

Fast Testing of Critical Properties through Passive Testing*

José Antonio Arnedo,¹ Ana Cavalli¹ and Manuel Núñez²

¹ Institut National des Télécommunications GET-INT
91011 Evry Cedex, France

{Jose-Antonio.Arnedo-Rodriguez, Ana.Cavalli}@int-evry.fr

² Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, E-28040 Madrid, Spain.
mn@sip.ucm.es

Abstract. We present a novel methodology to perform *passive testing*. The usual approach consists in recording the trace produced by the implementation under test and trying to find a fault by comparing this trace with the specification. We propose a more *active* approach to passive testing where the minimum set of (critical) properties required to a correct implementation may be explicitly indicated. In short, an invariant expresses that each time that the implementation under test performs a given sequence of input/output actions, then it must show a behavior reflected in the invariant. By using an adaptation of the classical pattern matching algorithms on strings, we obtain that the complexity of checking whether an invariant is fulfilled by the recorded trace is in $\mathcal{O}(n \cdot m)$, where n and m are the lengths of the trace and the invariant, respectively. If the length of the invariant is much smaller than the length of the trace, then we have that this complexity is *almost lineal* with respect to the length of the trace. Actually, this is usually the case for most practical examples. In addition to our methodology, we present the case study that was the driving force for the development of our theory: The Wireless Application Protocol (WAP). We present a test architecture for WAP as well as the experimental results obtained from the application of our passive testing with invariants approach.

1 Introduction

The main purpose of testing is to find out whether an implementation presents the behavior that was indicated by the corresponding specification. In order to perform this task, several techniques, algorithms, and semantic frameworks have been introduced in the literature (see e.g. [LY96,Lai02] for two overviews on the topic). Most of the proposals for testing are based on the so-called *active* testing. Intuitively, the tester sends an input to the implementation and waits for an output. If the output belongs to the expected ones, according to the specification, then the process continues; otherwise, a fault has been detected in the implementation. This kind of testing is called active because the tester has total control over the inputs provided to the implementation under test. On the contrary, *passive testing* (see e.g. [LNS⁺97,Mil98,TC99,TCI99]) does not involve

* Research supported in part by the CICYT project TIC 2000-0701-C02-01. This research was carried out while the third author was visiting the GET-INT.

the presence of an *active* tester. In passive testing the implementation under test is allowed to run independently without any interaction with a tester. However, the trace that the implementation is executing is observed so that it can be analyzed. By comparing the obtained trace with the specification we may detect some faults in the implementation.

Unfortunately, passive testing is less powerful than active testing. This is so because active testing allows a closer control of the implementation under test. For example, depending on the received output, we may choose among a set of inputs to be applied to the implementation. In passive testing this capability is lost. Thus, it can happen that faults that could be detected by choosing an appropriate input are not found because the implementation does not take that path. Nevertheless, passive testing presents some important advantages. First, active testing is in general more costly because the testing process has to be closely controlled. Second, there are situations where active testing is not even feasible. For instance, passive testing is attracting a lot of study in the field of network management where the testing process has to be minimized in order to reduce the use of the network (see [MA01b,MA01a,LCH⁺02] for some recent works on the topic).

Even though there is ongoing work on passive testing³ most of these proposals share a common pattern. Usually, the trace is taken and the specification is traversed in order to detect a fault. A drawback of this approach is that it presents a low performance (in terms of complexity in the worst case) if non-deterministic specifications are considered. A notable exception is presented in [CGP01] where the specification is (partially) set apart. Actually, they extract some test sequences from the specification, called *invariants*, and they check whether the trace obtained from the implementation under test is correct with respect to them. So, once these test are extracted, the specification plays no role in the testing process. They consider three kinds of invariants. However, the most useful for practical purposes are the so-called *output* invariants. Intuitively, an (output) invariant as $i_1/o_1, \dots, i_{n-1}/o_{n-1}, i_n/o_n$ must be interpreted as “*each time the implementation performs the sequence $i_1/o_1, \dots, i_{n-1}/o_{n-1}, i_n$ the next observed output must be o_n* ”.

This paper extends and improves [CGP01] in several ways. Actually, our methodology represents a real example of theory guided by practice. In fact, we came out with our different notions of invariants when we tried to apply [CGP01] to a real protocol. By doing so we found some shortcomings. First, their invariants were not expressive enough for our purposes. For example, the complete sequence has to be indicated. That is, properties as

Each time that a user asks for connection and the connection is granted, if after performing some operations the user asks for disconnection then he is disconnected.

cannot be easily represented by using their invariants, because all the possible sequences of actions expressing the idea of *some operations* must be explicitly written. So, we have added the possibility of specifying wild-card characters in invariants. Besides, we now allow a set of outputs (instead of a single output) as termination of the invariant. Thus, we may specify properties as

³ Nevertheless, the effort spent in passive testing is not yet comparable with the one for the study of active testing techniques.

Each time that a user asks for a resource (e.g. a web page) either the resource is obtained or an error is produced.

In addition, we found other kind of properties that could not be expressed at all. For example,

A user cannot get disconnected if he was not previously connected.

Finally, they extracted invariants from the specification. This approach presents two drawbacks. First, *interesting* invariants cannot be distinguished from *trailing* invariants. Second, and more important, the complexity of extracting invariants exponentially increases with their length. That is, the complexity of extracting the invariants of length n is in $\mathcal{O}(|Tr|^n)$, where $|Tr|$ is equal to the number of transitions in the specification. We claim that invariants should be supplied by the specifier/tester. In this case, the first step must be to check that the invariant is in fact correct with respect to the specification. We provide an algorithm that checks this correctness in linear time, with respect to the number of transitions, if the invariant does not contain the wild-card character $*$; this complexity is quadratic if the symbol $*$ appears in the invariant.

Once we have a set of (correct) invariants, our approach to passive testing proceeds as follows: We observe the trace produced by the implementation under test and we decide whether this trace respects the invariants. In order to do so, we have implemented a simple variant of the classical algorithms for pattern matching on strings (see e.g. [BM77,KMP77]). Our algorithm works, in the worst case, in time $\mathcal{O}(m \cdot n)$, where n and m are the length of the trace and the length of the invariant, respectively. Let us remark that in most practical cases the length of the invariant is several orders of magnitude smaller than the length of the trace. Thus, we may consider that the complexity is almost lineal with respect to the length of the trace.

In addition to the formal framework, in this paper we also report our experiments on the WAP (Wireless Application Protocol). This protocol is an open global specification that empowers mobile users with wireless devices to easily access and interact with information and services instantly. It is worth to point out that this protocol represents a typical example where active testing cannot be applied. In general, there is not direct access to the interfaces between the different layers. Thus, the tester cannot control how internal communications are established. However, in our experiments we have used a software free protocol stack, namely Kannel, and we have the possibility of installing points of observation, in short POs, between the different layers. Moreover, a platform and a test architecture capable to deal with passive testing in a mobile phone environment (WAP, GPRS, UMTS) have been defined. The platform and the architecture are used to apply our passive testing with invariants approach. We take the output of the platform, in the form of log files, and we apply the appropriate invariants to the obtained trace. This experiment represents an original contribution because such a study has been never performed in a systematic way.

The rest of the paper is organized as follows. In Section 2 we present our notions of invariants. First, we introduce what we call *simple invariants*. These invariants are able to express properties as “after x has happened then we must have that y happens”. Next, we introduce a new notion of invariant to express properties as “if y happens then we must have that x had happened before”. We call these invariants *obligation invariants*. We give algorithms to decide the

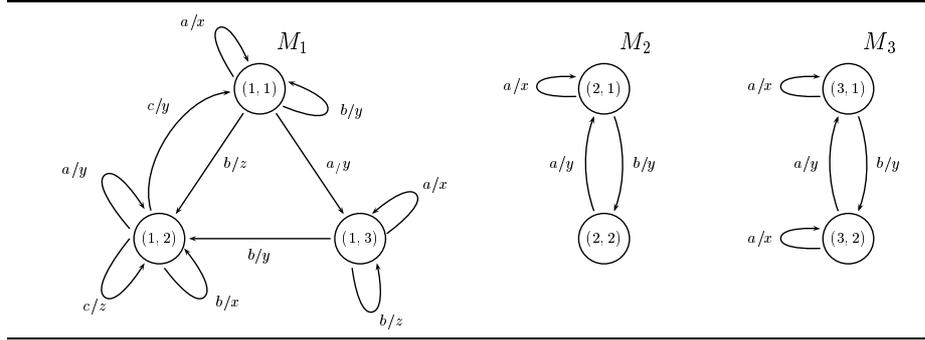


Fig. 1. Examples of FSMs.

correctness of our invariants with respect to a given specification and we explain how our invariants are *applied* to the observed trace. In Section 3 we present the WAP and we briefly comment on the performed experiments by using our notion of passive testing. Finally, in Section 4 we give our conclusions and some lines for future work.

2 Invariants and Passive Testing

In this section we introduce our invariants and the corresponding algorithms to decide whether they are correct with respect to specifications. We consider that specifications are represented as Finite State Machines. However, we will comment on how our invariants can be extended to deal with data.

Definition 1. A *Finite State Machine*, in the following **FSM**, is a tuple $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ where S is a finite set of states, \mathcal{I} is the set of input actions, \mathcal{O} is the set of output actions, Tr is the set of transitions, and s_{in} is the initial state.

Each transition $t \in Tr$ is a tuple $t = (s, s', i, o)$ where $s, s' \in S$ are the initial and final states of the transition, respectively, and $i \in \mathcal{I}$, $o \in \mathcal{O}$ are the input and output actions, respectively.

Let $s, s' \in S$ be states and $tr = i_1/o_1, \dots, i_n/o_n$, for $n \geq 1$, be a sequence of pairs such that for any $1 \leq j \leq n$ we have $i_j \in \mathcal{I}$ and $o_j \in \mathcal{O}$. We write $s \xrightarrow{tr} s'$ if either $tr = \epsilon$ and $s = s'$ or there exist n transitions $t_1, \dots, t_n \in Tr$ and states $s_1, \dots, s_{n-1} \in S$ 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 $t_j = (s_{j-1}, s_j, i_j, o_j)$. \square

First, let us note that this notion of **FSM** does not restrict specifications to be deterministic, so that we work with a general notion of **FSM**. Intuitively, a transition $t = (s, s', i, o)$ indicates that if the machine is in state s and receives the input i then the machine emits the output o and the current state becomes s' . We will sometimes write $s \xrightarrow{i/o} s'$ to denote that we have the transition $(s, s', i, o) \in Tr$. We can extend the notion of single transition to a sequence of transitions. Thus, $s \xrightarrow{tr} s'$ simply denotes that we can traverse from the state s to the state s' by following transitions containing the corresponding pairs i/o appearing in tr .

We may extend the notion of transition to deal with the wild-card characters ? and *. In particular, we can consider that sequences may contain these special symbols.

Definition 2. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM. We write $s \xrightarrow{?/o} s'$ (resp. $s \xrightarrow{i/?} s'$) if there exists $i \in \mathcal{I}$ (resp. $o \in \mathcal{O}$) such that $s \xrightarrow{i/o} s'$. Besides, we write $s \xrightarrow{?/?} s'$ if there exist $i \in \mathcal{I}$ and $o \in \mathcal{O}$ such that $s \xrightarrow{i/o} s'$. We write $s \xrightarrow{*} s'$ if there exists a sequence of input/output pairs tr such that $s \xrightarrow{tr} s'$. \square

2.1 A First Notion of Invariants

Next we present our first notion of invariant. We call them *simple invariants*, or just *invariants*. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM. Intuitively, a trace $i_1/o_1, \dots, i_{n-1}/o_{n-1}, i_n/O$ is a simple invariant for M if each time that the trace $i_1/o_1, \dots, i_{n-1}/o_{n-1}$ is observed, if we obtain the input i_n then we necessarily get an output belonging to O , where $O \subseteq \mathcal{O}$. In addition to sequences of input/output symbols, we will allow the *wildcard* characters ? and *. The meaning of ? in our framework is the standard one in the pattern matching community (that is, to replace any symbol). However, we will slightly modify the usual meaning of *. For example, the intuitive meaning of an invariant as $i/o, *, i'/O$ is that if we detect the transition i/o then the first occurrence of the input symbol i' is followed by an output belonging to the set O . In other words, * replaces any sequence of symbols not containing the input symbol i' .

Definition 3. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM. We say that the sequence tr is a (simple) *invariant* for M if the following two conditions hold:

1. tr is defined according to the EBNF $tr ::= i/O | *, tr | a/z, tr$. In the previous expression we consider $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $O \subseteq \mathcal{O}$.
2. tr is *correct* with respect to M .

We denote the set of (simple) invariants for M by Inv_M . \square

In Figures 2 and 3 we introduce two algorithms to decide whether an invariant is *correct* with respect to a specification. First, we present some examples of invariants to show what kind of critical properties can be tested as well as how our invariants work.

Example 1. Our notion of invariant allows us to express several interesting properties. For example, we can test that when a user requests a disconnection then he is in fact disconnected by using the invariant

$$I_1 = req_disconnect / \{disconnected\}$$

The idea is that each time that the symbol *req_disconnect* appears in the trace then it is followed by the output symbol *disconnected*. For instance, this invariant has the same distinguishing power as the invariant

$$I'_1 = *, req_disconnect / \{disconnected\}$$

We can specify a more complex property by taking into account that we are interested in disconnections only if a connection was requested. In this case we have

$$I_2 = req_connect/? , *, req_disconnect/\{disconnected\}$$

We can refine the previous invariant if we only consider the cases where the connection was granted

$$I_3 = req_connect/granted_connection, *, req_disconnect/\{disconnected\}$$

For example, a trace is correct with respect to I_3 if each time that we find a (sub)sequence starting with the pair *req_connect/granted_connection* then the first occurrence of the input symbol *req_disconnect* is paired with the output symbol *granted_connection*. Let us remark that if the trace contains the pair *req_connect/granted_connection* but the input *req_disconnect* does not appear before the end of the recorded trace then we cannot conclude that the implementation fails: It may happen that we have stopped *too soon* observing the behavior of the implementation.

Finally, an invariant as

$$I_4 = req_connect/\{granted_connection, error\}$$

indicates that after requesting a connection we either are granted with it or an error is produced. \square

We could adapt to our framework the algorithm given in [CGP01] to extract their invariants, up to a length n , for a specification M . However, as we explained in the introduction of the paper, this process presents several drawbacks. In particular, the complexity exponentially increases with the length of invariants. On the contrary, we advocate that invariants should be indicated by the specifier/tester as the set of critical properties that the implementation must fulfill. Fortunately, we have found an algorithm that detects in linear time (with respect to the number of transitions in the FSM) whether a sequence of symbols not containing the character $*$ is in fact an invariant for a specification. Obviously, before we try to find out whether the trace recorded from the implementation is *correct* with respect to an invariant, we should assure that the invariant is in fact *correct* with respect to the specification. In order to facilitate the reading, we first present an algorithm (see Figure 2) deciding correctness of invariants without the $*$ wild-card character. In Figure 3 we extend this algorithm to deal with invariants containing the symbol $*$.

The algorithm given in Figure 2 works as follows. The first *while-loop* computes those states $s \in S$ such that they can be reached from one of the states in S by following the sequence $i_1/o_1, \dots, i_{n-1}/o_{n-1}$. Let us remark that if one of the symbols in the sequence is the wild-card character $?$ then any symbol can be used. Besides, it may happen that after some steps we find that there do not exist two states connected by the analyzed sub-sequence. In this case, S' becomes empty. Let us note that for each execution of the loop we perform a number of operations proportional to the number of transitions in the corresponding specification. So, in the worst case we perform a number of operations proportional to the number of transitions times the length of the sequence, that is, n . The second *while-loop* analyzes the last pair of the invariant. If the auxiliar

Input: $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$, $I = i_1/o_1, \dots, i_n/o_n, \forall 1 \leq j \leq n : i_j \neq * \wedge o_j \neq *$
Ouput: true/false \equiv invariant I correct/incorrect

```

j := 1; S' := S;
while j < n and S' ≠ ∅ do begin
  T := Tr; S'' := ∅;
  while T ≠ ∅ do begin
    choose t ∈ T; {t = (s, s', a, z)};
    T := T - {t}
    if s ∈ S' and a = i_j and z = o_j then S'' := S'' ∪ {s'}
  end;
  S' := S''; j := j + 1
end;
{last pair of the invariant or empty set S' of current states}
if S' = ∅ then return(false)
else begin
  error := false; T := Tr;
  while T ≠ ∅ and not error do begin
    choose t ∈ T; {t = (s, s', a, z)};
    T := T - {t};
    if s ∈ S' and a = i_n and z ∉ O then error := true
  end;
  return (not error)
end
end

```

We consider that both $i = ?$ and $o = ?$ hold.

Fig. 2. Checking correctness of Invariants (1/2).

set of states S' is empty then the invariant is incorrect. Actually, this means that there does not exist a state in the specification such that the sequence of pairs forming the invariant can be performed from it. Thus, we should not consider that this *candidate* represents any property of the specification.⁴ If that set is not empty then we check that for any transition labelled by the input i_n we receive an output belonging to O . Again, the complexity of this last loop is given by the number of transitions. Besides, we need $|S| + |Tr|$ additional space. Let us note that if the graph induced by the corresponding FSM is connected then $|S| \leq |Tr| + 1$ (otherwise we can discard those states and transitions not reachable from the initial state and the result holds for the new sets of states and transitions).

Proposition 1. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM and $I = i_1/o_1, \dots, i_n/o_n$ be an invariant such that for any $1 \leq j \leq n$ we have $i_j \neq * \wedge o_j \neq *$. The worst case of the algorithm given in Figure 2 checks the correctness of the invariant I with respect to M in time $\mathcal{O}(n \cdot |Tr|)$ and space $\mathcal{O}(|Tr|)$. \square

Next, we present some examples of correct/incorrect invariants for a given specification.

⁴ Another possibility would be to consider the usual meaning of the logical implication. Thus, the predicate “each time that the prefix is performed then *something* happens” would hold since the premise is false.

Input: $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$, $I = i_1/o_1, \dots, i_n/o$
Output: true/false \equiv invariant I correct/incorrect

```

 $I' := I; S' := S;$ 
while  $I' \neq b/O$  and  $S' \neq \emptyset$  do begin
   $first := \text{head}(I'); I' := \text{tail}(I');$ 
  if  $first \neq *$  then begin  $\{first := i/o\}$ 
     $T := Tr; S'' := \emptyset;$ 
    while  $T \neq \emptyset$  do begin
      choose  $t \in T; \{t = (s, s', a, z)\};$ 
       $T := T - \{t\};$ 
      if  $s \in S'$  and  $i = a$  and  $o = z$  then  $S'' := S'' \cup \{s'\}$ 
    end;
     $S' := S''$ 
  end
else begin  $\{first = *\}$ 
  while  $\text{head}(I') = *$  do  $I' := \text{tail}(I'); \{\text{skip a sequence of } *'s\}$ 
   $first := \text{head}(I'); \{first := i/o\}$ 
   $S' := \{s \in S \mid \exists s' \in S' : \text{path}(s', s, i)\}$ 
end
end;
 $\{\text{last pair of the invariant or empty set } S' \text{ of current states}\}$ 
See Algorithm in Figure 2 for dealing with the last pair of the invariant.

```

We consider that both $i = ?$ and $o = ?$ hold.

Fig. 3. Checking correctness of Invariants (2/2).

Example 2. Let us consider the FSMs presented in Figure 1. For example, the following invariants are correct for M_1 :

$$I_1 = a/\{x, y\} \quad I_2 = a/?, c/z, b/\{x\}$$

Let us remark that I_1 is also correct for both M_2 and M_3 . On the contrary, I_2 is incorrect for them since the sequence $a/?, c/z$ cannot be performed from any state belonging either to M_2 or M_3 . If we consider the invariant

$$I_3 = b/y, a/\{y\}$$

we have that I_3 is incorrect for M_1 . For instance, we have a transition labelled by b/y outgoing from the state $(1, 1)$ and reaching the same state $(1, 1)$. Then, we have a transition labelled by a/x from the state $(1, 1)$. So, there exists a state (in this case $(1, 1)$) such that the sequence of transitions b/y can be performed in such a way that the reached state (i.e. $(1, 1)$) may perform a transition whose input action is a but the corresponding output action does not belong to the set $\{y\}$. Besides, this invariant is correct for M_2 while it is incorrect for M_3 . \square

In Figure 3 we extend the previous algorithm to deal with invariants containing the wild-card character $*$. As in the previous case, we traverse the invariant from left to right. We also have that the external *while-loop* has as termination condition that either the remaining sequence has length one or that the current

set of states is empty. However, instead of advancing by incrementing a counter, we consider two auxiliary functions: **head**(I) returns the first element of I and **tail**(I) removes the first element from I . If the first element of the remaining invariant is a pair i/o where $i \in \mathcal{I} \cup \{?\}$ and $o \in \mathcal{O} \cup \{?\}$ then the algorithm proceeds as the algorithm presented in Figure 2. If we have that the first element is $*$ then we skip all consecutive $*$'s. Afterwards, we consider the first element of the remaining trace. Let us remark that this element must be a pair i/o . We compute those states s connected with one of the states $s' \in S'$ by a path that does not contain the symbol i . These paths are computed by the predicate $path(s', s, i)$. Formally, $path(s', s, i)$ if there exists a sequence of input/output pairs $tr = a_1/z_1, \dots, a_r/z_r$ such that $s' \xrightarrow{tr} s$ and for any $1 \leq j \leq r$ we have $a_j \neq i$. As a special case, if i is equal to $?$ then $path(s', s, ?)$ holds if $s' \xrightarrow{*} s$. Let us remark that the complexity in time in the worst case for computing this new set of states is in $\mathcal{O}(|S| \cdot |Tr|)$. This is so because we only need to compute a breath-first-search (the complexity of this operation is in $\mathcal{O}(|Tr|)$) for each of the states belonging to S' (at most $|S|$ states). Again, if we consider that the induced graph is connected then we have that the previous complexity is bounded by $\mathcal{O}(|Tr|^2)$. Besides, we need $|S| + |Tr|$ additional space. Finally, the last element of the sequence is treated as in the algorithm given in Figure 2.

The next result indicates the complexity of the previous algorithm. We consider that there are no trailing occurrences of the wild-card character $*$ in invariants, that is, no consecutive occurrences of $*$.

Proposition 2. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a FSM and $I = i_1/o_1, \dots, i_n/o_n$ be an invariant without trailing occurrences of $*$. The worst case of the algorithm given in Figure 3 checks the correctness of the invariant I with respect to M in time $\mathcal{O}(k \cdot |Tr|^2 + (n - k) \cdot |Tr|)$, where k is equal to the number of $*$'s in I . Besides, the needed extra space is in $\mathcal{O}(|Tr|)$. \square

Example 3. If we consider again the FSMs depicted in Figure 1 we have that $a/x, *, b/\{y, z\}$ is a correct invariant for all of the specifications. \square

Next, we have to determine whether the trace obtained from the implementation satisfies the properties indicated by the invariants that we are interested in. Let us remark a very important difference with respect to previous proposals for passive testing. That is, a homing state phase is not needed for this kind of invariants. This is so because invariants have to be fulfilled at any point of the implementation. Thus, it is not relevant the state where the machine was placed when we started to observe the trace. In order to test the trace, we need to perform a pattern matching strategy. We have implemented a simple adaptation of the classical algorithms for pattern matching on strings (e.g. [BM77, KMP77]). The inclusion of wild-card characters is easy. In addition, for an invariant of length n we have to consider all the occurrences of the first $n - 1$ elements in the trace and then if we find a pair i/o such that $i_n = i$ (let us remind that if $i_n = ?$ then this equality holds) we have to check that $o \in \mathcal{O}$. We can say that we have found a mismatch (that is, a fault) if this last condition does not hold. Regarding the complexity of our pattern matching strategy, in the worst case we obtain $\mathcal{O}(m \cdot n)$. Let us remark that even though *good* algorithms for pattern matching on strings perform in $\mathcal{O}(m)$ (after the *pre-processing* phase) we cannot achieve this complexity because we must check all the occurrences of the pattern

in the trace. However, as we commented before, if we consider that the length of the invariant is *much smaller* than the length of the trace, as it is usually the case, we have that this complexity is almost lineal with respect to the length of the trace.

We finish this section by presenting some relations between different invariants and their correctness with respect to a given specification. The proofs of these results are easy (but tedious) with respect to the algorithm given in Figure 3.

Lemma 1. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM. The following properties hold:

- The invariant $*, i_1/o_1, \dots, i_n/O$ is correct for M iff $i_1/o_1, \dots, i_n/O$ is correct for M .
- If $i_1/o_1, \dots, i_n/O$ is correct for M and $O \subseteq O'$ then $i_1/o_1, \dots, i_n/O'$ is correct for M .
- Let $I = i_1/o_1, \dots, ?/o_j, \dots, i_n/O$ be a correct invariant for M . Then, for any $I' = i_1/o_1, \dots, i/o_j, \dots, i_n/O$, for $i \in \mathcal{I}$, such that $\exists s, s' \in S : s \xrightarrow{I'} s'$ we have that I' is correct for M .
- Let $i_1/o_1, \dots, i_j/?, \dots, i_n/O$ be a correct invariant for M . Then, for any $I' = i_1/o_1, \dots, i_j/o, \dots, i_n/O$, for $o \in \mathcal{O}$, such that $\exists s, s' \in S : s \xrightarrow{I'} s'$ we have that I' is correct for M .
- Let I be a correct invariant for M . If we consider the invariant I' where any occurrence of $*$ in I is replaced by a sequence of symbols $i_1/o_1, \dots, i_j/o_j$ such that $\exists s, s' : s \xrightarrow{I'} s'$ we have that I' is correct for M .

□

Let us note that the condition $s \xrightarrow{I'} s'$ appearing in the last three cases indicates that there exists (at least) a pair of states such that they are connected by the sequence I' . Moreover, the reverse implication of the last four results do not hold.

2.2 Introducing Obligation Invariants

In the previous section we have given our basic framework for invariants. Even though simple invariants allowed us to specify most of the properties that we desired to test for the WAP, we found some properties that we were not able to express. Specifically, we were unable to indicate properties as “the (output) action a does not appear before the sequence of actions tr has been observed”. Thus, we decided to extend the set of invariants to consider such properties. These new invariants, that we call *obligation invariants*, are introduced in the following definition.

Definition 4. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM. We say that the sequence tr is an *obligation invariant* for M if the following two conditions hold:

1. tr is defined according to the EBNF $tr ::= a/\overline{O} | *, tr | a/z, tr$. In the previous expression we consider $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $O \subseteq \mathcal{O}$.
2. tr is *correct* with respect to M .

We denote the set of obligation invariants for M by Obl_Inv_M .

□

Let us remark that, in contrast with simple invariants, we do not force the first symbol of the last pair to be an input action (it can also be the wild-card character ?).

Example 4. Obligation invariants may be used to express properties where the occurrence of an event must be necessarily preceded by a sequence of events. For example, the intuitive meaning of an invariant as

$$I = \text{request_page}/\text{req_ack}, *, ?/\overline{\{\text{page_sent}\}}$$

is that if the event *page_sent* is observed in the trace then we must have that a page had been requested before and that the server has acknowledged the reception of the request. \square

The first step consists again in deciding whether a candidate for invariant is indeed correct with respect to the specification. Due to space limitations we omit the code of the algorithm. The algorithm is similar to the ones presented in the previous section. However, there is an important difference: In this case, we have to traverse the invariant from right to left. Intuitively, we compute the set of states S' having an outgoing transition labelled by an output symbol in \overline{O} . If this set is empty then we discard the invariant. Next, we check that these transitions have as associated input i_n (if $i_n = ?$ then this condition always hold). If this is not the case then the invariant is incorrect; otherwise, we traverse the invariant from right to left and check that for any possible evolution of the specification we could always have performed the previous element of the invariant.

Example 5. Let us consider the FSMs depicted in Figure 1 and the following invariants

$$I_1 = a/y, ?/\overline{\{z\}} \quad I_2 = a/\overline{\{x\}}$$

We have that I_1 is correct for M_1 since every evolution reaching a state having a transition labelled by the output symbol z could have started with the pair a/y . On the contrary, I_1 is correct neither for M_2 nor M_3 since no transition contains the output symbol z . The invariant I_2 is correct for M_2 and M_3 . However, I_2 is not correct for M_1 . For example, the state $(1, 2)$ has a transition labelled by b/x . Thus, it does not hold that every occurrence of x is paired with a . \square

The complexity in time and space of the previous algorithm is again given by the complexity orders presented in Proposition 2. In contrast with the invariants introduced in the previous section, we now need some kind of homing. Specifically, we need to find out whether the observed trace has passed through a state in the implementation corresponding to the initial state of the specification. For example, let us consider a vending machine that returns a tea only after two coins have been introduced. If we observe that a tea has been returned but we have not observed before that two coins were introduced we cannot deduce that the machine is faulty. It may happen that we started to observe *too late*. So, we need to know that the machine was in its initial state and we have to discard the prefix of the observed trace until that point. In order to perform the task of determining which point of the trace corresponds with the initial state we consider two possibilities. Next we briefly sketch them. The first one consists in using a simple adaptation of the UIO method [SD88,ADLU88]. If we can find a UIO sequence for the initial state, and we observe that sequence in the trace then

we know that we have to discard the preceding part of the trace. The second method is inspired by the classical homing state for passive testing. In this case, once a state has been identified we go forward trying to determine the point of the trace corresponding to the initial state. Finally, if we are able to identify the desired point of the trace, we have to match the trace and the invariants. In this case, we use an adaptation of [BM77] where pattern matching is performed from right to left. As in the previous case, we again obtain a performance in the worst case in $\mathcal{O}(m \cdot n)$ being m and n the sizes of the trace and of the invariant, respectively

We finish this section with a result similar to Lemma 1 but in the case of obligation invariants.

Lemma 2. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM. The following properties hold:

- The invariant $*, i_1/o_1, \dots, i_n/\overline{O}$ is correct for M iff $i_1/o_1, \dots, i_n/\overline{O}$ is correct for M .
- If $i_1/o_1, \dots, i_n/\overline{O}$ is correct for M and $O' \subseteq O$ then $i_1/o_1, \dots, i_n/\overline{O'}$ is correct for M .
- If $i_1/o_1, \dots, i_j/o_j, \dots, i_n/O$ is correct for M then $i_1/o_1, \dots, ?/o_j, \dots, i_n/O$ and $i_1/o_1, \dots, i_j/?, \dots, i_n/O$ are both correct for M .
- Let I be a correct invariant for M . If we consider the invariant I' where we have replaced any sequence of symbols $i_1/o_1, \dots, i_j/o_j$ by $*$ then we have that I' is correct for M .

□

2.3 Extending Invariants to deal with EFSMs

The extension of our framework to deal with EFSMs is far from trivial. In particular, we have the problem that the values of the variables cannot be, in general, observed. However, it is very easy to adapt our formalism to deal with invariants containing only constant data. Actually, this small inclusion is rather useful when dealing with real protocols as the WAP. For instance, we may use invariants as

$$req_connect(Peter)/?, *, req_disconnect(Peter)/\{disconnected(Peter)\}$$

$$req_connect(Peter)/?, *, ?/\overline{\{disconnected(Peter)\}}$$

to make sure that the disconnection of the service performed by *Peter* is linked to a request of connection made by Peter himself.

3 Applying Passive Testing with Invariants

In this section we report our experiments with the WAP (Wireless Application Protocol) when using our notion of passive testing with invariants. We briefly present this protocol, we explain how our observation points are placed, and we discuss on the invariants that we use as well as on the results that we obtained.

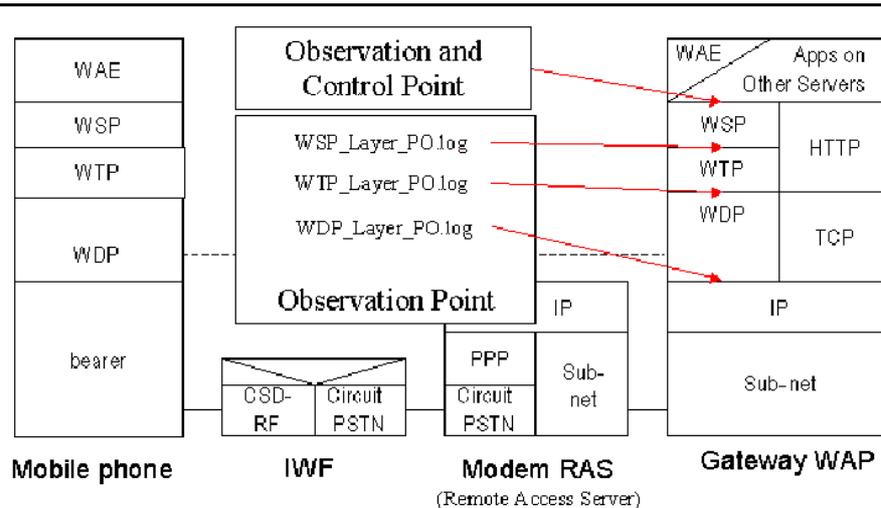


Fig. 4. A Passive Test Architecture for the WAP.

3.1 The Wireless Application Protocol

The WAP is the standard protocol conceived to provide Internet content and advanced telephony services to wireless terminals. The Wireless Application Environment, in short WAE, is the main interface to the client device. It includes the content to be displayed (that is, a WML page). The WSP is a stateful binary protocol used in conjunction with WTP and WDP to provide session oriented services, or directly with WDP to provide connectionless service. It supports sessions initiation, suspension, and resumption. A session is initiated by a WAP client and is maintained until it is explicitly disconnected. WTP, is a confirmed transaction protocol, a light weighted version of TCP. There are three classes: a non-confirmed simple flow of information in one direction (class 0), a simple send-acknowledge exchange (class 1) and a class 2 for a three-way handshake. WTP also has an optional capability to segment and reassemble data. WDP, is a datagram oriented, network layer protocol. Its main purpose consists in making lower layers transparent to higher ones. It makes no delivery confirmation, packet retransmission or error correction. WTLS is a session oriented, secure protocol layer conceived after the Secure Session Layer (SSL) and Transaction Layer Security (TLS) protocols. This is optional and independent of the other parts. A schematic presentation of the protocol stacks is given in Figure 4.

We have developed an architecture capable of dealing with passive testing in a mobile phone environment (GSM-WAP). In order to do so, we deployed a platform that behaves as a normal WAP gateway.⁵ In addition, we have included observation points, in short POs. These POs are placed in every layer of the WAP stack to show the flow of information in real time. So, whenever a communication between a mobile phone and a gateway exists, we have access to the involved

⁵ That is, a network component that works as interface between the mobile phone side (wireless communication) and the Internet. The gateway we took is called Kannel. It is a free-software and it can be downloaded from <http://www.kannel.org>.

was running autonomously and we were recording traces of 300 input/output pairs. Next we comment on the most relevant invariants that we were specifying. First, we will present a *misleading* example of our method. We considered the invariant

$$I_1 = TR-Invoke.ind/? , *, TR-Result.res / \{TR-Result.cnf\}$$

In other words, we tried to check that whenever the client (mobile phone side) asks to download a WAP page, it is successfully received. Our experiments indicated that the traces were correct with respect to this invariant. However, we knew that this *invariant* was in fact not correct! Actually, an *abort* event can appear if the operation cannot be completed. For instance, this is the case if the web page is not available. So, we *removed* the requested web page and we found that the new observed trace did not respect the invariant. In fact, the correct invariant is

$$I_2 = TR-Invoke.ind/? , *, TR-Result.res / \{TR-Result.cnf, TR-Abort.ind\}$$

Continuará.... Por cierto, he aumentado el tamaño de página porque sino nos pasábamos de las 16 (ya tenemos 18 con su estilo....). Pero casi no se nota ;-) Además, no he encontrado la cita que me dijiste por teléfono sobre un overview de testing.

Finally, we would like to briefly comment on the implementation details of our framework. We have developed a package where all the algorithms related to this paper are implemented. Besides, suitable interfaces between the corresponding modules have been implemented so that the process can be completely automatized. The code was completely written in C (on Linux) so that the whole test platform can be as portable as possible. As we have explained in the bulk of the paper, we have implemented two modifications of classical pattern matching algorithms to work with pattern matching functions. Specifically, the pattern matching algorithm for simple invariants is an adaptation of [KMP77] where pattern matching is performed from left to right; the algorithm for obligation invariants is an adaptation of [BM77] where pattern matching is performed from right to left. The algorithms for deciding whether invariants are correct with respect to a specification are completely original. The whole code for all of our algorithms and interfaces is around 1000 lines long.

4 Conclusions and Future Work

To be provided...

References

- [ADLU88] A. Aho, A. Dahbura, D. Lee, and M. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman tours. In *Protocol Specification, Testing and Verification VIII*, pages 75–86. North Holland, 1988.
- [BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

- [CGP01] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. In *Concordia Prestigious Workshop on Communication Software Engineering*, pages 225–250, 2001.
- [KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.
- [Lai02] R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62:21–46, 2002.
- [LCH⁺02] D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions, 2002. To appear in Proceedings of IEEE Int. Conf. on Network Protocols, ICNP'02.
- [LNS⁺97] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *IEEE Int. Conf. on Network Protocols, ICNP'97*, pages 113–122. IEEE Computer Society Press, 1997.
- [LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [MA01a] R.E. Miller and K.A. Arisha. Fault coverage in networks by passive testing. In *International Conference on Internet Computing 2001, IC'2001*, pages 413–419. CSREA Press, 2001.
- [MA01b] R.E. Miller and K.A. Arisha. Fault identification in networks by passive testing. In *34th Simulation Symposium, SS'01*, pages 277–284. IEEE Computer Society Press, 2001.
- [Mil98] R.E. Miller. Passive testing of networks using a CFSM specification. In *IEEE Int. Performance Computing and Communications Conference*, pages 111–116. IEEE Computer Society Press, 1998.
- [SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15:285–297, 1988.
- [TC99] M. Tabourier and A. Cavalli. Passive testing and application to the GSM-MAP protocol. *Journal of Information and Software Technology*, 41:813–821, 1999.
- [TCI99] M. Tabourier, A. Cavalli, and M. Ionescu. A GSM-MAP protocol experiment using passive testing. In *World Congress on Formal Methods in the Development of Computing Systems, FM'99, LNCS 1708*, pages 915–934. Springer, 1999.