A Passive Testing Approach based on Invariants: Application to the WAP*

Emmanuel Bayse[1], Ana Cavalli[1], Manuel Núñez[2] and Fatiha Zaïdi[1]

[1] Institut National des Télécommunications GET-INT
91011 Evry Cedex, France
{Emmanuel.Bayse,Ana.Cavalli,Fatiha.Zaidi}@int-evry.fr
[2] Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, E-28040 Madrid. Spain.
mn@sip.ucm.es

Abstract This paper presents a new methodology to perform passive testing based on invariants. This novel approach is supported by the following idea: A set of invariants represent the most relevant expected properties of the implementation under test. Intuitively, an invariant expresses the fact that each time the implementation under test performs a given sequence of actions, then it must exhibit a behavior reflected in the invariant. For example, an invariant such as $i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/O$ must be interpreted as "*each time the implementation performs the sequence* $i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n$ *the next observed output belongs to the set* O". We call these invariants *simple invariants*. In this work we introduce a new notion of invariants to deal with more subtle properties. For instance, we will consider invariants to express properties such as "*if* y *happens then we must have that* x *had happened before*". These invariants are called *obligation invariants*. We present algorithms to decide the correctness of the proposed invariants with respect to a given specification. Once we have that an invariant is correct with respect to a given specification, we check whether the execution traces observed from the implementation *respect* the invariant. In order to perform this phase we present two algorithms based, respectively, on left-to-right and right-to-left pattern matching algorithms.
In addition to the theoretical framework we have developed a software tool, called TestInv, that helps in the automation of our passive testing approach. In particular, the algorithms presented in this paper are fully implemented in the tool. Finally, in order to test the usefulness of our approach we have chosen a real-life case study: The Wireless Application Protocol (WAP). We present a test architecture as well as the most relevant results obtained from the application of our approach to the WAP.

Keywords: passive testing, conformance testing, invariants, software tools for testing, Wireless Application Protocol (WAP).

* Research supported in part by the Spanish *Ministerio de Ciencia y Tecnología* MCyT project *MASTER* (TIC2003-07848-C02-01), the Junta de Castilla-La Mancha project *DISMEF* (PAC-03-001) and the Marie Curie RTN *TAROT* (MCRTN 505121). This research was carried out while the third author was visiting the GET-INT under the financial support of the Platonis project.

1 Introduction

The activity of conformance testing is essentially focused on verifying the conformity of a given implementation to its specification. In most cases testing is based on the ability of a tester that stimulates the implementation under test and checks the correction of the answers provided by the implementation [Lai02,LY96]. However, in some situations this activity becomes difficult and even impossible to perform. For example, this is the case if the tester is not provided with a direct interface to interact with the implementation under test (IUT). Another conflictive situation appears when the implementation is built from components that are running in their environment and cannot be shutdown or interrupted for a long period of time. In these situations, there is a particular interest in using other types of testing techniques such as *passive testing*. In passive testing the tester does not need to interact with the IUT. On the contrary, the execution traces are observed without interfering with the behavior of the IUT. Passive testing has very large domains of application. For instance, it can be used as a monitoring technique to detect and report errors (this is the use that we consider in this paper). Another area of application is in network management to detect configuration problems, fault identification, or resource provisioning (e.g. [MA01,WZY01]). It can be also used to study the feasibility of new features as classes of services, network security, and congestion control.

Even though passive testing techniques are not new (see for example the approach shown in [AAD79]) in the last years a very active research on passive testing has been developed. Usually, the execution traces of the implementation are compared with the specification to detect faults in the implementation [LNS+97,Mil98,TC99,TCI99]. In general, the specification has the form of a finite state machine (`FSM`) and the work consisted in verifying that the executed trace is accepted by the `FSM` specification. A drawback of these first approaches is the low performance of the proposed algorithms (in terms of complexity in the worst case) if non-deterministic specifications are considered.

A new approach was proposed in [CGP03]. There, a set of properties, called *invariants*, were extracted from the specification and checked on the traces observed from the implementation to test their correctness. That is, in this approach information was extracted from the specification and then used to process the trace. However, one of the drawbacks of this work is the limitation on the grammar used to express invariants. For instance, 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.*

could not be easily represented by using their invariants since all the possible sequences of actions expressing the idea of *some operations* must be explicitly written.

A new formalism to express invariants was presented in [ACN03]. For instance, the possibility of specifying wild-card characters in invariants was added.

Besides, a set of outputs was allowed (instead of a single output) as termination of the invariant. Thus, properties such as

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

could be easily specified. However, heavy experimentation using the invariants approach reported in [CGP03,ACN03] has shown additional lines for improvement. For example, properties such as

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

could not be expressed at all by using these approaches. This paper represents an extended and revised version of [ACN03]. In order to easily understand the new contributions we provide a short outline of our approach. In [ACN03], it was proposed that invariants should be supplied by the expert/tester. In this case, the first step must be to check that the invariant is in fact correct with respect to the specification. An algorithm to check this correctness was provided. The complexity, in the worst case, of the algorithm was linear, with respect to the number of transitions in the specification, if the invariant did not contain the wild-card character $*$; this complexity is quadratic if the symbol $*$ appears in the invariant. Once a set of (correct) invariants is generated the second step consists in checking whether the trace produced by the IUT respects the invariants. In order to do so a simple adaptation of the classical algorithms for pattern matching on strings (see e.g. [BM77,KMP77]) was implemented. This algorithm works, in the worst case, in time $\mathcal{O}(m \cdot n)$, where m and n are the length of the trace and 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 previous complexity is almost linear with respect to the length of the trace. In the current paper we introduce a new notion of invariant to express properties as "if y happens then we must have that x had happened before". These invariants are called *obligation invariants*.

A parallel between the approach presented in this paper and *model checking* techniques used for the verification of distributed systems can be established (see e.g. [CGP99]). Model checking techniques deal with checking whether the design of a finite state system satisfies some properties. In other words, it consists of verifying the correctness of requirements that can be expressed as invariants, i.e. properties that remain invariantly true for all possible executions of the system. Model checking is well adapted for the verification of properties of an abstract model of the system implementation. However, it is not so suitable for the verification of properties on a real system implementation.

In our approach, we perform two types of property verification: one on the specification and another one on the implementation. For the first type of verification we have developed algorithms whose complexity are better than classical algorithms for model checking, since these ones are usually exponential on the number of transitions. For the second type of verification we have developed new algorithms that check the properties on the real implementation traces. These

algorithms are adaptations of classical algorithms for pattern matching. Let us remark that this kind of verification is not performed at all by model checking. Thus, our techniques are indeed closer to conformance testing or system monitoring than to model checking.

In order to ease the passive testing process we have also developed a software tool, called TestInv, to automate all the phases of our passive testing approach. In particular, the tool includes the algorithms to check the correctness of invariants with respect to the specification and to decide whether the trace observed from the implementation respects the invariants. Moreover, TestInv has been integrated into the PLATONIS platform [Pla01]. This platform consists of a set of tools to perform both active and passive testing. In fact, PLATONIS has been already applied to the validation of protocols and services on mobile environments.

In order to test the suitability of our framework we have extensively worked with a real case study: The Wireless Application Protocol (WAP). This protocol is an open global specification that empowers mobile users with wireless devices to easily access and interact with Internet information and services instantly. It is worth to point out that this protocol represents a typical example where active testing cannot be applied since, in general, there is no direct access to the interfaces between the different layers. Thus, the tester cannot control how internal communications are established. In order to perform the corresponding experiments we have installed the software free protocol stack Kannel. Moreover, both a platform and a test architecture capable to deal with passive testing in a mobile phone environment (WAP, GPRS, UMTS) have been defined. The architecture is based on points of observation installed between the different layers. Finally, let us remark that the experiments performed on the WAP, and reported in this paper, represent an original contribution because such type of study has been never performed in a systematic way.

The rest of the paper is organized as follows. In Section 2 our notions of invariant and passive testing are presented. First, we introduce what we call *simple invariants*. Next, *obligation invariants* are presented. We also present the algorithms to check the correctness of these different types of invariants on the specification and explain how invariants are checked on the observed execution traces. In Section 3 we discuss some of the limitations of our approach. In Section 4 we describe the capabilities of the software tool TestInv as well as some implementation details. In Section 5 the protocol WAP and results of the experiments performed using TestInv are presented. This section also introduces the new test architecture used for the experiments. Finally, in Section 6 we give our conclusions and guidelines for future work.

2 Invariants and Passive Testing

This section introduces 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.

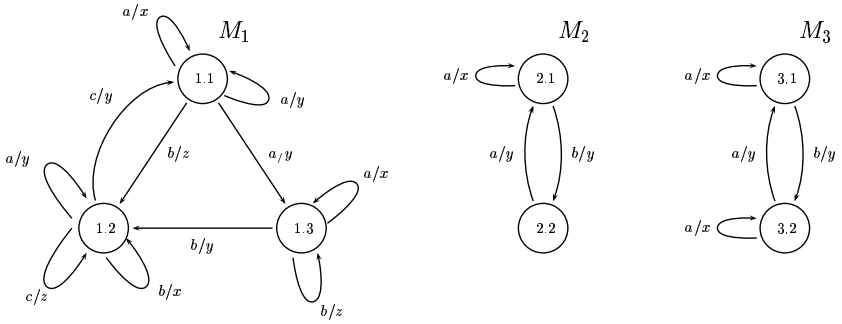

Figure 1. Examples of FSMs.

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 $tr = i_1/o_1, \ldots, i_n/o_n$, with $n \geq 1$, 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 $t_1, \ldots, t_n \in Tr$ and states $s_1, \ldots, s_{n-1} \in S$ 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 $t_j = (s_{j-1}, s_j, i_j, o_j)$. □

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 usually write $s \xrightarrow{i/o} s'$ to denote the transition $(s, s', i, o) \in Tr$. We can extend the notion of single transition to a sequence of transitions. Thus, $s \xrightarrow{tr} s'$ 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. In Figure 1 we present some simple examples of Finite State Machines.

Once we have a FSM we may extend its set of transitions so that the wild-card characters ? and $*$ can be taken into account. In particular, these special symbols can appear as part of a sequence of transitions.

Definition 2. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be an FSM. We write $s \xrightarrow{?/o} s'$ if there exists $i \in \mathcal{I}$ such that $s \xrightarrow{i/o} s'$. We write $s \xrightarrow{i/?} s'$ if there exists $o \in \mathcal{O}$ 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 $s \xrightarrow{tr} s'$. □

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 such as $i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/O$ is a simple invariant for M if each time that the trace $i_1/o_1, \ldots, i_{n-1}/o_{n-1}$ 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 and output symbols we will allow the *wildcard* characters ? and $*$. In our framework, the meaning of ? 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 I is a (simple) *invariant* for M if the following two conditions hold:

1. I is defined according to the following EBNF:

$$I ::= i/O \mid *, I \mid a/z, I$$

   In this expression we consider $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $O \subseteq \mathcal{O}$.
2. I is *correct* with respect to M.

□

Intuitively, the previous EBNF expresses that an invariant is a sequence of symbols where each component, but the last one, is either a pair a/z, with a being an input action or the wildcard character ? and z being an output action or the wildcard character ?, or the wildcard character $*$. The last component is an input action followed by a set of output actions. 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 each time that 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 occurrence of the symbol *req_disconnect* 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 consider only the cases where the connection was granted

$$I_3 = req\_connect/granted\_connection, *, req\_disconnect/\{disconnected\}$$

For instance, an observed 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 $disconnected$. Let us remark that we could not deduce that we have found an error if the pair $req\_connect/granted\_connection$ appears in the observed trace but the input $req\_disconnect$ is not detected afterwards in the corresponding trace. In such a situation 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. □

We could adapt to our framework the algorithm given in [CGP03] 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 observed from the behavior of 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 firstly present an algorithm (see Figure 2) deciding correctness of invariants without occurrences of the $*$ wild-card character. In Figure 3 we extend this algorithm to deal with invariants where the symbol $*$ can appear.

The algorithm given in Figure 2 works as follows. The first *while-loop* computes those states $s \in S$ that can be reached starting from one of the states belonging to S after performing the sequence $i_1/o_1, \ldots, i_{n-1}/o_{n-1}$. Let us remark that if one of the symbols in the sequence is the wild-card character ? then any symbol can be used to match it. Besides, it may happen that after some steps we find that there do not exist two states connected by the analyzed

Input: $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$, $I = i_1/o_1, \ldots, i_n/O$, $\forall\ 1 \leq j \leq n : i_j \neq * \ \wedge\ o_j \neq *$
Output: `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; transition_found := 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 then begin
                                  transition_found := true;
                                  if z ∉ O then error := true
                          end
                  end;
                  if not transition_found then error := true;
                  return (not error)
          end;
```

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

Figure 2. Checking correctness of Simple Invariants (1/2).

sub-sequence. In this case S' becomes empty and we exit the *while-loop*. 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 auxiliary 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. 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. Once again, if there is no possible transition we consider that the invariant is incorrect. Let us note that the complexity of this last loop is also

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 $I = i_1/o_1, \ldots, i_n/O$ 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 $\mathcal{O}(|Tr|)$. □

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

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

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

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_3. For instance, there exists a transition labelled by b/y outgoing from the state 3.1 and reaching the state 3.2. In addition, we have a transition labelled by a/x from the state 3.2. So, there exists a state (in this case 3.1) such that the sequence of transitions b/y can be performed in such a way that the reached state (i.e. 3.2) may perform a transition whose input action is a but the corresponding output action does not belong to the set $\{y\}$. On the contrary, it is straightforward to check that this invariant is correct for M_1 and M_2. □

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 one 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. Obviously, this element must be a pair i/o. Given $s' \in S'$ we compute the states $s \in S$ connected to s' by a path that does not contain the symbol i. These paths are computed by using the predicate $path(s', s, i)$. Formally, $path(s', s, i)$ holds if there exists a sequence of input/output pairs

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

$I' := I$; $S' := S$;
while $I' \neq b/O$ **and** $S' \neq \emptyset$ **do begin**
  $first := \mathbf{head}(I')$; $I' := \mathbf{tail}(I')$;
  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 $\mathbf{head}(I') = *$ **do** $I' := \mathbf{tail}(I')$; {`skip seq of *'s`}
    $first := \mathbf{head}(I')$; $\{first = i/o\}$
    $S' := \{s \in S \mid \exists s' \in S' : path(s', s, i)\}$
  end
end;
{`last pair of the invariant or empty set` S' `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.

Figure 3. Checking correctness of Simple Invariants (2/2).

$tr = a_1/z_1, \ldots a_r/z_r$ 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 there exists a path from s' to s, that is, if $s' \xrightarrow{*} s$.

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

Let us remark that the time complexity in the worst case for computing this new set of states belongs to $\mathcal{O}(|S| \cdot |Tr|)$. 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 $I = i_1/o_1, \ldots, i_n/O$ 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. The required extra space is in $\mathcal{O}(|Tr|)$. □

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 comment a very important difference with respect to previous proposals for passive testing: A *homing* state phase (that is, to identify when the sequence was passing by the initial state) 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 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) then we have to check that $o \in O$. 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 $\mathcal{O}(m \cdot n)$ (n is the length of the invariant and m is the length of the observed trace). 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 linear 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 $*, i_1/o_1, \ldots, i_n/O$ is correct for M iff $i_1/o_1, \ldots, i_n/O$ is correct for M.
- If $i_1/o_1, \ldots, i_n/O$ is correct for M and $O \subseteq O'$ then $i_1/o_1, \ldots, i_n/O'$ is correct for M.
- Let $i_1/o_1, \ldots, ?/o_j, \ldots, i_n/O$ be a correct invariant for M. Then, for any $I' = i_1/o_1, \ldots, i/o_j, \ldots, i_n/O$, with $i \in \mathcal{I}$, such that $\exists\, s, s' \in S :\ s \xrightarrow{I'} s'$ we have that I' is correct for M.
- Let $i_1/o_1, \ldots, i_j/?, \ldots, i_n/O$ be a correct invariant for M. Then, for any $I' = i_1/o_1, \ldots, i_j/o, \ldots, i_n/O$, with $o \in \mathcal{O}$, 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 $i_1/o_1, \ldots, i_j/o_j$ 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 connected by the sequence I'. Let us also remark that 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 with them. Specifically, we were unable to indicate properties such as "an output action does not appear before a sequence of actions (i/o pairs) 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 I is an *obligation invariant* for M if the following two conditions hold:

1. I is defined according to the following EBNF:

$$I ::= a/\overline{O} \mid *, I \mid a/z, I$$

   In this expression we consider $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $\overline{O} \subseteq \mathcal{O}$.
2. I is *correct* with respect to M.

□

Let us remark that, in contrast with simple invariants, we do not force the first symbol of the last pair of an obligation invariant 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 such as

$$I = request\_page/req\_ack, *, ?/\overline{\{page\_sent\}}$$

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. □

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

```
{last pair of the invariant}
T := T_r; S' := ∅; error := false;
while T ≠ ∅ and not error do begin
        choose t ∈ T; {t = (s, s', a, z)}
        T := T − {t};
        if z ∈ Ō then if a = i_n  then S' := S' ∪ {s}
                                 else error := true
end;
if S' = ∅ then error := true;
{last pair of the invariant treated}

j := n − 1;
while j ≥ 1 and not error do begin
        S'' := ∅; T := T_r; S''' := S';
        while T ≠ ∅ do begin
                choose t ∈ T; {t = (s, s', a, z)}
                T := T − {t};
                if s' ∈ S''' then if a = i_j and z = o_j then begin
                                                                S' := S' − {s'};
                                                                S'' := S'' ∪ {s}
                                                              end
                                                              else error := true
        end;
        if S' ≠ ∅ then error := true
                  else begin
                          S' := S'';
                          j := j − 1
                  end
end;
return (not error);
```

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

Figure 4. Checking correctness of Obligation Invariants (1/2).

The first step consists again in deciding whether a candidate for invariant is indeed correct with respect to the specification. Even though the philosophy underlying the algorithm is similar to the one for the algorithms presented in the previous section, there is an important difference: We traverse the invariant from right to left. The algorithm given in Figure 4 works as follows. The first *while-loop* analyzes the last pair of the invariant. We compute the set of states S' having an outgoing transition labelled by an output symbol in $\overline{O}$. Moreover, those transitions must necessarily have as associated input i_n (if i_n =? then this condition always hold); otherwise the invariant is discarded. Furthermore, if the

set of computed states is empty then the invariant is incorrect. A similar decision was taken for simple invariants, that is, we consider that if the trace defining the invariant cannot be performed by the specification then the invariant is *useless*. Once the last pair of the invariant has been treated, we traverse the rest of the invariant from right to left. We check for all states belonging to S' that there exists at least one transition reaching one state of S' and satisfying the previous element of the invariant. Then, we remove these state from the set S'. Meanwhile, we compute the new set of states corresponding to the originating state of those transitions. If the new set S' is not empty at the end of an iteration then the invariant is not correct. Actually, an *unused* state in S' indicates that there exists a partial path that cannot be completed.

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

$$I_1 = a/?, ?/\overline{\{x\}} \qquad I_2 = a/\overline{\{x\}}$$

We have that I_1 is correct for M_2 since every path reaching a state having a transition labelled by the output symbol x must have been preceded by a pair a/o, for some output action o. On the contrary, I_1 is correct neither for M_1 nor M_3. 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. □

In Figure 5 we present the algorithm to deal with invariants containing the wild-card character $*$. As the algorithm described above, we traverse the invariant from right to left. However, instead of using an index to go backward, we use two auxiliary functions: **last**(I) returns the last element of I, and **tail**$'(I)$ removes the last element from I. Moreover, we use the **empty**(I) function which returns `true` if the corresponding invariant has been completely traversed. We treat the last pair of the invariant as explained in Figure 4. Once this last pair is checked, so that we obtain a set of states S', we continue the process from right to left. Afterwards, if the previous element of the invariant is an i/o pair then we proceed as explained in Figure 4. Otherwise, we have that the next element to be dealt with is a wild-card character $*$. We compute the new set of states S such that for any state belonging to S', there exists a path which satisfied the predicate $path'(s, s', o)$ with $s \in S$, $s' \in S'$ and o being the output of the previous element of the invariant (before the $*$). Let us note that if the predicate $path'(s, s', o)$ is satisfied then the state s' is removed from the set S'. Formally, $path'(s, s', o)$ holds if there exists a sequence of input/output pairs $tr = a_1/z_1, \ldots a_r/z_r$ such that $s \xrightarrow{tr} s'$ and for any $1 \leq j \leq r$ we have $z_j \neq o$.

Finally, let us remark that the complexity in time and space of the previous algorithms is again given by the complexity orders presented in Proposition 2.

Regarding the validity (with respect to an invariant) of traces observed in the implementation under test, the procedure is very similar to simple invariants but we have to point out a notable difference. In contrast with the invariants introduced in the previous section, we now need some kind of homing sequence. Indeed, we need to find out whether the observed trace has passed through a

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

{`last pair of the invariant`}
See Algorithm in Figure 4 for dealing with the last pair of the invariant.

```
I' := I;
while not empty(I') and not error do begin
        S'' := ∅;
        end_elt := last(I'); I' := tail'(I');
        if end_elt ≠ * then begin {end_elt = i/o}
                T := T_r; S''' := S';
                while T ≠ ∅ do begin
                        choose t ∈ T; {t = (s, s', a, z)}
                        T := T − {t};
                        if s' ∈ S''' then if a = i and z = o then begin
                                                                  S' := S' − {s'};
                                                                  S'' := S'' ∪ {s}
                                                          end
                                                          else error := true
                end
        end
        else begin {end_elt := *}
                while last(I') = * do I' := tail'(I'); {skip a seq of *'s}
                end_elt := last(I'); {end_elt = i/o}
                S'' := {s ∈ S | ∀s' ∈ S' : path'(s, s', o)}
        end;
        if S' ≠ ∅ then error := true else S' := S''
end;
return (not error);
```
We consider that both i =? and o =? hold.

Figure 5. Checking correctness of Obligation Invariants (2/2).

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 is returned but we did not observed before that two coins were inserted then 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 move 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 obtain again 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 $*, i_1/o_1, \ldots, i_n/\overline{O}$ is correct for M iff $i_1/o_1, \ldots, i_n/\overline{O}$ is correct for M.
- If $i_1/o_1, \ldots, i_n/\overline{O}$ is correct for M and $O' \subseteq O$ then $i_1/o_1, \ldots, i_n/\overline{O'}$ is correct for M.
- If $i_1/o_1, \ldots, i_j/o_j, \ldots, i_n/O$ is correct for M then $i_1/o_1, \ldots, ?/o_j, \ldots, i_n/O$ and $i_1/o_1, \ldots, i_j/?, \ldots, i_n/O$ 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 $i_1/o_1, \ldots, i_j/o_j$ by $*$ then we have that I' is correct for M.

□

3 Remarks on the proposed Approach

In this section we briefly review some of the limitations of our proposal and we show how they can be (partially) overcome. We will also comment on some of the issues regarding the definition of invariants.

First, let us remark that our approach suffers of the usual disadvantages of passive testing with respect to active testing. Specifically, for any context allowing non-determinism, any *passive approach* will have less distinguishing power than an appropriate *active approach.* Next, it can be asked *what can be expressed with our invariants?* In fact, we said that we were introducing obligation invariants as an extension and complement of simple invariants. However, we are aware that there exist properties that cannot be expressed in our notation. For instance, we can increase the expressive power of our invariants by considering *succession invariants* [CGP03]. This kind of invariants allows a tester to express that there exists a loop. They can be used, for instance, to detect whether an implementation does not authorize several attempts before a final rejection. We plan to include this kind of invariants as soon as we adapt our log files to cope with the new situation.

Another shortcoming of the current framework is that we do not explicitly consider time aspects. However, there are several ways of taking into account time aspects that are implicitly represented in the corresponding protocols. For

instance, and considering the application to WAP that we present in the forthcoming sections, we are able to identify problems due to an expiration of a timer because a service data unit primitive (e.g. Abort) is used to inform the lower and upper layers.

Finally, we do not completely cope with the data part of protocols. Indeed, 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. As we will comment in the conclusions of the paper, the work reported in [LCH+02] opens a new perspective for passive testing to cope with data values. Nevertheless, 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 an invariant as

$$req\_connect(Peter)/?, *, req\_disconnect(Peter)/\{disconnected(Peter)\}$$

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

Finally, it can be thought that the definition of invariants is a hard task. However, we claim that this is not the case. Indeed, even non-experts can write invariants for a given protocol with the help of the information provided by the standards. For instance, many standards supply Message Sequence Charts describing the exchanges of messages between layers. Once, the corresponding notations are assimilated, any tester should be able to write invariants with the same degree of effort as it would take to write a *usual* test.

4 Description of the TestInv Tool

This section is devoted to implementation and tool details. A tool, that we call TestInv, has been developed where all the algorithms described in this paper are fully implemented. The main task of the tool is to automate the process of checking the correctness of invariants on a specification and on determining whether real traces respect a given invariant. The code has been completely written in Java (J2SE 1.4.0 API specification). Let us remark that we have rewritten a previous prototype developed in C, and used in [ACN03], where only simple invariants were considered. We have used the *Regex* package to express invariants as regular expressions. Besides, Java allowed us to straightforwardly improve the existing Graphical User Interface (GUI). In TestInv, the new graphical interfaces have been developed by using the *Awt* and *Swing* packages. A high level description of the tool is shown in Figure 6.

TestInv has been integrated into the PLATONIS platform (see [Pla01] for a complete description of the overall project PLATONIS). This platform offers several capabilities and allows real experimentation for validating protocols and services related to mobile environments (e.g. GSM/WAP, GPRS, UMTS). By using different techniques, the aim is to check whether some protocol exchanges in WAP over GSM are correct, as well as to decide if different entities can

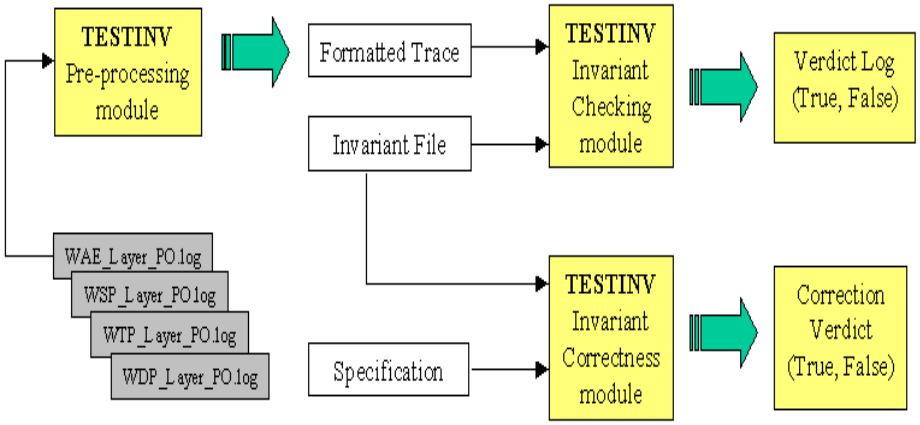

Figure 6. General diagram of TestInvtool

cooperate. In addition, it can be also used to decide whether the service is properly delivered. Active testing techniques are used if the interaction with the entity under test is feasible. In this case, the tester can be a PDA (Personal digital Assistance) or a programmable mobile phone. However, active testing is not always applicable, especially when testing layers do not have a direct access. So, in order to overcome this drawback we have included in PLATONIS the possibility of using the passive testing approach presented in this paper. Thus, TestInv is currently a component of the PLATONIS platform, being a complementary tool that is used when active testing is not possible. In this project we have implemented different test architectures that allow to combine active and passive testing. The verdict emitted by TestInv is sent to the active tester that receives all the verdicts and emits a global one. There is a script that permits to launch the required tools according to the test architecture.

In order to start the passive testing process, a preliminary phase is necessary. First, we have to obtain real traces from a running implementation. In order to obtain such traces, we use the observation points that have been implemented as shown in Figure 7. Then, the preprocessing module treats the log file containing the observed trace. This file has to be transformed into a suitable form. The tool takes as input the log file obtained from the real execution of the platform. This file is filtered and parsed in order to obtain a suitable trace, that is, a trace keeping the information concerning input and output primitive names as well as relevant data (e.g. source address, destination address, the accessed url, etc). The model of the output file can be customized. Thus, for each primitive name we assign the type in terms of input or output. Finally, in order to perform the verification of invariants on the specification we need to generate the underlying FSM. To obtain a FSM we use the ObjectGEODE tool [Ver97], that allows us

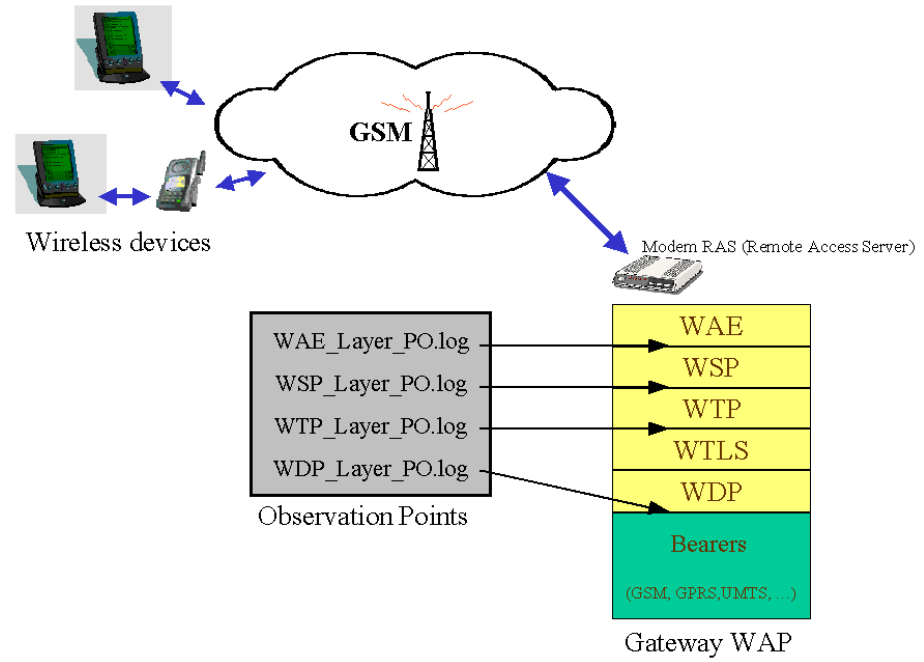

Figure 7. A Passive Testing Architecture for the WAP.

to translate an SDL [ITU92] specification into a `FSM` (i.e. a deployed `FSM`). In other words, this `FSM` represents an accessibility graph of the SDL description. This graph is engendered by using the exhaustive simulation capability of the ObjectGEODE tool. Moreover, this graph is constrained by fixing the values of the parameters according to the messages received from the system environment. These constraints are defined by configuring the environment by means of a specific `startup` file. Furthermore, the constraint on the parameters allows to obtain a calculable graph, and thus to avoid the state space explosion (more details on this method can be found in [CLRZ99]). In the framework of the `PLATONIS` project, three WAP layers have been described by using SDL. Thus, we were able to produce the `FSM` on which our experiments have been exercised (see Section 5).

Once this preprocessing phase has been completed, different modules of the tool take control of the (passive) testing process. The tool TestInv consists of two main modules. The first module is used to check the correctness of invariants on a given specification. In this part of the tool, the algorithms presented in the previous section (both for simple and obligation invariants) are implemented. The second module allows us to check the correctness of invariants on the traces observed from the implementation. As we have already mentioned, this module

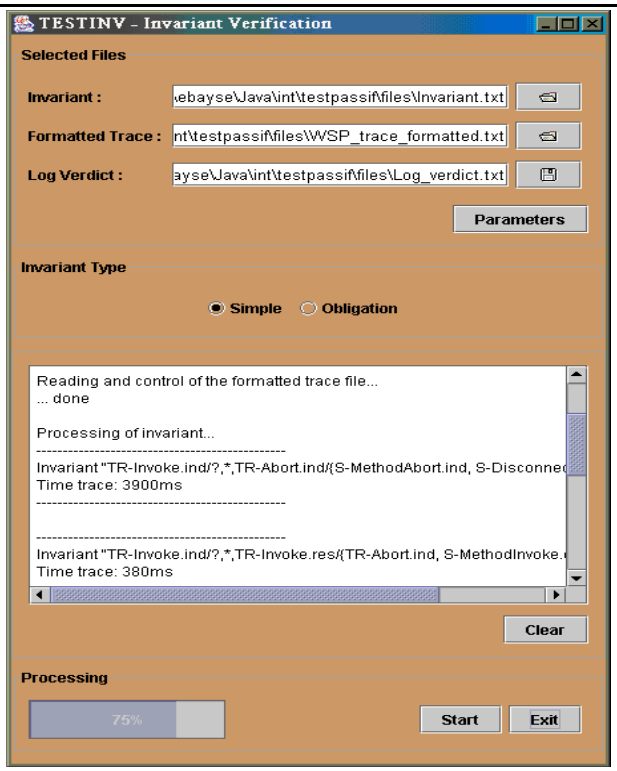

Figure 8. TESTINV- Invariant Checking Module

requires a preprocessing to deal with real protocol traces as well as to produce suitable traces. In other words, the generated traces are usually not directly usable. As explained in the bulk of the paper, two modifications of classical pattern matching algorithms have been implemented. 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. As an example of the provided graphical interfaces, in Figure 8 we present the one corresponding to this module.

We will conclude this section with some remarks concerning the experiments that we have performed on real protocols. Even though our tool has been designed to primarily deal with `FSM`s, we provide it with enough flexibility so that other models can be also considered. For instance, some implementations may be based in the `FSM` formalism but do not follow a strict alternation of inputs and outputs. In this case, the trace will be processed so that *dummy* symbols are added to the trace. For example, if an implementation produces a trace

$\cdots i, o_1, o_2, o_3, \cdots$ then the trace $\cdots i/o_1, d_i/o_2, d_i/o_3, \cdots$, where d_i represents a dummy input symbol, will be checked against the invariant.

5 Case Study: Testing of WAP properties

This section presents a report on the experiments performed on the WAP (Wireless Application Protocol). It also presents an outline of the protocol, an explanation of how observation points are placed, and the invariants that we were checking.

5.1 The Wireless Application Protocol

The WAP is the result of the efforts of the WAP Forum to promote industry-wide specifications for developing applications and services that operate over wireless communication networks. WAP specifies an application framework and network protocols for wireless devices such as mobile telephones, pagers, and PDA. One of the main objectives is to bring Internet content and advanced data services to digital cellular phones and other wireless terminals. An overview of the WAP architecture is given in Figure 7. On top of all, the application layer with the Wireless Application Environment (WAE) offers a framework for the integration of different WWW and mobile telephony applications. It includes the content to be displayed (that is a WML page). The Wireless Session Protocol (WSP) has been designed to work on top of either the datagram service (WDP) or the transaction service (WTP). A session oriented service is provided by WSP over WTP and WSP. A connectionless service is provided over WDP directly. WSP supports sessions initiation, suspension, and resumption.

Regarding services, the WTP is a confirmed transaction protocol, that is, a light weight version of the 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. Moreover, the WTP also has an optional capability to segment and reassemble data. On the other hand, the main purpose of the WDP is to make 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.

As explained in Section 4, we have developed, as part of the `PLATONIS` platform, a test architecture to deal with passive testing in a mobile phone environment (GSM/WAP). This test architecture includes a normal WAP gateway,[1] and a modem that behaves as a remote access point allowing any wireless client devices to access WAP pages using our WAP gateway. In addition, we placed some points of observation (PO) inside the WAP Kannel gateway to have a

[1] 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`.

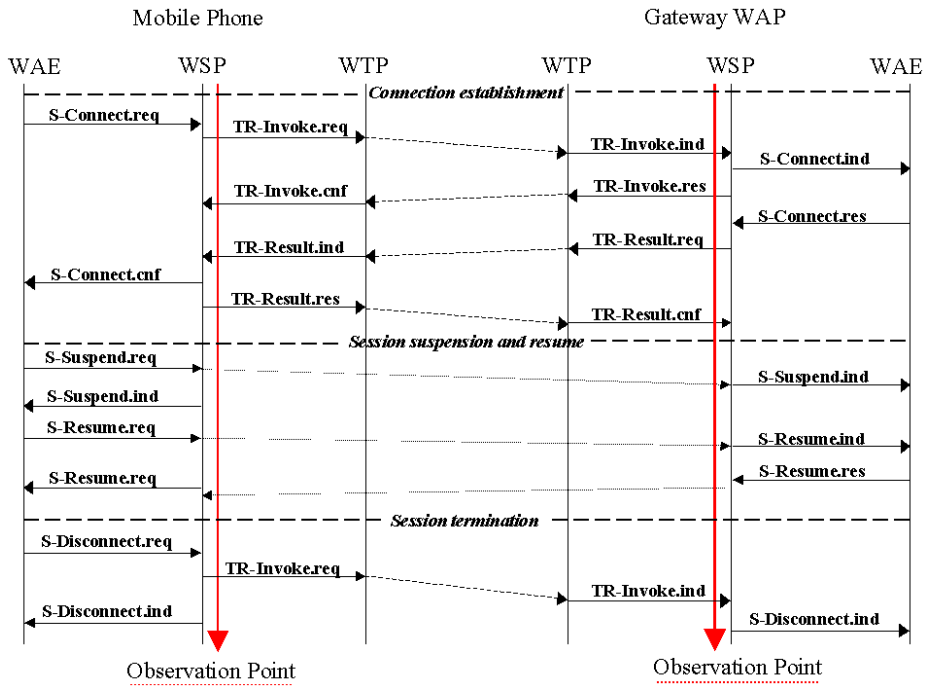

Figure 9. Messages between layers and WSP observation points.

real-time view of the information flow. So, whenever a communication between a wireless terminal and a gateway happens, we have access to the involved messages and the information contained inside them. It is important to note that a layer is able to interpret only data belonging to the layer itself. This means that embedded data (that is, data from an upper layer) is not visible. The current state of every layer is also shown. In order to have a closer control, an HTML interface with several PHP and CGI routines has also been developed.

5.2 WAP experiments

The experiments presented in this section strongly enhance the initial study performed in [ACN03] on testing the WAP. In particular, in this paper we deal with more complex log files as we have to cope with several POs. The experiments have been carried out in a connection oriented mode, through the WTP layer. For the results that we are going to relate in this section, only the POs placed in the WSP and WTP layers are relevant. Thus, we consider those messages that are sent either to or received from either the WTP layer or the WSP layer.

The relation between layers, with respect to the transmitted messages, can be seen in Figure 9 for WSP POs and in Figure 10 for WTP POs. The events for

the WSP layer correspond to message exchanges between its WAE upper layer and its WTP lower layer. The set of interesting events that can be observed with WSP POs concerning the exchanges between WAE and WSP is given by the following primitives:

- *S-Connect* is used for a new session.
- *S-Disconnect* allows to abort all current methods or *push* transactions used to transfer data.
- *S-Suspend* is used to request a session to be suspended, so that no activity can occur on it until it is either resumed or disconnected.
- *S-Resume* allows a client to resume a suspended session. Session suspension will automatically abort all data transmission. This feature is useful in situations where a client notifies that it will be soon unavailable. For example, this is the case if the bearer network is not available due to roaming to another network or if the user switches off the device.

Each of these events may have one of the following attributes: *req*, *ind*, *res*, and *cnf*. In the case of exchanges between the WSP and WTP layers, the set of interesting messages is:

- *TR-Invoke* initiates a new transaction.
- *TR-Result* is used to send back a result of a previously initiated transaction.
- *TR-Abort* is used to abort an existing transaction.

The attributes of these events are again the same as mentioned above. The idea is that the WTP layer POs allow to observe the messages between WSP and WTP. Moreover, the messages exchanged between a client (a wireless terminal) and a gateway through the network are *PDUs* (Protocol Data Unit). For the sake of simplicity, in this section we abstract these exchanges. Intuitively, the *TR* primitives can be used as follows. A client C sends an *invoke* message. This message is considered in the protocol as *TR-Invoke.ind* and it is received by the gateway G. Afterwards, an acknowledgment *TR-Invoke.res* is sent. Then, G tries to get the page requested by the user. Once G gets the corresponding WML page, it sends a message to the client: *TR-Result.req*. The client receives this message as a *TR-Result.ind* event. Then, it sends an acknowledgment, denoted by the event *TR-Result.res*. Finally, G receives the event *TR-Result.cnf* denoting that the client received the requested information. This exchange of messages is graphically represented in Figure 10.

In order to test the protocol, we consider invariants (both simple and obligation) that correspond to properties extracted from the Wap Forum specification (see `http://www.wapforum.org`). These properties cover the main features of the communication protocol, that is, the connection, data transfer, and disconnection phases. The whole protocol was running autonomously and we were observing traces of 500 input/output pairs. Next we comment on the most relevant invariants that we studied. First, we describe an invariant for a connection establishment through a WTP class 2 transactions. This class, as said before,

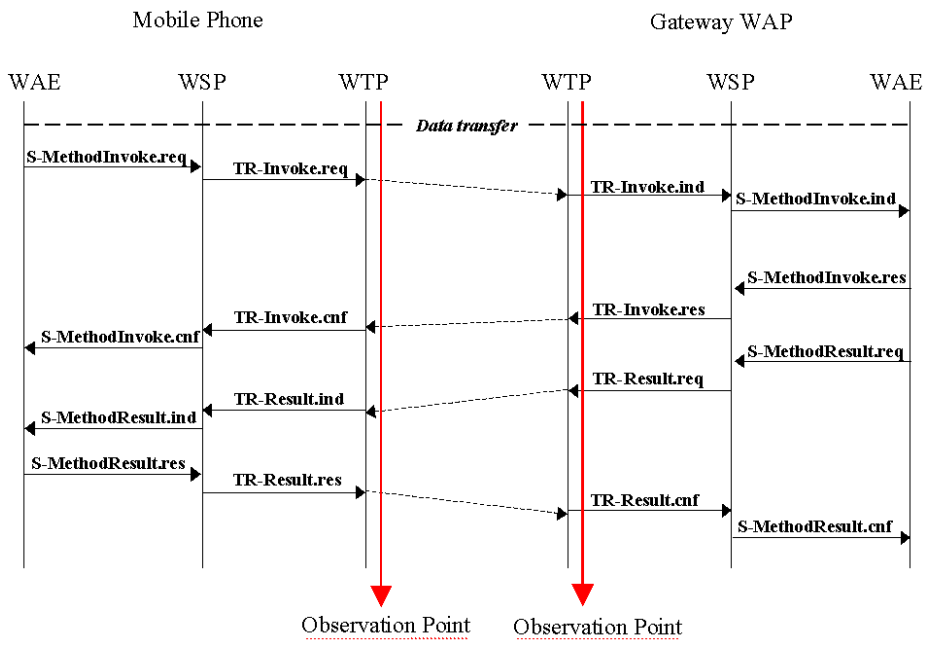

Figure 10. Messages between layers and WTP observation points.

provides the classical reliable request/response transaction common in many client/server scenarios. The invariant to check the connection is as follows:

$$I_1 = S - Connect.req/TR - Invoke.req, *, TR - Result.ind/O$$

where $O = \{S - Connect.cnf, S - Disconnect.ind\}$. Next, we present an invariant to check a connection proper release. Let us remark that a disconnection can happen also after the occurrence of a $TR - Result.ind$ primitive during a connecting session. Thus, once a connection has been established, we need to have a successful disconnection. This property is expressed by the following simple invariant:

$$I_2 = ?/S - Connect.cnf, *, TR - Invoke.ind/\{S - Disconnect.ind\}$$

Another important property of the WAP protocol concerns the resume of a connection. When a server resumes a connection, we must have that a suspension of a connection has been asked. In order to check this characteristic, the following obligation invariant can be used:

$$I_3 = S - Suspend.req/?, *, ?/\overline{\{S - Resume.ind\}}$$

The previous invariants are checked using WSP POs. Next, we present invariants related to the WTP POs. First, we give an interesting example which can lead to wrong conclusions if one is not careful.

Invariant	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
Verdict	True	True	True	False	True	True	True	True
Duration (seconds)	1.02	0.99	1	1.01	1.01	1.01	1.13	1.40

Figure 11. Checking of invariants on the specification

$$I_4 = \textit{TR-Invoke.req}/?, *, \textit{TR-Result.res}/\{\textit{TR-Result.cnf}\}$$

This invariant is useful to check that whenever the client (wireless terminal side) asks to download a WAP page, it is successfully received. More precisely, this is the way used by the WTP class 2 to acknowledge messages. 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 requested web page is not available. So, we *removed* some of the requested web pages and we found that the new observed trace did not respect the invariant. In fact, the correct invariant is

$$I_5 = \textit{TR-Invoke.req}/?, *, \textit{TR-Result.res}/\{\textit{TR-Result.cnf}, \textit{TR-Abort.ind}\}$$

If we work within WTP class 0, no acknowledges are sent after a message is received. In this case, in order to check that the gateway sends the data to the cell phone we have to consider the event *TR-Result.req*. Thus, in WTP class 0 we have the invariant

$$I_6 = \textit{TR-Invoke.req}/?, *, \textit{TR-Result.req}/\{\textit{TR-Result.ind}, \textit{TR-Abort.ind}\}$$

Let us note that this invariant also holds in WTP class 2.

The POs log files include a field called *handle*. The transaction *handle* is an index returned to the higher layer, that enables the higher layer to identify the transaction and to associate the data received with an active transaction. Actually, any user of the protocol is uniquely identified. So, we may use our invariants to check some privacy aspects of the protocol. For example,

$$I_7 = ?/\textit{TR-Invoke.ind}(\text{user1}), *, \ \textit{TR-Result.res}(\text{user1})/O$$

where $O = \{\textit{TR-Result.cnf}(\text{user1}), \textit{TR-Abort.ind}\}$. In the following, we use an obligation invariant to check that if a given user has received a confirmation it must happen that he was asking before for the corresponding page:

$$I_8 = \textit{TR-Invoke.req(user1)}/?, *, ?/\overline{\{\textit{TR-Result.cnf(user1)}\}}$$

Figures 11 and 12 report the results obtained for each of the invariants previously presented. All the experiments have been performed on Windows 2000 (Intel III Processor, 933MHz). Let us simply remark that in Figure 12, the time associated with the fourth invariant is (much) smaller because the process is stopped once a fault is detected.

Invariant	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
Verdict	True	True	True	False	True	True	True	True
Duration (seconds)	3.12	2.78	4.52	1.2	5.27	5.07	4.32	6.53

Figure 12. Checking of invariants on traces

6 Conclusions and Future Work

In this paper we have introduced a new methodology for passive testing. This methodology includes the definition of a novel concept of invariant as well as a corresponding test architecture to deal with them. Two types of invariants have been defined: simple and obligation invariants. They can express a wide range of properties. Algorithms checking the correctness of these invariants with respect to a given specification have been presented. We have introduced a software tool, called TestInv, that implements all the algorithms presented in this paper and helps in the automation of our passive testing with invariants approach. This tool has been integrated in the PLATONIS platform. In order to show the application of the proposed methodology on real protocols, a real case study, the Wireless Application Protocol, has been extensively studied.

We plan to continue our work on passive testing with invariants to the detection of errors in critical systems where active testing is not feasible. Another field of application that we are exploring is network security. We consider that our techniques are well adapted to detect anomalies, attacks and intrusions, but this point has to be further investigated. Finally, we would like to extend our invariants with capabilities so that not only the control part of protocols but also the data part can be taken into account. In this line, [LCH+02] represents a step forward in the passive testing methodology because data is formally considered. Following this direction, we have started a work to define algorithms for passive testing that takes into account the data part and improve the fault detection power.

Acknowledgments

We would like to thank José Antonio Arnedo for his involvement in the previous work on passive testing during his training period at GET-INT. We would also like to thank the anonymous reviewers for the careful reading and the useful suggestions that have helped to improve the quality of the paper.

References

[AAD79] J.M. Ayache, P. Azema, and M. Diaz. Observer: A concept for on-line detection of control errors in concurrent systems. In *9th Symposium on Fault-Tolerant Computing*, 1979.

[ACN03] J.A. Arnedo, A. Cavalli, and M. Núñez. Fast testing of critical properties through passive testing. In *TestCom 2003, LNCS 2644*, pages 295–310. Springer, 2003.

[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.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CGP03] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Journal of Information and Software Technology*, 45:837–852, 2003.

[CLRZ99] A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaïdi. Hit-or-Jump: An algorithm for embedded testing with applications to IN services. In *Formal Description Techniques for Distributed Systems and Communication Protocols (XII), and Protocol Specification, Testing, and Verification (XIX)*, pages 41–56. Kluwer Academic Publishers, 1999.

[ITU92] ITU. Recommendation Z.100: CCITT Specification and Description Language (SDL), 1992.

[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. In *10th IEEE Int. Conf. on Network Protocols, ICNP'02*, pages 122–131. IEEE Computer Society Press, 2002.

[LNS+97] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *5th 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.

[MA01] 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.

[Pla01] Platonis Consortium. The platonis project. In *Applications and Services in Wireless Networks, ASN'01*, pages 251–262. Hermes, 2001.

[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.

[Ver97] Verilog. ObjectGEODE Simulator, Reference Manual, 1997.

[WZY01] J. Wu, Y. Zhao, and X. Yin. From active to passive: Progress in testing of internet routing protocols. In *FORTE 2001*, pages 101–116. Kluwer Academic Publishers, 2001.