Passive Testing with Proxy-testers

Sébastien Salva
LIMOS - UMR CNRS 6158,
University of Auvergne,
Campus des Cézeaux, Aubière, France
sebastien.salva@u-clermont1.fr

Abstract

Passive testing is an alternative testing approach whose purpose is to passively analyze an implementation behaviour without disturbing it. Usually, passive testing methods extract traces by means of sniffer-based tools, running in the same environment as the implementation. Nevertheless, many implementation environments prevent from setting a sniffer-based tool for security or technical reasons. We propose a passive testing method based on the notion of proxy-tester which represents an intermediary between client applications and the implementation. We define a proxy-tester as a product between the specification and its canonical tester, which is able to receive the client traffic and to forward it to the implementation and vice versa. It also aims to analyze the implementation traces to detect faults. We define a non conformance relation between the implementation, its specification and the external environment from which is received the client traffic. We also provide some preliminary results on the Amazon E-commerce Web service and discuss about the proxy-tester benefits.

Keywords: passive testing; proxy-tester; STS; conformance relation.

1: Introduction

Testing is an important engineering activity widely used to find defects in systems or programs. In particular, black box testing, which is the topic of this paper, consists in testing whether an implementation behaves as described in its specification. Most testing methods are said to be active which means that test cases are extracted from the specification and experimented on the implementation to conclude whether a test relation is satisfied. For instance, *ioco* [14] is a well-know test relation for active testing, which is based on trace (action sequences) equivalence. Active methods require to deploy a pervasive test environment (test architecture) to execute test cases and to observe implementation reactions. They may also interrupt the system normal functioning arbitrarily, for example by resetting it after each test case execution. However, when a system is deployed in an integrated environment, it becomes quite difficult to access it. Moreover, active methods may disturb the natural operation of the implementation under test. So, these ones may not be suitable in regards to the tested system.

Passive testing represents another interesting alternative, which offers several advantages, e.g., to publish more rapidly a system or to not disturb it while testing. Passive testing

methods aim to detect faults by passively observing the implementation input/output actions, without interrupting its normal behaviour. Usually, the tester corresponds to a kind of sniffer which can extract both the stimuli sent to the implementation and its reactions in the environment where it is running. Then, the resulting traces can be used to check that the implementation behaviour does not contradict the specification one, or to check the satisfaction of specific properties defined by means of invariants.

Nevertheless, some hypotheses are also required for passive testing. In particular, a module, which often corresponds to a sniffer-based tool, has to be set in the implementation environment to observe its reactions. This assumption is difficult to maintain with many environments in practice: for instance, as soon as a system is deployed on an infrastructure which does not allow a test environment installation (testers, sniffer-based tools, etc.) for security reasons, the messages generated by the implementation cannot be extracted. Another recent example concerns Clouds, i.e. virtualized environments where the resources are not owned by the software development companies. The dynamic nature of the Cloud architecture does not enable a tester to retrieve the messages exchanged between applications since we do not know in advance where they are geographically deployed. Consequently, if these messages cannot be observed, the traces cannot be extracted and thus the test cannot be performed.

This paper proposes another passive testing solution, based on the notion of proxy-tester. This one is a standalone application which can be seen as an intermediary between client applications and the implementation under test. It aims to receive the client traffic that it forwards to the implementation and vice versa. While receiving implementation reactions, a proxy-tester is also able to detect incorrect behaviours and to conclude on the non conformance of the implementation. Moreover, proxy-testers offer a lot of flexibility since they can be deployed in the same environment as the implementation but also outside of it, in condition that proxy-testers may interact with the implementation. In the remainder of the paper, we formally define the notion of proxy-tester for symbolic systems. We also define a non conformance test relation between the implementation and its specification that we model with STS (Symbolic Transition System [5]). And finally, we present some of the possibilities offered by a proxy-tester, e.g., for security testing.

This paper is structured as follows: we briefly present, in Section 2, selected publications on passive testing, related to the topics covered in this paper. Section 3 defines the specification modelling. Section 4 describes our passive testing method by defining the proxy-tester of a specification and the corresponding passive conformance test relation. We provide some preliminary results on the Amazon E-commerce Web service and discussions in Section 5. Finally, we conclude in Section 6.

2: Related work on passive testing

Some works dealing with passive testing of protocols or components (Web services) have been proposed recently.

In [10], Lee et al. propose a passive testing approach dedicated to wired protocols modelled with Extended Finite State Machines (EFSM), composed of variables. Several algorithms on the EFSM model and their applications to OSPF and TCP state machine are presented. Algorithms aim to check whether traces, composed of actions and parameters, satisfy the specification on the fly. As is described in the experimentation part, reactions

are extracted by means of sniffers.

Some works deal with Mobile ad hoc network routing protocol testing. In [11], the passive testing approach is based on the notion of Relay Node Set (RNS) concept. A RNS is a set of nodes that allow reaching all nodes in the network. RNS passively computes some metrics from the packets passing through the node and from formulae constructed manually according to the tested protocol. Metrics may be based on the overhead of the network, or on the number of retransmissions etc. Based on the previous metrics, the protocol can be optimized by managing the node number for instance. In [3], the passive testing approach is based on the correctness of an implementation through a set of invariants (or properties) and traces. This approach is constructed by different steps: definition of invariants from the specification, extraction of execution traces by means of sniffers, verification of the invariants on the traces.

Other works focus on Web service testing: in this case, passive methods are used to check conformance or security. In [12], the confidence level between the implementation and its specification is also defined with invariants. As in [3], invariants are constructed from the specification and traces are collected with network sniffers. Then, the *TIP* tool performs automated analysis of the captured traces to determine if the given timed extended invariants are satisfied or not. Security of Web service compositions are passively tested in [2]. Security rules are modelled with the Nomad language which can express authorizations or prohibitions with timed properties. Firstly, a rule set is manually constructed from a specification. Traces of the implementation are extracted with modules which are placed at each workflow engine layer which executes Web services. Then, authors check that the implementation does not contradict the security rules with the collected traces.

To our knowledge, the passive testing methods proposed in literature (and especially the previous ones) rely on a sniffer-based tool as a central point to extract all the client requests and implementation reactions (messages, packets, etc.). In this paper, we consider that the access to the implementation environment is restricted. So, a sniffer-based tool cannot be set up in the environment. Instead, we define the notion of proxy-tester which is constructed from a specification and its canonical tester. We assume that the client traffic is routed to the proxy-tester which forwards it to the implementation and vice versa. It also analyzes the implementation reactions on the fly to detect non conformance. Non conformance is defined formally with a passive test relation which depends on an implementation, its specification and the external environment which stimulates (calls) the implementation.

3: Model Definition and notations

Several models, e.g., UML, Petri nets, process algebra, have been proposed to formalize systems or applications. The STS (Symbolic Transition Systems [5]) model is one of them and has been used with many testing methods [6, 7, 13]. The STS formalism offers a large formal background (definitions of implementation relations, test case generation algorithms, etc.). So, we based our choice on this latter to model specifications and test cases. An STS is a kind of input/output automaton extended with a set of variables, with guards and assignments on variables labelled on the transitions. In the following, we assume that STSs are deterministic.

Definition 1 *A Symbolic Transition System STS is a tuple* $< L, l_0, V, V_0, I, \Lambda, \rightarrow >$*, where:*

- *L is the finite set of locations, with l_0 the initial one,*
- *V is the finite set of internal variables, while I is the finite set of external or interaction ones. We denote D_v the domain in which a variable v takes values. The internal variables are initialized with the assignment V_0, which is assumed to take an unique value in D_V,*
- *Λ is the finite set of action labels, partitioned by $\Lambda = \Lambda^I \cup \Lambda^O$: inputs, beginning with ?, are provided to the system, while outputs (beginning with !) are observed from it,*
- *$\rightarrow$ is the finite transition set. A transition $(l_i, l_j, a(p), \varphi, \varrho)$, from the location $l_i \in L$ to $l_j \in L$, also denoted $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$ is labelled by $a(p) \in \Lambda \times \mathcal{P}(I)$, with $a \in \Lambda$ an action and $p \subseteq I$ is a finite set of external variables $p = (p_1, ..., p_k)$. We denote $type(p) = (t_1, ..., t_k)$ the type of the variable set p. $\varphi \subseteq D_V \times D_p$ is a guard which restricts the firing of the transition. Internal variables are updated with the assignment $\varrho : D_V \times D_p \rightarrow D_V$ once the transition is fired.*

An immediate STS extension is called the STS *suspension* which also expresses quiescence. A quiescence occurs when no output can be observed. The system is blocked unless the environment provides an input. Quiescence is modelled by a new symbol $!\delta$ and an augmented STS denoted $\Delta(STS)$. For an STS $\mathcal{S}$, $\Delta(\mathcal{S})$ is obtained by adding a self-loop labelled by $!\delta$ for each location where quiescence may be observed. The guard of this new transition must return true for each value of $D_{V \cup I}$ which does not allow firing a transition labelled by an output. An STS suspension example is illustrated in Figure 1. This straightforward specification describes a banking component with two methods *lg* for logging on the bank system and *transfer* whose purpose is to transfer money from one bank account to another one. The specification handles an internal and private method *valid* which returns true if the given bank accounts or keys are valid and false otherwise.

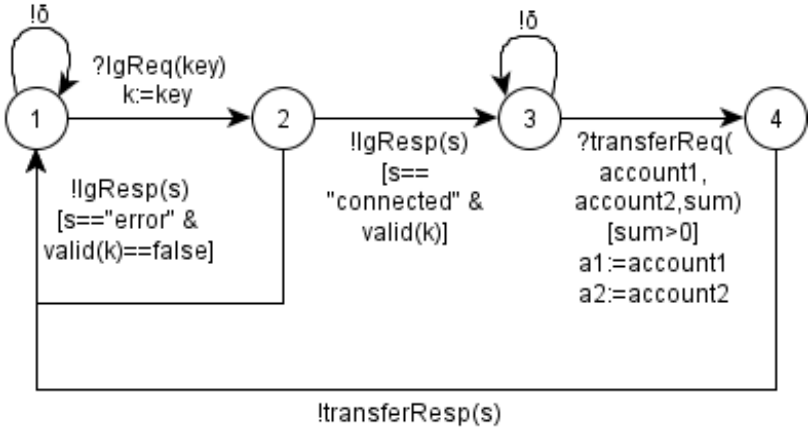

Figure 1. A suspension STS

An STS is also associated to an LTS (Labelled Transition System) to define its semantics. The LTS semantics corresponds to a valued automaton without symbolic variables: the LTS states are labelled by internal variable values while transitions are labelled by actions and parameter values.

Definition 2 *The semantics of an STS* $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ *is an LTS* $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$ *where:*

- $Q = L \times D_V$ *is the finite set of states,*
- $q_0 = (l_0, V_0)$ *is the initial state,*
- $\sum = \{(a(p), \theta) \mid a(p) \in \Lambda, \theta \in D_p\}$ *is the set of valued symbols,*
- $\rightarrow$ *is the transition relation* $Q \times \Sigma \times Q$ *deduced by the following rule:*

$$\frac{l_i \xrightarrow{a(p),\varphi,\varrho} l_j, \theta \in D_p, v \in D_V, v' \in D_V, \varphi(v,\theta) \models true, v' = \varrho(v,\theta)}{(l_i, v) \xrightarrow{a(p),\theta} (l_j, v')}$$

Intuitively, for an STS transition $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$, we obtain an LTS one $(l_i, v) \xrightarrow{a(p),\theta} (l_j, v')$ with v an internal variable value set, if there exists a parameter value θ such that the guard $\varphi(v, \theta)$ returns true. Once the transition is executed, the internal variables take the value v' derived from the assignment $\varrho(v, \theta)$. An STS suspension $\Delta(\mathcal{S})$ is associated to its suspension LTS semantics by $||\Delta(\mathcal{S})|| = \Delta(||\mathcal{S}||)$.

For defining STS-based conformance relations, an implementation under test is usually assumed to behave like an LTS. So, runs and traces can then be extracted during the test execution.

For an STS $\mathcal{S}$, interpreted by its LTS semantics $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$, a run $q_0\alpha_0...\alpha_{n-1}q_n$ is an alternate sequence of states and valued actions. $RUN(\mathcal{S}) = RUN(||\mathcal{S}||)$ is the set of runs found in $||\mathcal{S}||$. $RUN_F(\mathcal{S})$ is the set of runs of $\mathcal{S}$ finished by a state in $F \subseteq Q$.

It follows that a trace of a run is defined as a projection on actions. So, $Traces_F(\mathcal{S}) = Traces_F(||\mathcal{S}||)$ is the set of traces of runs finished by states in $F \subseteq Q$.

4: Passive testing with proxy-tester

4.1: Proxy-tester definition

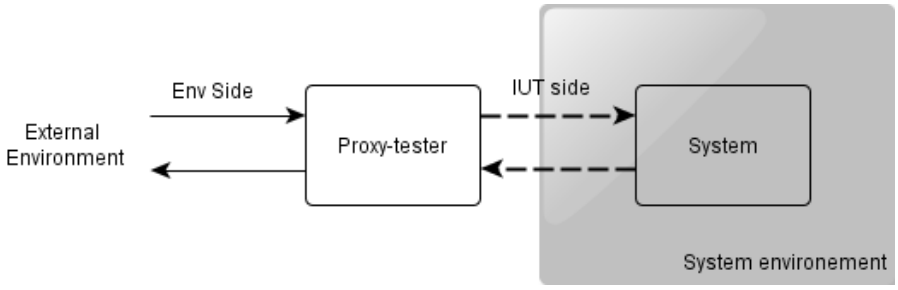

Figure 2. The proxy-tester use

This paper proposes an alternative passive testing method by defining a proxy-tester which can be seen as a kind of intermediary between the external environment (client side) and the implementation. Figure 2 illustrates an example of a proxy-tester use. This one may be deployed in the implementation environment or outside of it, in condition that it may interact with the implementation. From the external side, it replaces (conceals) the implementation so this requires the client traffic to be routed to the proxy-tester.

Thus, a proxy-tester must interact with the external environment as it is described in the specification. Each action received from the external environment by the proxy-tester is forwarded to the implementation as well. It must be also able to receive actions from the implementation which are forwarded to the external environment, but which can be also analyzed to check properties and especially the implementation conformance.

So, to be defined as an intermediary between the external environment and the implementation, the proxy-tester must behave like the specification for interacting with the external environment side and must also act as a mirror specification to interact with the implementation side. To analyze the implementation actions or more precisely the implementations traces, it must also recognizes the correct traces (those found in the specification) and the incorrect ones. Consequently, we can deduce that a proxy-tester must be a combination of the specification with a mirror specification blent with incorrect behaviours. And this second part corresponds to the canonical tester of the specification.

The canonical tester of an STS, gathers the specification transitions labelled by mirrored actions (inputs becomes outputs and vice versa) and transitions leading to a new location $Fail$, modelling the receipt of unspecified actions [8].

Let $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}}>$ be a deterministic STS and $\Delta(\mathcal{S})$ be its suspension. The canonical tester for $\mathcal{S}$ is the deterministic STS $CAN(\mathcal{S}) =< L_{CAN}, l0_{CAN}, V_S, V0_S, I_S, \Lambda_{CAN}, \rightarrow_{CAN}>$ such that:

- $\Lambda^I_{CAN} = \Lambda^O_S \cup \{?\delta\}$ and $\Lambda^O_{CAN} = \Lambda^I_S$,
- $L_{CAN}, l0_{CAN}, \rightarrow_{CAN}$ are defined by the rules:

(keep $\mathcal{S}$ transitions): $\dfrac{t \in \rightarrow_{\Delta(\mathcal{S})}}{t \in \rightarrow_{CAN}}$

(incorrect behaviour completion): $\dfrac{a \in \Lambda^O_{\mathcal{S}} \cup \{!\delta\}, \varphi_a = \bigwedge_{l_1 \xrightarrow{a(p),\varphi_n,\varrho_n}_{\Delta(\mathcal{S})} l_n} \neg\varphi_n}{l_1 \xrightarrow{?a(p),\varphi_a,\emptyset}_{CAN} Fail}$

The canonical tester of the specification, given in Figure 1, is illustrated in Figure 3. For readability reason, locations are identified by letters but it is not mandatory. The resulting STS is completed on the input set. For instance, from the location B, new transitions to $Fail$ are added to model the receipt of unspecified responses or quiescence.

In the proxy-tester, to clearly separate the external environment side to the implementation one, we separate the variable set of $\mathcal{S}$ to the variable set of $CAN(\mathcal{S})$ with the renaming function $\phi : V \cup I \rightarrow V' \cup I'$, $\phi(v) \rightarrow v'$. So, for an STS $\mathcal{S}$, we denote $\phi(\mathcal{S}) =< L_{\mathcal{S}}, l0_{\mathcal{S}}, \phi(V_{\mathcal{S}}), \phi(V0_{\mathcal{S}}), \phi(I_{\mathcal{S}}), \Lambda_S, \rightarrow_{\phi(\mathcal{S})}>$ where $\rightarrow_{\phi(\mathcal{S})}$ is the finite transition set composed of transitions
$l \xrightarrow{a(p),\phi(\varphi),\phi(\varrho)}_{\phi(\mathcal{S})} l'$ with $\phi(\varphi) \subseteq D_{\phi(V)} \times D_{\phi(p)}$ and $\phi(\varrho) : D_{\phi(V)} \times D_{\phi(p)} \rightarrow D_{\phi(V)}$. In the following proxy-tester definition, we also add, for each transition, an internal variable *side* which helps to clearly identify the interactions with the external environment ($side := Env$) from the interactions with the implementation under test ($side := IUT$).

A proxy-tester $\mathcal{P}(\mathcal{S})$ of the specification $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}}>$ is a combination of $\Delta(\mathcal{S})$ with its canonical tester $\phi(CAN(\mathcal{S}))$. $\mathcal{P}(\mathcal{S})$ is defined by a deterministic STS $< L_{\mathcal{P}}, l0_{\mathcal{P}}, V_{\mathcal{S}} \cup V_{\phi(\mathcal{S})} \cup \{side\}, V0_{\mathcal{S}} \cup V0_{\phi(\mathcal{S})} \cup \{side := ""\}, I_{\mathcal{S}} \cup I_{\phi(\mathcal{S})}, \Lambda_{\Delta(\mathcal{S})} \cup \Lambda_{CAN}, \rightarrow_{\mathcal{P}}>$

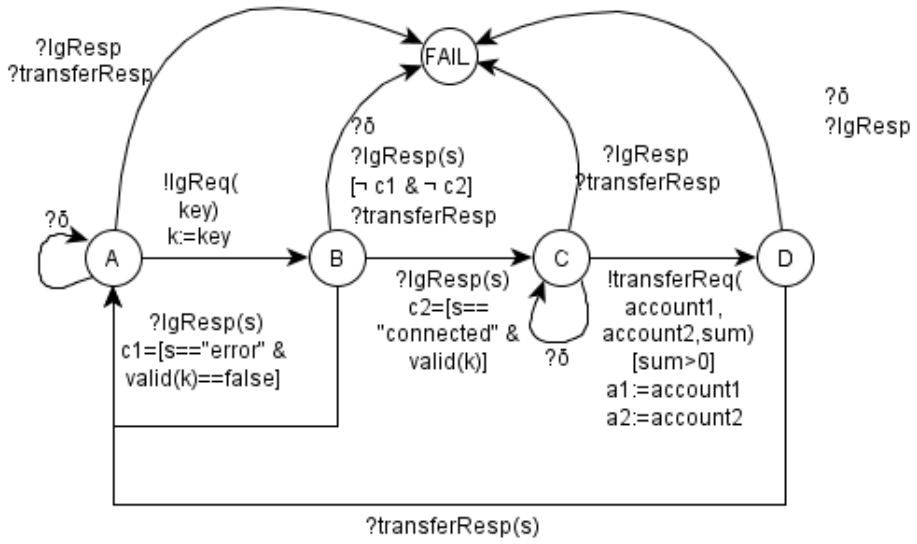

Figure 3. A canonical tester

such that $L_{\mathcal{P}}, l0_{\mathcal{P}}$ and $\rightarrow_{\mathcal{P}}$ are constructed by the following inference rules:

(Env to IUT):
$$l_1 \xrightarrow{?a(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l_2, l_1 \xrightarrow{!\delta}_{\Delta(\mathcal{S})} l_1, l_{1'} \xrightarrow{!a(p'),\varphi',\varrho'}_{CAN} l_{2'},$$
$$l_{1'} \xrightarrow{?\delta}_{CAN} l_{1'}, \varphi' = \phi(\varphi), \varrho' = \phi(\varrho)$$
$$\vdash$$
$$(l_1 l_{1'}) \xrightarrow{?a(p),\varphi,[\varrho \wedge temp:=p \wedge side:=Env]}_{\mathcal{P}} (l_2 l_{1'} a\varphi)$$
$$\xrightarrow{!a(p'),[p'==temp \wedge \varphi'],[\varrho' \wedge side:=IUT]}_{\mathcal{P}} (l_2 l_{2'}),$$
$$(l_2 l_{1'} a\varphi) \xrightarrow{?\delta,\emptyset,[side:=IUT]}_{\mathcal{P}} (l_2 l_{1'} \delta) \xrightarrow{!\delta,\emptyset,[side:=Env]}_{\mathcal{P}} (l_2 l_{1'} a\varphi)$$

(Env to IUT)$_2$:
$$l_1 \xrightarrow{?a(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l_2, l_{1'} \xrightarrow{!a(p'),\varphi',\varrho'}_{CAN} l_{2'},$$
$$l_{1'} \xrightarrow{b(p''),\varphi_b,\varrho_b}_{CAN} Fail, \varphi' = \phi(\varphi), \varrho' = \phi(\varrho)$$
$$\vdash$$
$$(l_2 l_{1'} a\varphi) \xrightarrow{b(p''),\varphi_b,[\varrho_b \wedge side:=IUT]}_{\mathcal{P}} Fail$$

(IUT to Env):
$$l_1 \xrightarrow{!a(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l_2, l_{1'} \xrightarrow{?a(p'),\varphi',\varrho'}_{CAN} l_{2'},$$
$$l_{1'} \xrightarrow{b(p''),\varphi_b,\varrho_b}_{CAN} Fail, \varphi' = \phi(\varphi), \varrho' = \phi(\varrho)$$
$$\vdash$$
$$(l_1 l_{1'}) \xrightarrow{?a(p'),\varphi',[\varrho' \wedge temp:=p' \wedge side:=IUT]}_{\mathcal{P}} (l_1 l_{2'} a\varphi)$$
$$\xrightarrow{!a(p),[p==temp \wedge \varphi],[\varrho \wedge side:=Env]}_{\mathcal{P}} (l_2 l_{2'}),$$
$$(l_1 l_{1'}) \xrightarrow{b(p''),\varphi_b,\varrho_b}_{\mathcal{P}} Fail$$

The first rule (Env to IUT) combines a specification transition and a canonical tester one labelled by the same mirrored actions to express that if an action is received from

the external environment then this one is spread to the implementation. The two steps (receiving an action and forwarding it) are separated by a new location ($l_2 l_{1'} a\varphi$) which is unique for each transition of the specification since this one and its canonical tester are deterministic (for any action a and any pair of transitions carrying a with two guards φ_1 and φ_2, $\varphi_1 \wedge \varphi_2$ is unsatisfiable). Transitions labelled by δ modelling quiescence are also combined: so if quiescence is detected from the implementation, quiescence is also observed from the external environment. Note that a suspension STS location has inevitably a self-loop transition labelled by $!\delta$ if a transition labelled by an input may be fired from the same location. The second rule (Env to IUT)$_2$ adds eventually the canonical tester transitions leading to $Fail$. The last rule (IUT to Env) similarly combines a specification transition and a canonical tester one labelled by the same mirrored actions to express that if an action is received from the implementation then this one is spread to the external environment. In this case, we always have a canonical tester transition leading to $Fail$ which is added to the proxy-tester (at least a transition labelled by $?\delta$ expressing quiescence). For each rule, the resulting transitions are identified by means of the *side* internal variable.

The resulting proxy-tester obtained from the previous specification (Figure 1) and its canonical tester (Figure 3) is depicted in Figure 4. For sake of readability, the *side* variable is replaced with solid and dashed transitions: solid transitions stand for interactions with the external environment ($side := Env$), dashed ones for interactions with the implementation ($side := IUT$). Figure 4 clearly illustrates that the initial specification behaviour (paths) is kept and that the incorrect behaviour depicted in the canonical tester is present too.

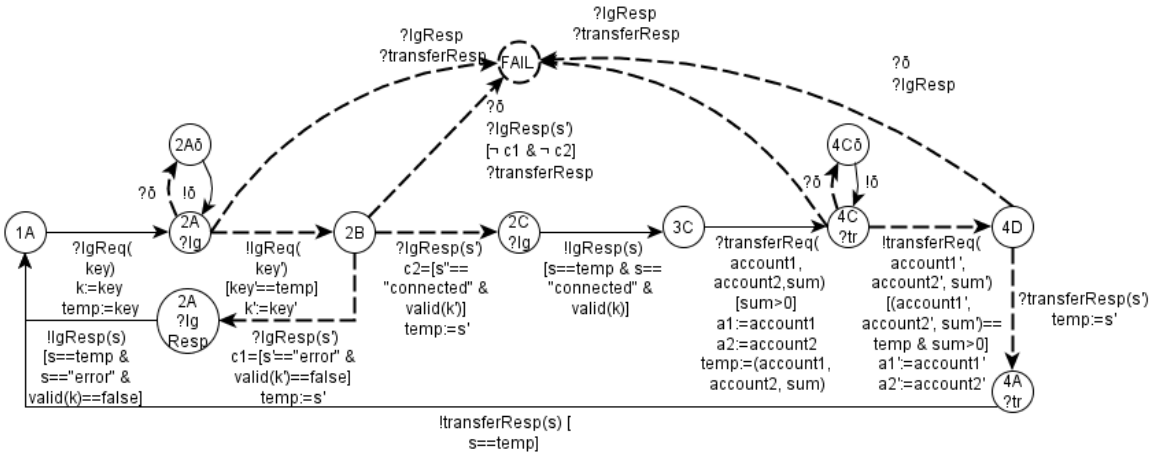

Figure 4. A proxy-tester

The transitions representing interactions with the external environment are separated to the other ones by means of the *side* variable. As a consequence, runs and traces of the proxy-tester can be also separated.

Let $\mathcal{P} =< L_\mathcal{P}, l0_\mathcal{P}, V_\mathcal{P}, V0_\mathcal{P}, I_\mathcal{P}, \Lambda_\mathcal{P}, \rightarrow_\mathcal{P} >$ be an STS and $||\mathcal{P}|| = P =< Q_P, q0_P, \sum_P, \rightarrow_P >$ be its LTS semantics. We define $Side : Q_P \rightarrow D_{V_\mathcal{P}}$ the mapping which returns the value of the *side* variable of a state. (STS internal variable values are located in LTS states).

Let $RUN(\mathcal{P})$ be the set of runs of $\mathcal{P}$. We define $RUN^E(\mathcal{P})$ as the projection $proj_{(q\alpha q_1 \in RUN(\mathcal{P}), Side(q)=E)}(RUN(\mathcal{P}))$ which denotes the set of partial runs $q_i\alpha_i ... \alpha_{n-1} q_n$ relative to the E side. It follows that $Traces^E(\mathcal{P})$ is the set of traces of partial runs in

$RUN^E(\mathcal{P})$.

For a proxy-tester $\mathcal{P}(\mathcal{S})$, we can write $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ for representing the non conformance traces extracted by the proxy-tester from the implementation side. For instance, in the proxy-tester of Figure 4, !lgReq("a incorrect key").?lgResp("connected") belongs to $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$.

With these notations, we can also write some interesting trace properties.

Proposition 3 *The traces of the canonical tester are equal to the traces given by the transitions of the proxy-tester labelled by the assignment side := IUT. So, we have* $Traces^{IUT}(\mathcal{P}(\mathcal{S})) = Traces(CAN(\mathcal{S}))$

Proof sketch: for simplicity, we give only the intuition of proof here (the complete proof requires to consider all the cases described in the proxy-tester construction rules i.e. transitions with inputs, outputs, quiescence, transitions leading to $Fail$, etc.). This equality can be proved by focusing on the proxy-tester construction rules. These ones keep the path structure of both the specification and its canonical tester by interleaving one transition of the specification with its mirrored transition and by adding the transitions modelling incorrect behaviours of the canonical tester. Let $\mathcal{S}$ be an STS, and $\mathcal{P}(\mathcal{S})$ be its proxy-tester, which is a combination of $\Delta(\mathcal{S})$ with its canonical tester $\phi(CAN(\mathcal{S}))$. If we suppose that we have two transitions $l_1 \xrightarrow{a(p)}_{\Delta(\mathcal{S})} l_2 \xrightarrow{b(p)}_{\Delta(\mathcal{S})} l_3$ and $l'_1 \xrightarrow{a(p')}_{\phi(CAN(\mathcal{S}))} l'_2 \xrightarrow{b(p')}_{\phi(CAN(\mathcal{S}))} l'_3$. Either of the rules produce a path $l_1 l'_1 \xrightarrow{?a(p)}_{\mathcal{P}(\mathcal{S})} l_{t1} \xrightarrow{!a(p)}_{\mathcal{P}(\mathcal{S})} l_2 l'_2 \xrightarrow{?b(p)}_{\mathcal{P}(\mathcal{S})} l_{t2} \xrightarrow{!b(p)}_{\mathcal{P}(\mathcal{S})} l_3 l'_3$, where l_{t1} and l_{t2} are unique. Transitions labelled by δ in $\Delta(\mathcal{S})$ are joined in the same way, and transitions leading to $Fail$ in $CAN(\mathcal{S})$ are added in the proxy-tester $\mathcal{P}(\mathcal{S})$. Thereby, the path structure (behaviours of the suspension specification and of its canonical tester) is kept.

Consequently, for a run $r = q_0\alpha_0...\alpha_{n-1}q_n$ of the LTS semantics of $\phi(CAN(\mathcal{S}))$, there exists a run r' of the LTS semantics P of $\mathcal{P}(\mathcal{S})$ such that $proj_{(q\alpha q_1 \in r', Side(q)=IUT)}\ (r') = q'_i\alpha_0...\alpha_{n-1}q'_m$. It follows that $Traces^{IUT}($
$\mathcal{P}(\mathcal{S})) = Traces(\phi(CAN(\mathcal{S}))) = Traces(CAN(\mathcal{S}))$. The last equality is straightforward since ϕ is a mapping which renames variables in $CAN(\mathcal{S})$ for the proxy-tester definition. However, traces are not constructed over variable names but with values. We can also deduce that we have the same trace set leading to $Fail$. So, $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces_{Fail}(CAN(\mathcal{S}))$.

4.2: Passive conformance

To reason about conformance, the implementation under test is assumed behaving like its model and is represented by an LTS I. $\Delta(I)$ represents its suspension LTS. Active testing methods usually define the confidence degree of the implementation I with its specification $\mathcal{S}$ by means of a test relation. For instance, *ioco* is a well known test relation based on trace equivalence, dedicated for systems modelling with either LTSs or STSs. In [8], *ioco* is defined for STSs by:

$$I\ ioco\ \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap NC\_Traces(\Delta(\mathcal{S})) = \emptyset$$

where $NC\_Traces = Traces(\Delta(\mathcal{S})).(\sum^O \cup \{!\delta\} \backslash Traces(\Delta(\mathcal{S}))$ is the minimal non-conformant trace set. Nevertheless, *ioco* is not well suited for passive testing since the test depends on

the external environment stimuli which is not expressed. Moreover, passive testing does not help to check conformance but rather detects non conformance (defects) of the implementation on the received stimuli set during a long period of time. We prefer defining the following relation, based on *ioco*:

$$I \text{ non-conform } \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap NC\_Traces(\Delta(\mathcal{S})) \neq \emptyset$$

However, $NC\_Traces(\mathcal{S})$ is equivalent to $Traces_{Fail}(CAN(\mathcal{S}))$ since $CAN(\mathcal{S})$ recognizes non-conformant behaviours in its $Fail$ states [8]. And previously, we have shown that $Traces_{Fail}(CAN(\mathcal{S})) = Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$. So, we can write:

$$I \text{ non-conform } \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) \neq \emptyset$$

To take into account the passive test execution, we need to reason about the parallel composition of the External environment, the proxy-tester and the implementation. $P =< Q_P, q0_P, \sum_P, \rightarrow_P >$ is the LTS semantics of a proxy-tester $\mathcal{P}(\mathcal{S})$. We assume that the external environment can be modelled with an LTS $Env =< Q_{Env}, q0_{Env}, \sum_{Env} \subseteq \sum_P, \rightarrow_{Env} >$, and $I =< Q_I, q0_I, \sum_I \subseteq \sum_P, \rightarrow_I >$ is the implementation. The passive testing of I is modelled by the parallel composition $||(Env, P, I) =< Q_{Env} \times Q_P \times Q_I, q0_{Env} \times q0_P \times q0_I, \sum_{Env} \subseteq \sum_P, \rightarrow_{||(Env,P,I)} >$ where the transition relation $\rightarrow_{||(Env,P,I)}$ is defined by the following rules. For readability reason, we denote an LTS transition $q_1 \xrightarrow[E]{?a} q_2$ when $Side(q_1) = E$ (a variable *side* is valued to E in q_1).

(Env to IUT): $$\frac{q_1 \xrightarrow{!a}_{\Delta(Env)} q_2, q_2'' \xrightarrow{?a}_{\Delta(I)} q_3'', q_1' \xrightarrow[Env]{?a}_P q_2' \xrightarrow[IUT]{!a}_P q_3'}{q_1 q_1' q_2'' \xrightarrow[Env]{?a}_{||(Env,P,I)} q_2 q_2' q_2'' \xrightarrow[IUT]{!a}_{||(Env,P,I)} q_2 q_3' q_3''}$$

(IUT to Env): $$\frac{q_2 \xrightarrow{?a}_{\Delta(Env)} q_3, q_1'' \xrightarrow{!a}_{\Delta(I)} q_2'', q_1' \xrightarrow[IUT]{?a}_P q_2' \xrightarrow[Env]{!a}_P q_3', q_3' \neq Fail}{q_2 q_1' q_1'' \xrightarrow[IUT]{?a}_{||(Env,P,I)} q_2 q_2' q_2'' \xrightarrow[Env]{!a}_{||(Env,P,I)} q_3 q_3' q_2''}$$

(IUT to Fail): $$\frac{q_2 \xrightarrow{?\delta}_{\Delta(Env)} q_3, q_1'' \xrightarrow{!a}_{\Delta(I)} q_2'', q_1' \xrightarrow[IUT]{?a}_P Fail}{q_2 q_1' q_1'' \xrightarrow[IUT]{?a}_{||(Env,P,I)} Fail}$$

The immediate deduction of the $\rightarrow_{||(Env,P,I)}$ definition is that $\rightarrow_{||(Env,P,I)}$ is exactly composed of the same transition number as $\rightarrow_P$ and keeps the same symbol set. Only states are modified.

According to the previous rules, it follows that the trace set of $||(Env, P, I)$ is equal to the one of P. Since $P = ||\mathcal{P}(\mathcal{S})||$, $Traces(P)$ is equivalent to $Traces(\mathcal{P}(\mathcal{S}))$ (Section 3) and to $Traces(||(Env, P, I))$.
In particular, $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces^{IUT}_{Fail}(||(Env, P, I))$. Finally, we can also write:

$$\begin{aligned} I \text{ non-conform } \mathcal{S} &\Leftrightarrow Traces(\Delta(I)) \cap Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) \neq \emptyset \\ &\Leftrightarrow Traces(\Delta(I)) \cap Traces^{IUT}_{Fail}(||(Env, P, I)) \neq \emptyset \end{aligned}$$

In other terms, this relation means that the implementation I does not conform to its specification if a trace of the suspension implementation is found in a trace of the parallel composition $||(Env, P, I)$ leading to one of its Fail states.

In practice, to detect non conformance of the implementation, we assume that the client traffic is routed to the proxy-tester. We also assume that the proxy-tester instance configuration is the same as the implementation one. For instance, if the implementation is a multi-instance system (one implementation instance per client), the proxy-tester mode must be identical.

During its execution, the proxy-tester receives valued requests from both the external environment and the implementation under test. It covers its transitions until it reaches a Fail state. In this case the implementation is non conforming. Algorithm 1, derived from the one given in [10], details the proxy-tester functioning. The latter considers both the external environment and the implementation as LTS suspensions. The covering of the proxy-tester transitions is based on the notion of *Configuration* which reminds the LTS semantics states. A configuration $C =< l, Assert_l >$ is a pair composed of a location l and of an assertion $Assert_l$ which is a set expressing the current variable state. This set may be composed of variable guards and assignments. The complete definition of an assertion is given in [10]. The proxy-tester starts from its initial configuration i.e. the couple (initial location, initial internal variable value set). Upon a received valued event $e(p)$ (line 2), which is either an action with values or quiescence, it checks wether a next transition may be fired (lines 5-6): this one must have the same start location than the current configuration C, the same action than the received event $e(p)$ and $Eval(\varphi, Assert_l)$ must not return $false$. This subroutine evaluates the guard of the transition along with the assertion $Assert_l$ of the current configuration. Then, the algorithm computes the next configuration with the $Merge$ subroutine (lines 9-10). If the $Fail$ location is reached then the algorithm ends and returns Fail (lines 11-12). A trace of the implementation belongs to $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ therefore the implementation is non-conform to its specification. Otherwise, the event $e(p)$ is forwarded with the next proxy-tester transition t' either to the implementation or to the external environment, depending on the *side* value found in the assignment ϱ' of t', denoted $side(t')$ for simplicity in the algorithm (line 14). The new corresponding configuration is computed (lines 15-16). Then, the proxy-tester waits for the next event. Both subroutines $Merge$ and $Eval$ are discussed and detailed in [10].

5: Experimentation and discussion

We experimented our method on the Amazon E-commerce Web service [1] whose specification part is illustrated in Figure 6. We chose this application because we tested it in [13] and detected some bugs that we have reused in this experimentation to check whether a proxy-tester is able to detect them too.

We have implemented a tool which computes an STS proxy-tester from an STS specification. Then, we coded the proxy-tester manually, from Algorithm 1, as a Web service and deployed it on a Web server. For simplicity, we have only considered the string variable type and have replaced the *Eval* subroutine by the Hampi solver [9] which takes strings as inputs. The whole architecture is given in Figure 5. Finally, from known bugs of the implementation, we prepared some test sequences to simulate client requests and executed them with the SOAPUI tool [4], which is a unit testing tool for Web services. These sequences have been constructed to call the proxy-tester instead of the Amazon Web service. For each test sequence, the proxy-tester has returned Fail as expected.

We also performed this experimentation to check the feasibility of this method. Fig-

Algorithm 1: Proxy-tester algorithm

input : A proxy-tester $\mathcal{P}(\mathcal{S})$
output: Fault detected

1 //initialize the proxy-tester to its initial Configuration $C :=< l0_{\mathcal{P}(\mathcal{S})}, V0_{\mathcal{P}(\mathcal{S})} >$;
2 **while** *Event(e(p))* **do**
    `// possible next Configuration`
3 $C' = \emptyset$;
    `// possible next firable transition`
4 $tnext := \emptyset$;
5 **foreach** $t = l \xrightarrow{?e(p),\varphi,\varrho}_{\mathcal{P}(\mathcal{S})} l_{next} \in \rightarrow_{\mathcal{P}(\mathcal{S})}$ **do**
6 **if** $C ==< l, Assert_l >$ & $Eval(\varphi, Assert_l) \neq false$ **then**
7 $tnext := t$;
8 break;
9 $Assert_{l_{next}} := Merge(Assert_l, \varphi, \varrho)$;
10 $C' =< l_{next}, Assert_{l_{next}} >$;
11 **if** $l_{next} == Fail$ **then**
12 return Fail;
13 **else**
        `// event forwarded to the right side`
14 Execute($t' = l_{next} \xrightarrow{!e(p),\varphi',\varrho'}_{\mathcal{P}(\mathcal{S})} l_{next2}$) ; `// send !e(p) to` $side(t')$
15 $Assert_{l_{next2}} := Merge(Assert_{l_{next}}, \varphi', \varrho')$;
16 $C =< l_{next2}, Assert_{l_{next2}} >$;

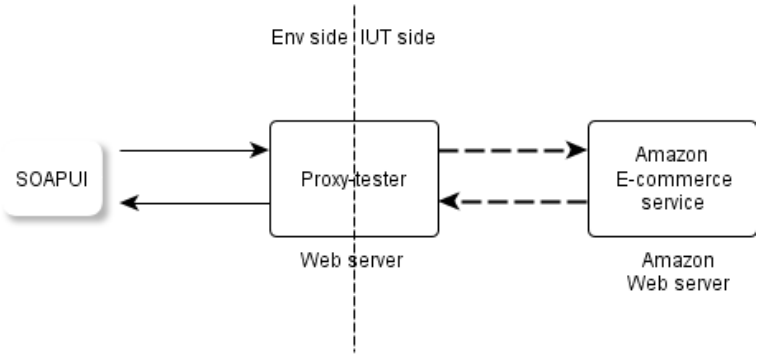

Figure 5. The experimentation architecture

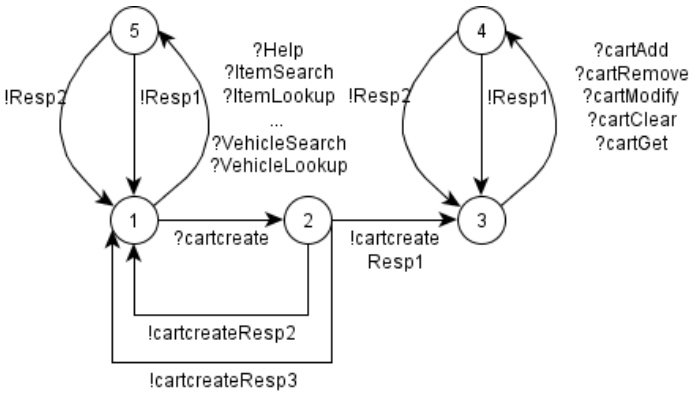

Figure 6. A part of the the Amazon E-commerce Web Service specification

ure 7 gives the location and transition numbers obtained for the Amazon Web service. Even though the proxy-tester transition number grows rapidly, this one is finite since it corresponds to the combination of the specification and of its canonical tester whose transition number is finite too. The STS proxy-tester can be computed rapidly (at most some minutes).

This experimentation showed that proxy-testers offer a very interesting alternative to classical sniffer-based tools since they detect non conformance of the implementation without invariants, even if the environment access is restricted.

A proxy-tester can also offer much more possibilities (and future works) than those described in the paper. Below, we propose some of them:

	Location nb	Transitions nb	Transitions to Fail nb
Specification	24	67	0
Canonical tester	25	202	133
Proxy-tester	49	269	133

Figure 7. Statistics on the Amazon Web Service

- *Component-based systems testing in restricted environments:* such systems could be passively tested by means of a set of proxy-testers interconnected to one another. If we assume that we have one specification modelling the whole system, we need to extract one sub-specification per component to generate a proxy-tester for each. The main issues concern the proxy-tester synchronization or the recomposition of the partial traces collected by proxy-testers. It is also manifest that the test relation must be redefined,
- *Combination with invariant-based methods:* it is still possible to extract implementation traces with proxy testers. So, passive testing methods which check the satisfaction of invariants can be used in combination with proxy-testers when the environment access is restricted. For instance, once traces are collected with a proxy-tester on Web services, the tool described in [2] can be executed to check the satisfaction of security rules,
- *Security testing and protection:* an interesting proxy-tester advantage is the separation of the events received from the external environment to those produced by the implementation. On the one hand, this separation may help to protect a system from attacks received from the external environment. And on the other hand, the proxy-tester can also check whether the system behaviour respects a security rule set in the meantime. Intuitively both protection rules and security rules could be defined with STSs: the first rules have to be synchronized with transitions composed of the assertion $side := Env$ modelling interactions with the external environment, while security rules must be synchronized with the other transitions. The result corresponds to a specialized application firewall combined with a security passive testing tool,
- *System quality improvement:* a proxy-tester can be seen as a kind of upper layer which encompasses the implementation. So, we could augment the proxy-tester model so that it could responds instead of the implementation, for example if this one has crashed. A modified proxy-tester could analyze the implementation responses and could modify its behaviour at runtime to improve some of its properties such as its observability, or to replace incorrect action responses.

6: Conclusion

A sniffer-based passive testing method cannot be applied on systems deployed in environments whose access is restricted for security or technical reasons. We believe that the use of a proxy-tester, which represents an intermediary between the external environment and the implementation under test, is an interesting alternative. So, we have defined the notion of proxy-tester in this paper and have defined the *non-conform* test relation expressing non conformance only since passive testing methods are not exhaustive.

An immediate line of future work is to extend the notion of proxy-tester for both security testing and protection. We have to define security rules with STSs for protecting and for testing. Then, these rules must be synchronized with a proxy-tester. It is also manifest that a new test relation must be defined.

References

[1] Amazon. Amazon e-commerce service. 2010. http://docs.amazonwebservices.com/AWSEcommerceService/4-0/.

[2] Ana Cavalli, Azzedine Benameur, Wissam Mallouli, and Keqin Li. A Passive Testing Approach for Security Checking and its Practical Usage for Web Services Monitoring. In *NOTERE 2009*, 2009.

[3] Ana Cavalli, Stephane Maag, and Edgardo Montes de Oca. A passive conformance testing approach for a manet routing protocol. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 207–211, New York, NY, USA, 2009. ACM.

[4] Eviware. Soapui. 2011. http://www.soapui.org/.

[5] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[6] Lars Frantzen, Jan Tretmans, and René de Vries. Towards model-based testing of web services. In Antonia Bertolino and Andrea Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 67–82, Palermo, Sicily, ITALY, June 9th 2006.

[7] ir. H.M. Bijl van der, Dr.ir. A. Rensink, and Dr.ir. G.J. Tretmans. Component based testing with ioco., 2003.

[8] Thierry Jéron. Symbolic model-based test selection. In P. Machado, A. Andrade, and A. Duran, editors, *Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2008), Salvador, Bahia, Brazil*, pages 17–32, 2008.

[9] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[10] David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, and Xia Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Trans. Netw.*, 14:424–437, April 2006.

[11] Tao Lin, S.F. Midkiff, and J.S. Park. A framework for wireless ad hoc routing protocols. In *Wireless Communications and Networking, 2003. WCNC 2003, New Orleans, LA, USA*. IEEE society press, 2003.

[12] Gerardo Morales, Stéphane Maag, Ana R. Cavalli, Wissam Mallouli, Edgardo Montes de Oca, and Bachar Wehbi. Timed extended invariants for the passive testing of web services. In *ICWS'10*, pages 592–599, 2010.

[13] Sébastien Salva and Issam Rabhi. Stateful web service robustness. In *ICIW '10: Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, pages 167–173, Washington, DC, USA, 2010. IEEE Computer Society.

[14] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

Authors

Sébastien Salva graduated in M.Sc. degree in Computer Science from the University of Versailles in 1999 and received his PhD from the University of Reims, France in 2001. Dr. Salva is now associate professor at the University of Auvergne and a fellow of the CNRS LIMOS laboratory. His research interests are on testing methodologies for conformance and robustness testing, active testing and monitoring techniques, the validation of security properties and their application to services and protocols. For more information, visit his web page at `http://sebastien.salva.free.fr`