ for the invariants of length 2. We generalise these methods for the invariants of length $n$ (algorithm 1).

Let $M = (I, O, S, X, T, s_0)$ be an EFSM. We denote by $IS_n$ the set of the output invariants of length $n$ ($n \geq 1$), by $IS = \bigcup_{i=1}^{\infty} IS_i$ the set of the output invariants. Let $\text{succ}(t_i)$ be the set of transitions of $T$ which can be carried out just after the transition $t_i$. Thus, the starting state of the transitions from $\text{succ}(t_i)$ is the arrival state of the transition $t_i$. Let $Out(e_1, o_1, \ldots, e_{n-1}, o_{n-1}, e_n)$ be the set of the output symbols associated with $(e_1, o_1, \ldots, e_{n-1}, o_{n-1}, e_n)$. If there are $n$ successive transitions $t_1, t_2, \ldots, t_n$ in $T$ (i.e. $t_1 \in \text{succ}(t_1), t_2 \in \text{succ}(t_2), \ldots$) having input/output pairs $(e_1, o_1), \ldots, (e_n, o_n)$ then $Out(e_1, o_1, \ldots, e_{n-1}, o_{n-1}, e_n) = \{o_n\}$. The function $\text{elem}$ takes as an input a set containing only one element and returns this element.

Algorithm 1. Derivation of the set of output invariants of length $n$.

Input: A specification EFSM $M$.

Output: A set of output invariant of length $n$. 

begin
  for each \((t_1, t_2, \ldots, t_n)\) in \((T \times \text{succ}(t_1) \times \cdots \times \text{succ}(t_{n-1}))\) do
    add \(a_n\) to \(\text{Out}(e_1, a_1, \ldots, e_{n-1}, a_{n-1}, e_n)\),
  end
  for each \((e_1, a_1, \ldots, e_n)\) in \((I \times O \times \cdots \times I)\) do
    if \(|\{\text{Out}(e_1, a_1, \ldots, e_{n-1}, a_{n-1}, e_n)\}| = 1\) do
      add \((e_1, a_1, \ldots, e_n, \text{elem}(\text{Out}(e_1, a_1, \ldots, e_{n-1}, a_{n-1}, e_n)))\) to \(I S_n\).
    end
  end
end

The complexity of this algorithm is \(O(|T|^n + |I|^n |O|^{n-1})\). Let \(m\) be such that \(m \geq |I|\), \(m \geq |O|\) and \(m \geq |T|\). We can limit complexity by the value \(O(m^n + m^{2^{n-1}}) = O(m^{2^{n-1}})\).

This complexity appears high, but one must keep in mind that this algorithm will be used with small values of \(n\) (\(n = 1\) and \(n = 2\)). Beyond these lengths the effectiveness of the found invariants is very reduced (see Section 3.2.4). Thus, its complexity will be \(O(m)\) for the invariants of length 1 and \(O(m^n)\) for the invariants of length 2.

To obtain succession invariants, the process is different. It is necessary to find loops presented in the specification and to check whether they can compulsory to be carried out several times before another event occurs. In this paper, we do not provide an algorithm for the extraction of these invariants because our experience has shown that they are not very relevant to detect errors. In fact, our experiments showed that input invariants and output invariants are much more suitable to detect errors than succession invariants. In addition, the algorithm for the extraction of succession invariants is equivalent to find loops in graphs, so it does not contribute anything new to the field.

3.2.6. Nondeterminism influence

We previously saw how to extract a set of invariants, but there still remains a significant point that needs to be taken into consideration: the influence of nondeterminism on a capacity to extract invariants. Indeed, in the case of nondeterministic specifications, the machines have several possible behaviors for the same input message from a given state. Thus, it is significant to study the effects of the nondeterminism in our method. There exist two types of nondeterminism, both presented in Fig. 1:

- the observable nondeterminism (case 1),
- the nonobservable nondeterminism (case 2).

In the first case, the input symbol is the same for the two transitions resulting from the state \(s_0\), but the output symbols are different. Thus, we will be able to distinguish the fired transition using the output symbol produced. We know what the fired transition is with certainty. In the second case, the input and output symbols are the same for the two transitions resulting from the state \(s_0\). Thus, it is impossible to distinguish between these two transitions. It is this incapacity to know the specification behavior which qualifies this form of nonobservable nondeterminism.

Section 3.2.6.1 studies the influence of the observable nondeterminism on our method and Section 3.2.6.2 studies that of the non observable nondeterminism.

3.2.6.1. Problems due to the observable nondeterminism. As we previously said, this section studies the influence of the observable nondeterminism on our capacity to extract the input and output invariants. To illustrate this we provide the following examples.

Example 3. Table 3 presents the influence of an observable nondeterminism on the invariants of length 1.

Example 4. Table 4 presents the influence of an observable nondeterminism on the invariants of length 2.

Example 3 shows that the observable nondeterminism does not have any influence on the extraction of input invariants of length 1. On the other hand, it has a very negative effect on the extraction of output invariants of length 1. Indeed, we did not succeed in extracting only one output invariant. This is unfortunate because the invariants of length 1 have the greatest capacity of error detection.

Example 4 also shows that the observable nondeterminism does not have an influence on the extraction of input invariants of length 2. However, for the output invariants of length 2, the results are more moderate. Indeed, when the nondeterminism appears on the first transition this does not have an effect on the extraction, but when it is declared on the second transition, we are again in the incapability of extract the output invariants.

Thus, we can conclude that the presence of the observable nondeterminism does not influence the extraction of input invariants, but it is hindrance for the extraction of output invariants.

3.2.6.2. Problems due to the nonobservable nondeterminism. In this section we will study the influence of nonobservable nondeterminism on the extraction of input and output invariants. As in the preceding part, we illustrate this with several examples.
### Table 3
Observable nondeterminism on the invariants of length 1

<table>
<thead>
<tr>
<th>Type</th>
<th>Specification</th>
<th>Invariants found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Input</td>
</tr>
<tr>
<td>Deterministic specification</td>
<td>( e_1/o_1 )</td>
<td>( e_1/o_1 )</td>
</tr>
<tr>
<td></td>
<td>( e_2/o_2 )</td>
<td></td>
</tr>
<tr>
<td>Nondeterministic specification</td>
<td>( e_1/o_1 )</td>
<td>( e_1/o_1 )</td>
</tr>
<tr>
<td></td>
<td>( e_1/o_2 )</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4
Observable nondeterminism on the invariants of length 2

<table>
<thead>
<tr>
<th>Type</th>
<th>Specification</th>
<th>Invariants found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Input</td>
</tr>
<tr>
<td>Deterministic specification</td>
<td>( e_1/o_1, e_2/o_2 )</td>
<td>( e_1/o_1, e_2/o_2 )</td>
</tr>
<tr>
<td></td>
<td>( e_2/o_2, e_2/o_2 )</td>
<td></td>
</tr>
<tr>
<td>Nondeterministic specifications</td>
<td>( e_1/o_1 )</td>
<td>( e_1/o_1 )</td>
</tr>
<tr>
<td></td>
<td>( e_1/o_2 )</td>
<td>( e_1/o_2 )</td>
</tr>
<tr>
<td></td>
<td>( e_1/o_2 )</td>
<td>( e_1/o_2 )</td>
</tr>
<tr>
<td></td>
<td>( e_1/o_1 )</td>
<td>( e_1/o_1 )</td>
</tr>
<tr>
<td></td>
<td>( e_1/o_2 )</td>
<td>( e_1/o_2 )</td>
</tr>
<tr>
<td></td>
<td>( e_1/o_2 )</td>
<td>( e_1/o_2 )</td>
</tr>
</tbody>
</table>
Example 5. In Table 5 the influence of a non observable nondeterminism on the invariants of length 1 is presented.

Example 6. Table 6 presents an influence of non observable nondeterminism on the invariants of length 2.

Example 5 shows that the non observable nondeterminism does not have any influence on the extraction of invariants of length 1. Example 6, for its part, shows that the non observable nondeterminism is seldom a hindrance for the extraction of invariants of length 2, except in the second case of nondeterministic specification where we cannot extract output invariants.

To conclude, the non observable nondeterminism does not influence the extraction of input invariants. However, it can be a hindrance for the output invariants.

3.2.7. Effectiveness of a set of invariants

Once we extracted a set of invariants, it is significant to be able to evaluate its effectiveness for the detection of errors. This effectiveness corresponds to the number of tests carried out on the trace due to the set of found invariants. Thus, as the effectiveness of a set of invariants increases, so does the number of tests applied to the trace.

There are two possible approaches:

- The first one consists in calculating the probabilities of the effectiveness of a set of invariants of an unspecified trace $t$ of length $l_t$. This method gives us an overall view of the effectiveness.
- The second approach precisely calculates the number of tests carried out on a given trace by a certain set of invariants. This method gives us a specific view of the effectiveness.

The first method: Probabilistic approach. Let $IS$ be a set of invariants of the form $e_1/o_1, \ldots, e_n/o_n$, where $n \geq 1$. $I$ be the set of input symbols, $O$ be the set of output symbols, $t$ be a trace of the form $e_1/s_1, \ldots, e_l/s_l$, where $l \geq 1$ is the length of the trace $t$, and $l(c)$ be a function which gives us the length of an invariant $c \in IS$. For an invariant $c$ of size $l(c)$, the probability of being found in a trace of length $l_t$ is calculated by the following formula:

\[
l_t - l(c) + 1
\]

\[|I|^{|O|}O^{l(c)-1}
\]

Thus, to know how many tests will be applied to a trace $t$ of length $l_t$, we can make the following calculation:

\[
\sum_{c \in IS} \frac{l_t - l(c) + 1}{|I|^{|O|}O^{l(c)-1}}
\]

The second method: Evaluation of the effectiveness on a given trace $t$. In order to apply this method we propose the following algorithm. The function $\text{preamble}_{\text{ok}}(c,n)$ tells us if the preamble part of the invariant $c$ is well in adequacy with the portion of trace pointed by $n$. If so, the test corresponding to the preamble will be applied to the trace, if not nothing is done.

Let $IO$ be the set of output invariants, $ISucc$ be the set of succession invariants and $II$ be the set of input invariants.

Algorithm 2. Test length resulting from invariants
Input: sets of invariants $IO$, $ISucc$, $II$
Output: the test resulting from output, succession and input invariants

Table 5
Non observable nondeterminism on the invariants of length 1

<table>
<thead>
<tr>
<th>Type</th>
<th>Specification</th>
<th>Input Invariants found</th>
<th>Output Invariants found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nondeterministic</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
begin
 nb_test := 0;
 /* Tests resulting from output invariants and succession invariants */
 for each c in IO \ ISucc do
  n := 1;
  while (n ≤ l(c) + 1) do
   if (preambleOk(c,n)) {
    nb_test ++;
   }
  end
  n++;
 end
 /* Tests resulting from input invariants */
 for each c in II do
  n := l(c);
  while (n ≥ l(c)) do
   if (preambleOk(c,n)) {
    nb_test ++;
   }
  end
 end
 return nb_test;
end

3.3. The second approach: constraint application

3.3.1. Principle

The idea of this method is to express properties resulting from the EFSM in a formalism close to the first order logic. Thus, a network service or protocol can be described using a set of properties expressed in this new formalism.

A specification EFSM will give us a set \( Q \) of properties \( \theta \). A property will consist of two parts:

- a \textit{preamble} part containing an input/output sequence,
a constraint part containing a logical formula relating to several elements such as the current state, the next state, the predicates, the actions, and the definition fields of variables.

The trace resulting from the implementation is, first, put in the shape of an automaton $Aut$. Then we apply to this automaton an algorithm for marking the states to see whether it checks the various constraints resulting from the specification. In fact, we used as a starting point a marking algorithm for model checking. The summary diagram is presented in Fig. 2.

Then, we will see how to extract the properties stemming from the specification and how to transform the trace into an automaton. To finish, we will develop the algorithm used for marking the states.

### 3.3.2. Definition of the logic used

These are the various connectors and predicates that we will use. Connectors resulting from the propositional logic are $\land$, $\lor$, not ($\lnot$), imply ($\rightarrow$), the predicates are equality ($=$), belonging to a set ($\in$). $I(e)$ is true if $e$ is an input message, $O(o)$ is true if $o$ is an output message. $CS(s)$ is true if $s$ is the current state, $NS(s)$ is true if $s$ is the next state.

### 3.3.3. Extraction of the properties

The extraction of the properties stemming from the specification follows the next steps. For each transition $t_j$ from the EFSM we create a new property $\theta_j$ such that:

- the preamble of $\theta_j$ contains the following formula: $\text{preamble} = I(input(t_j)) \land O(output(t_j))$
- the constraint part of $\theta_j$ contains the following formula: $\text{constraint} = CS(current\_state(t_j)) \land NS(next\_state(t_j)) \land predicate(t_j) \land action(t_j) \land domains\_variables$,

where $input(t_j)$ is the parameterised (or not) input message of $t_j$, $output(t_j)$ is the parameterised (or not) output message of $t_j$, $current\_state(t_j)$ is the starting state of the transition $t_j$, $next\_state(t_j)$ is the final state of the transition $t_j$, $predicate(t_j)$ is the formula used as predicate to $t_j$, $action(t_j)$ is the set of the actions of $t_j$ on the EFSM's variables, $domains\_variables$ is a formula gathering the fields of definition of each variable used like parameter of a message in $t_j$ in a predicate or in an action.

**Example 7.** Consider the transition $t_1$ presented in Fig. 3. The values of $CONreq(qos)$ and $ReqQos$ belong to an interval $[0,3]$.

![Fig. 3. An example of a transition.](image)

The property $\theta_1$ of this transition is the following.

$$
\theta_1 = \begin{cases} 
I(\text{CONreq}(qos)) \\
\land O(\text{connect}(\text{ReqQos})) \\
\land CS(s_0) \\
\land NS(s_1) \\
\land (\text{CONreq.qos} \leq 1) \\
\land (\text{ReqQos} = \text{CONreq.qos}) \\
\land (\text{ReqQos} \in [0,3]) \\
\land (\text{ReqQos} \in [0,3]) 
\end{cases}
$$

### 3.3.4. Transformation of the trace into an automaton

Now, it remains to find a notation for the trace resulting from the implementation. We will use an automaton $Aut$ defined in the following way.

$Aut = (S, \rightarrow, L)$, where

- $S$ a finite set of states,
- $\rightarrow \subseteq S \times S$ such that $\forall s \in S \: \exists s' \in S \: (s, s') \in \rightarrow$,
- $L : S \rightarrow P$ such that $L$ associates with each state $s \in S$ the set of formulas, which are true in this state.

**Example 8.** Consider the SCP specification presented in Fig. 6 (Section 4.1). The formats of messages are: $CONreq(qos)$ with $qos \in [0,3]$, $\text{connect}(\text{ReqQos})$ with $\text{ReqQos} \in [0,3]$, $\text{accept}(qos)$ with $qos \in [0,3]$, $\text{CONcnf} (+, \text{FinQos})$ with $\text{FinQos} \in [0,3]$ and $\text{data}(\text{FinQos})$ with $\text{FinQos} \in [0,3]$. The trace consisting of four transitions has
the following form.

\[
\text{Trace} = \begin{cases} 
\text{CONreq}(3)/\text{NONsupport}(3), \\
\text{CONreq}(1)/\text{connect}(1), \\
\text{accept}(2)/\text{CONcnf}(+, 1), \\
\text{Data}/\text{data}(1)
\end{cases}
\]

The automaton corresponding to this trace is schematically presented in Fig. 4.

### 3.3.5. The marking algorithm

Now, we discuss how to extract the properties from the specification and to create the automaton corresponding to the trace. Thus, it remains to develop the procedure of marking of automaton states.

Initially, the automaton representing the trace contains several states \( s_i \) for \( i = 1, \ldots, l_t \), where \( l_t \) is the length of the trace. As we do not know the state in which the implementation is at the beginning of the observation, we are not able to conclude what state of specification corresponds to each state \( s_i \) of the automaton. We have to determine it during the detection of errors.

The procedure of marking of automaton states consists of the following steps.

1. To mark each state of automaton \( \text{Aut} \) which validates the preamble of the properties \( \theta_i \). If a state of the automaton does not validate the preamble of any property we detected an error and we stop the process. Else, we pass to the following step.
2. To replace preceding markings by the constraint part associated with each property \( \theta_i \).

3. To check if the predicates, actions and fields of definition presented in the constraints respect the information relating to each state of the automaton. If it is the case, we can pass to the following phase, else, we detected an error and announce it.
4. To keep information concerning the current state (\( CS \)) and the next state (\( NS \)) in a specific formula and to check a coherence between all the formulas of this nature.
5. If, finally, the constraints are not satisfied or there is a contradiction between them, we conclude that the implementation is erroneous. Else, we cannot establish a conclusion on the conformity of the implementation since another trace can reveal errors.

### 4. Preliminary experiments

This section deals with the application of the two approaches presented in Section 3 to the specification of \( \text{SCP} \). The protocol is described in detail in Section 4.2. Section 4.3 is devoted to the extraction of the properties and the invariants relating to the specification of \( \text{SCP} \). Section 4.4 introduces an implementation containing an output error and unrolls the two approaches to see whether they detect this error. And, finally, Section 4.5 presents experiments with the same implementation containing a transfer error.

#### 4.1. Description of the specification used

The Simple Connection Protocol (or \( \text{SCP} \)) allows us to connect an entity called the upper layer to the entity called the lower layer (Fig. 5). The upper layer performs a dialogue with \( \text{SCP} \) to fix the quality of service desirable for the future connection. Once this negotiation finished, \( \text{SCP} \) 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 transmit 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 again wishes to be connected to the lower layer, it is necessary to restart a connection from the beginning (i.e. the negotiation of the quality of service with SCP).

Fig. 6 presents the EFSM associated with SCP. TryCount, ReqQos, FinQos, CONreq-qos and accept.qos belong to the interval [0,3].

4.2. Invariant extraction

Here we present the invariants extracted from SCP. The test part is underlined.

**Output invariants of length 1**

**IS1-1:** accept(qos)/CONcnf($\pm$, FinQos)
**IS1-2:** Data/data(FinQos)
**IS1-3:** Reset/abort

**Output invariants of length 2**

**IS2-1:** CONreq(qos)/connect(ReqQos), accept(qos)/CONcnf($\pm$, FinQos)
**IS2-2:** refuse/connect(ReqQos), accept(qos)/CONcnf($\pm$, FinQos)
**IS2-3:** accept(qos)/CONcnf($\pm$, FinQos), Data/data(FinQos)
**IS2-4:** accept(qos)/CONcnf($\pm$, FinQos), Reset/abort
**IS2-5:** Data/data(FinQos), Data/data(FinQos)
**IS2-6:** Data/data(FinQos), Reset/abort

**Input invariants of length 1**

**IE1-1:** CONreq(qos)/NONsupport(ReqQos)
**IE1-2:** refuse/CONcnf(\(-\))
**IE1-3:** accept(qos)/CONcnf($\pm$, FinQos)
**IE1-4:** Data/data(FinQos)
**IE1-5:** abort/Reset

**Input invariants of length 2**

**IE2-1:** CONreq(qos)/NONsupport(ReqQos), CONreq(qos)/CONsupport(ReqQos)
**IE2-2:** CONreq(qos)/NONsupport(ReqQos), CONreq(qos)/connect(ReqQos)
**IE2-3:** accept(qos)/CONcnf($\pm$, FinQos), Data/data(FinQos)
**IE2-4:** Data/data(FinQos), Data/data(FinQos)
**IE2-5:** accept(qos)/CONcnf($\pm$, FinQos), Reset/abort
**IE2-6:** Data/data(FinQos), Reset/abort

**Succession invariants**

**ISucc-1:** CONreq(qos)/connect(ReqQos), refuse/connect(ReqQos)

---

Fig. 6. The EFSM associated with the Simple Connection Protocol.
4.3. Properties extraction

The properties relating to the EFSM are the following:

\begin{align*}
\theta_1 &= \{ P : I(-) \land O(-) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{TryCount} = 0) \land (\text{ReqQos} = 0) \land (\text{FinQos} = 0) \} \\
\theta_2 &= \{ P : (\text{CONreq(qos)}) \land O(\text{NONsupport(QQos)}) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{CONreq(qos)} > 1) \land (\text{ReqQos} = \text{CONreq(qos)} \land \text{CONreq(qos)} \in [0, 3] \land \text{ReqQos} \in [0, 3]) \} \\
\theta_3 &= \{ P : I(\text{CONreq(qos)}) \land O(\text{connect(QQos)}) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{CONreq(qos)} \leq 1) \land (\text{ReqQos} = \text{CONreq(qos)} \land \text{CONreq(qos)} \in [0, 3] \land \text{ReqQos} \in [0, 3]) \} \\
\theta_4 &= \{ P : I(\text{refuse}) \land O(\text{connect(QQos)}) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{TryCount} = 2) \land (\text{TryCount} = \text{TryCount} + 1) \land (\text{TryCount} \in [0, 3] \land (\text{ReqQos} \in [0, 3]) \} \\
\theta_5 &= \{ P : I(\text{refuse}) \land O(\text{CONcnf}(-)) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{TryCount} = 2) \land (\text{TryCount} \in [0, 3]) \} \\
\theta_6 &= \{ P : I(\text{accept(qos)}) \land O(\text{CONcnf}(+, \text{FinQos})) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{FinQos} = \text{min}(\text{accept(qos)}, \text{ReqQos})) \land \text{accept(qos)} \in [0, 3] \land \text{FinQos} \in [0, 3] \land (\text{ReqQos} \in [0, 3]) \} \\
\theta_7 &= \{ P : I(\text{Data}) \land O(\text{data(FinQos)}) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{FinQos} \in [0, 3]) \} \\
\theta_8 &= \{ P : I(\text{Reset}) \land O(\text{abort}) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{FinQos} \in [0, 3]) \} \\
\theta_9 &= \{ P : I(\text{Data}) \land O(\text{data(FinQos)}) \} \\
&\quad \{ C : CS(s_1) \land NS(s_2) \land (\text{FinQos} \in [0, 3]) \} \\
\end{align*}

4.4. Example of an output error

The implementation of SCP presented overleaf (Fig. 7) contains an error in the transition drawn using dotted lines. Indeed, the field action of this transition makes the assignment ‘ReqQos’:= CONreq(qos) − 1’ instead of the assignment ‘ReqQos’:= CONreq(qos)’ (action defined in the SCP specification). This modification induces an output error since the transmitted message has an erroneous parameter. Indeed, by receiving the input message CONreq(2), a correct implementation must transmit the output message NONSupport(2). However, the implementation described below transmits NONSupport(1). Thus, there is an output error since the output message emitted by the implementation is not that expected by the specification. We will show that one can apply the proposed methods to detect this output fault.

The trace resulting from the implementation has the following form.

Trace = CONreq(3)/NONSupport(2),
CONreq(1)/connect(1),
accept(2)/CONcnf(+, 1),
Data/data(1)

Method 1: invariant application

In order to implement this method we start by trying to match several invariants to the trace. First, we look for common patterns shared by the trace and the preamble part of each invariant. Table 7 gathers the different invariants allowing us to test the trace. The test part of each invariant is underlined.

We also need to check that the variables validate their definition fields. In our case, the variables have correct definition fields.

- CONreq(qos) = 3, CONreq(qos) \in [0, 3]
- ReqQos = 2, ReqQos \in [0, 3]
- accept.qos = 2, accept.qos \in [0, 3]
- FinQos = 1, FinQos \in [0, 3]
According to the formula allowing to predict the number of tests launched by a set of invariants, we wait for 8 or 9 tests.

\[
\sum_{c \in 2} l_c - l(c) + 1 = 5 \times \frac{8 - 1 + 1}{5^1 \times 6^0} + 4 \times \frac{8 - 2 + 1}{5^1 \times 6^1} = 8,2
\]

Outcome:

- nine tests are carried out (we check the preceding probability).
- five input/output symbols tested on 8, giving a 62.5% coverage.
- there is no problem concerning the definition fields.
- there is no detected error, we have to apply our second method.

Method 2: constraint application

We now apply the second method. Following we provide some explanations for the different steps implemented.

- T1 = Test 1: Selection of the properties \( \theta_i \) for which the preamble is validated by the states of the automaton. If a state of the automaton does not validate the preamble of any property, then an error has been detected, we stop the process and announce an error. On the other hand, if each state of the automaton validates the preamble to one or more properties, we pass to the following step: the constraint application.
- C1 = Constraint 1: Extraction of the constraints relating to each property \( \theta_i \) selected during Test 1. This step is made up of two parts. The first one contains all the constraints relating to the predicates, the actions and the variable fields of definition. The second part contains the constraints on the identification of states.

Fig. 8 presents an automaton corresponding to the chosen trace and in the FSM is presented in Table 8.

<table>
<thead>
<tr>
<th>Trace</th>
<th>CONreq(3)</th>
<th>NONsupport(2)</th>
<th>CONreq(1)</th>
<th>connect(1)</th>
<th>accept(2)</th>
<th>CONcnf(+,1)</th>
<th>Data</th>
<th>data(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE1-1</td>
<td>CONreq(qos)</td>
<td>NONsupport(ReqQos)</td>
<td>CONreq(qos)</td>
<td>connect(ReqQos)</td>
<td>accept(qos)</td>
<td>CONcnf(+,FinQos)</td>
<td>Data</td>
<td>data(FinQos)</td>
</tr>
<tr>
<td>IE1-2</td>
<td>CONreq(qos)</td>
<td>NONsupport(ReqQos)</td>
<td>CONreq(qos)</td>
<td>connect(ReqQos)</td>
<td>accept(qos)</td>
<td>CONcnf(+,FinQos)</td>
<td>Data</td>
<td>data(FinQos)</td>
</tr>
<tr>
<td>IE1-3</td>
<td>CONreq(qos)</td>
<td>NONsupport(ReqQos)</td>
<td>CONreq(qos)</td>
<td>connect(ReqQos)</td>
<td>accept(qos)</td>
<td>CONcnf(+,FinQos)</td>
<td>Data</td>
<td>data(FinQos)</td>
</tr>
<tr>
<td>IS2-2</td>
<td>CONreq(qos)</td>
<td>NONsupport(ReqQos)</td>
<td>CONreq(qos)</td>
<td>connect(ReqQos)</td>
<td>accept(qos)</td>
<td>CONcnf(+,FinQos)</td>
<td>Data</td>
<td>data(FinQos)</td>
</tr>
</tbody>
</table>

Fig. 7. The erroneous implementation with an output error.
The constraint application enabled us to detect the output error presented in the implementation. We know that it is an output error since it is a constraint relating to the parameter values that revealed it.

4.5. Example of a transfer error

The SCP implementation presented in Fig. 9 contains a transfer error in the transition drawn using dotted lines. Indeed, the predicate field of this transition tests that ‘TryCount = 0’ instead of testing that ‘TryCount = 2’ (a predicate defined in the SCP specification). This modification induces a transfer error since the implementation moves to the state $s_1$ while the specification forces the system to remain in the state $s_3$. Indeed, when receiving the refuse message for the first intent, the SCP envisages to remain in the state $s_3$ to make a new attempt. However, by moving to the state $s_1$,
the implementation definitively rejects the upper layer at the first attempt.

Thus, we are in the presence of a transfer error. We will show that to detect it we can apply the proposed test methods.

Consider a trace corresponding to a wrong transition.

\[
\text{Trace} = \begin{cases} 
\text{CONreq}(1)/\text{connect}(1), \\
\text{refuse}/\text{CONcnf}(-) 
\end{cases}
\]

**Method 1: invariant application**

As in the preceding example, Table 9 gathers the various invariants testing the trace.

<table>
<thead>
<tr>
<th>Trace</th>
<th>CONreq(1)</th>
<th>connect(1)</th>
<th>Refuse</th>
<th>CONcnf(−)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISucc1</td>
<td>CONreq(qos)</td>
<td>Connect(ReqQos)</td>
<td>Refuse</td>
<td>connect(ReqQos)</td>
</tr>
<tr>
<td>EI1-2</td>
<td>ReqQos = 1</td>
<td>ReqQos ∈ [0, 3]</td>
<td>TryCount = 2</td>
<td>TryCount = TryCount + 1</td>
</tr>
</tbody>
</table>

It is useless to check the variables definition fields since an error was already detected by the invariant of succession ISucc1.

**Outcome:**

- Two tests are carried out.
- Two input/output symbols tested on 4, giving 50% coverage.
- One error detected due to the succession invariant ISucc1.

**Method 2: constraint application**

The automaton associated with the trace is presented in Fig. 10, and Table 10 recapitulates the various phases of the constraint application.

The second method did not enable us to detect the transfer error contained in the implementation, while the invariant application method enabled us to update the error presented in this implementation.

### 5. Conclusion

The use of passive testing resulted in several research efforts, majority of which were focused on the simple FSM model. However, the actors of the telecommunications world develop state-of-the-art services and they must describe formal specifications which use variables, predicates and high level instructions. Thus, they do not have the possibility of using the FSM formalism and cannot always benefit from the studies carried out on the passive testing of FSM. This is why a study was launched by us to develop new methods for the passive testing of EFSMs.

It is within the framework of this study that we developed two new approaches. The first method looks for input/output invariants in a specification and checks that the trace resulting from the implementation is coherent with these invariants. The second method transforms the specification into a set of constraints and controls that the trace respects them.

Other approaches had already been proposed and the most advanced try to determine the variable values used by the specification. We wanted to compare the effectiveness of our methods with these types of approach to show the contributions brought by our work. Results proved to be positive since the experiments showed that certain errors detected by our methods were undetected by others (such as Ref. [2]).

This work has many prospects. Indeed, it would be interesting to improve the input/output invariant technique to obtain a better test quality of the data flow. For the moment, this method checks the variable definition fields, but it would be interesting to go beyond this simple control since many errors can still pass undetected. In addition, we saw that the nondeterminism could have a harmful influence on the input/output invariant extraction. And although we often try to write deterministic specifications, it is not always easy to obtain them. Thus, it will be necessary to adapt the invariant method to limit this bad influence. An idea could be to introduce the concept of fairness into the specification.

It would also be wise to adapt our techniques to be able to use them on SDL specifications since it is currently the standard language for the formal description of network protocols and services. Thus, many research direction remain open to explore in the field of passive testing of EFSMs.

### Acknowledgements

We would like to thank David Lee (Bell Labs) for his comments and discussion of results of this paper.
References


