Passive testing of symbolic systems - A ioco proxy-tester based approach

Sébastien Salva, LIMOS CNRS, UMR 6158,
PRES Clermont University, Campus des Cézeaux
Aubière, FRANCE
sebastien.salva@u-clermont1.fr

Abstract—This paper, dealing with model-based testing, proposes to formalize the concept of ioco proxy-tester for passively testing the conformance of implementations. This proxy oriented approach is particularly interesting when the implementation environment access is restricted, for security or technical reasons. In this case, a proxy-tester can be installed outside of this environment to interact with the implementation. It represents an intermediary application which receives the client traffic, forwards it to an implementation and vice-versa. While passively observing the implementation reactions, it can also detect its incorrect behaviours. We present, in the paper, a formal ioco proxy-tester definition that can be used to automatically generate one proxy-tester from a ioSTS specification. The notion of conformance is defined with the ioco relation. To directly check whether the implementation is ioco-conforming to its specification by means of a proxy-tester, we rephrase the ioco relation by defining sets of partial traces extracted from the proxy-tester. We also introduce other possibilities offered by the proxy-tester concept. In particular, we describe an approach which improves the functional quality of the implementation by completing its proxy-tester.

Index Terms—*passive testing, ioSTS, proxy tester, ioco*

1 Introduction

Testing is a widely known engineering activity whose purpose is to try to detect defects in systems. Many aspects may be tested e.g., correctness, robustness, performance, security, etc. This paper addresses conformance testing, which aims to check that a black box implementation, only accessible through its interfaces, behaves as described in its specification. Conformance testing is more and more performed by means of model-based approaches which rely upon a behavioural model to describe the specification and upon formal methods to improve the quality of testing. In this paper, we focus on models called input/output Symbolic Transition Systems (ioSTS) to describe symbolic systems with automata composed of input and output actions, extended with variables, guards and affectations.

In the context of model-based testing, conformance is often defined by means of a test relation between the specification model and the implementation. One commonly used test relation, for ioSTS, is the input output conformance (ioco) relation [1], which defines conformance as a partial inclusion of external behaviours (suspension traces) of the implementation in those of the specification. The test relation is usually checked with active testing methods: basically, these ones extract test cases from a specification and execute them successively on the implementation to detect errors. Active methods require to deploy a pervasive test environment (test architecture) to execute test cases and to observe implementation reactions. They also need to interrupt the system normal functioning arbitrarily, for example by resetting it after each test case execution. These requirements cannot be always fulfilled according to the tested system. For instance, if it consists of several modules, it may become difficult to interrupt it.

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. A passive tester observes the execution traces provided by the implementation, without experimenting it. Then, traces are analyzed to detect defects. This approach offers a large domain of applications: it can be used to observe or control network congestion, to detect configuration issues and to monitor a system or an application to detect errors. In this last case, passive testing methods aim at detecting faults by means of a tester which is often composed of a sniffer module. It is assumed that it can extract both the stimuli sent to the implementation and the reactions of the latter. Then, the resulting traces are analyzed to check that the implementation behaviour does not contradict the specification one, or to check the satisfiability of specific properties.

Passive testing is not a new approach. But, the new trends in computer science such as service or Cloud computing raise new issues to extract traces. Indeed, the typical assumption, of setting a sniffer-based tool in the implementation environment to observe its reactions, is difficult to maintain with many Web based environments in practice. For instance, when

applications are deployed on infrastructures which do not allow a test environment installation (testers, sniffer based tools, etc.) for security reasons, the messages generated by these applications cannot be extracted. The situation is identical with Clouds, i.e., virtualized environments where the resources are not owned by the software development companies. The dynamic nature of the Cloud architecture prevents from installing a sniffer-based module to retrieve the messages exchanged between applications, since we do not know in advance where they are geographically deployed. When the environment access is restricted, another possibility is to place sniffers in client environments. But this solution is very difficult to realize in practice: either clients are unknown in advance thus the installation of sniffers is impossible, or some clients are known but each client environment must be modified.

This paper proposes another passive testing solution, based on the notion of proxy-tester, for applications modelled with ioSTS. A proxy-tester corresponds to an intermediary between client applications and the implementation under test. It can be deployed in the same environment as the implementation but also outside of it, in condition that it may interact with the implementation. A proxy-tester receives the client traffic that it forwards to the implementation and vice-versa. While receiving the traffic of both the implementation and the client sides, it aims at detecting incorrect behaviours of the implementation. The notion of correct or incorrect implementation is defined by means of the ioco test relation in the paper. We begin to formally define the proxy-tester of a ioSTS specification. Thereafter, we rephrase the ioco relation to take into consideration the proxy-tester trace set. We show that the traces, extracted from the proxy-tester, are sufficient to check if the implementation is ioco-conforming to its specification.

As stated previously, proxy-testers represent intermediate applications which relay events from client and implementation sides. This notion of relay offers many other possibilities. We present some of them at the end of the paper. In particular, we propose an approach which completes proxy-testers, to reply to the client side whenever an error is detected, or to compensate the error.

The paper is structured as follows: Section 2 provides details on our main motivations and related works on the passive testing area. Section 3 defines the specification modelling. Section 4 describes our passive testing method by defining the proxy-tester of a specification and by proposing an execution algorithm. We provide preliminary results on the Amazon E-commerce Web service and discussions in Section 5. We present some proxy-tester extensions in Section 6. Finally, we conclude in Section 7.

2 Related work on passive testing

Several works, dealing with passive testing of protocols or components, have been proposed recently. For all of these, the tester is composed on a kind of sniffer-based module located in the implementation environment, and collects trace sets. However, the use of these sets may be different:

- Invariant satisfiability [2], [3], [4]: invariants represent properties which are always true. These invariants are constructed from the specification and are later checked on the collected traces. This approach helps to test complex properties but has the inconvenient to require to construct invariants manually. This approach gave birth to several works. For instance, the passive testing method, presented in [4], aims to test the satisfiability of invariants on Mobile ad-hoc network routing protocols. Different steps are required: definition of invariants from the specification, extraction of execution traces with sniffers, verification of the invariants on the traces.
  Other works focus on component testing: in this case, passive methods are used to check conformance or security. For instance, the *TIP* tool [5] performs automated analysis of captured trace sets to determine if a given set of timed extended invariants are satisfied. As in [4], invariants are constructed from the specification and traces are collected with network sniffers. Cavalli et al. propose an approach for testing passively the security of Web service compositions [6]. Security rules are here 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, the method checks, with the collected traces, that the implementation does not contradict the security rules,
- Forward checking [7], [8], [9]: traces are given on the fly to an algorithm which aims to check the implementation correctness by covering the specification while observing the implementation reactions. In this field, Lee and al. propose a passive testing method dedicated to wired protocols [9]. These protocols are modelled with Extended Finite State Machines (EEFSM), composed of variables. Several algorithms on the EEFSM model and their applications to the OSPF protocol and TCP state machines are presented. Algorithms check whether partial traces (not beginning from the initial implementation state), composed of actions and parameters, satisfy the symbolic specification on the fly. The coverage of the symbolic specification is performed by means of *configu-*

rations. A configuration represents a tuple which consists of the current state and of a set of assignment and guards modelling the variable state. As is described in the experimentation part, reactions are extracted by means of sniffers,

- Backward checking [10]: this approach processes a partial trace backward to narrow down the possible specifications. The algorithm performs two steps. It first follows a given trace backward, from the current configuration to a set of starting ones, according to the specification. The goal is to find the possible starting configurations of the trace, which leads to the current configuration. Then, it analyzes the past of this set of starting configurations, also in a backward manner, seeking for configurations in which the variables are determined. When such configurations are reached, a decision is taken on the validity of the studied paths (traces are completed). Such an approach is usually applied as a complement to forward checking to detect more errors.

The passive testing methods, proposed in literature, rely on a kind of sniffer-based tool as a central point to extract all the implementation reactions (messages, packets, etc.). This tool must be located inside the same environment as the implementation. As stated earlier, we have more and more implementation environments whose access is restricted for technical or security reasons.

This paper tackles this issue, by supposing that we do not have any privilege on environments and by focusing on the transparent proxy concept for ioco testing. We initially worked on the proxy-tester concept in [11] for deterministic systems only. And the obtained proxy-testers did no check ioco-conformance. In this paper, we completely redefine the proxy-tester concept and propose new algorithms. Our main contribution is to formally define the notion of ioco proxy-tester, constructed automatically from an ioSTS specification which may be undeterministic. It analyzes the implementation reactions on the fly to detect non-conformance. Conformance is expressed with the $ioco$ relation, that we reformulate by means of proxy-tester trace set. We also provide a proxy-tester algorithm which can detect passively whether an implementation is not ioco-conforming to its specification.

3 Model Definition and Notations

Several models e.g., UML, Petri nets, process algebra have been proposed to formally describe systems, applications or components. The STS model (Symbolic Transition Systems [12]) is one of them. An STS is a kind of automata model which is extended with a set of variables and with transition guards and assignments, giving the possibilities to express the system state and constraints on actions. This model also offers a large formal background (definitions of algebraic based operations, of implementation relations, test case generation algorithms, etc.). It has been also used with many testing methods [13], [14], [15], [16].

Beforehand, we assume that there exist a domain of values denoted D, a variable set X taking values in D and a set of terms over X denoted $T(X)$. The assignment of values of a set of variables $Y \subseteq X$ is denoted by valuations where a valuation is a function $v: Y \to D$. We denote D_Y the set of valuations over the set of variables Y. Two valuations $v \in D_Y$ and $w \in D_Z$ with $Y \cap Z = \emptyset$ can be combined with $(v \cup w)x =_{def} v(x)$ if $x \in Y$ or $(v \cup w)x =_{def} w(x)$ if $x \in Z$. Similarly, terms in $T(X)$ can be evaluated with a function $v: T(X) \to D$. The set of first order formula G over $Y \subseteq X \cup T(X)$ is denoted $\mathcal{F}(Y)$. A formula G may be satisfied with respect to a valuation v, which is denoted $v \models G$.

In this paper, we use an extended model, called ioSTS, which separates the action set with inputs beginning by ? to express the actions expected by the system, and with outputs beginning by ! to express actions produced (observed) by the system. Inputs of a system can only interact with outputs provided by the system environment and vice-versa. An ioSTS is also input-enabled, i.e., it always accepts any of its inputs. So, outputs of the environment are never rejected.

Definition 1 (ioSTS): An Input Output Symbolic Transition System (ioSTS) is a tuple $< L, l_0, V, V_0, I, \Lambda, \to >$, 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. The internal variables are initialized with the valuation $V_0 \in D_V$, which is assumed to be unique,
- Λ is the finite set of symbols, partitioned by $\Lambda = \Lambda^I \cup \Lambda^O$: Λ^I (Λ^O) represents the set of input symbols beginning with ? (the set of output symbols beginning with ! respectively).
- $\to$ is the finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \in L$ to $l_j \in L$, also denoted $l_i \xrightarrow{a(p),G,A} l_j$ is labelled by an (input or output) action $a(p) \in \Lambda \times \mathcal{P}(I)$, with $a \in \Lambda$ a symbol and $p \subseteq I$ a finite set of interaction variables $p = (p_1, ..., p_k)$. $G \in \mathcal{F}(p \cup V \cup T(p \cup V))$ is a guard which restricts the firing of the transition. Internal variables are updated with the assignment function $A: D_V \times D_p \times D_{T(p \cup V)} \to D_V$ once the transition is fired.

The distinction made between internal and external variables is convenient to clearly express the state of the system itself (internal variables) and to model complex actions composed with communication parameters (external variables).

The ioSTS *suspension* is an immediate ioSTS exten-

sion, which also expresses quiescence i.e., the absence of observation. Modelling quiescence in specifications is useful to express authorised blocking states and is later convenient to identify the authorised blocking states of the implementation to those obtained from a failure. Quiescence is modelled by a new symbol $!\delta$ and an augmented ioSTS denoted $\Delta(ioSTS)$. For an ioSTS $\mathcal{S}$, $\Delta(\mathcal{S})$ is obtained by adding a self-loop labelled by $!\delta$ for each location where quiescence may be observed.

Definition 2 (ioSTS suspension): For an ioSTS $\mathcal{S} = < L,\ l_0,\ V, V_0, I, \Lambda,\ \rightarrow >$, its suspension is the ioSTS $\Delta(\mathcal{S}) =< L, l_0, V, V_0, I, \Lambda \cup \{!\delta\},\ \rightarrow_{\Delta(\mathcal{S})} >$ where $\rightarrow_{\Delta(\mathcal{S})}$ is given by the following rule:

$$\frac{l_1 \in L, a \in \Lambda^O, G_a = \bigvee\limits_{l_1 \xrightarrow{a(p),G,A} \rightarrow l_2} \exists v \in D_p).G}{l_1 \xrightarrow{!\delta,G',A'} \rightarrow_{\Delta(\mathcal{S})} l_1, G' = \bigwedge\limits_{a \in \Lambda^O} \neg G_a,\ A' = (x := x)_{x \in V}}$$

An ioSTS 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 the client bank account to another one. This specification is composed of the symbol set $\Lambda = \{!\delta, ?lgReq, !lgResp, ?transferReq, !transferResp\}$. It also handles a term $valid$ which returns true if the given bank accounts or keys are valid and false otherwise. Quiescence may be observed from locations 1, 3 and 4 since there exists a valuation which does not satisfy the firing of a transition labelled by an output action, from these locations. For example, we obtain a self-loop labelled $!\delta$ and by the guard $valid(a) = false$ which is deduced from the previous definition with $\neg(\exists\exists v \in D_p)(s = "transfered" \wedge valid(a))$. This specification is also undeterministic since, from the location 4, we have two transitions labelled by the same action whose guards are not mutually exclusive.

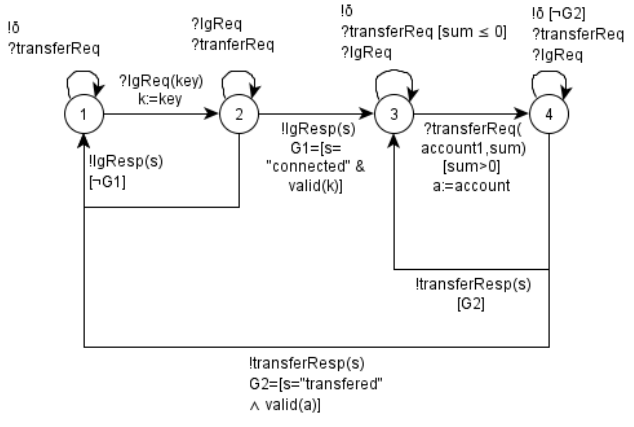

Fig. 1. An ioSTS suspension

An ioSTS is also associated to an ioLTS (Input/Output Labelled Transition System) to formulate its semantics. Intuitively, the ioLTS semantics corresponds to a valued automaton without symbolic variable, which is often infinite: the ioLTS states are labelled by internal variable valuations while transitions are labelled by actions and interaction variable valuations.

Definition 3 (ioSTS semantics): The semantics of an ioSTS $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ is an ioLTS $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$ where:

- $Q = L \times D_V$ is the finite set of states,
- $q_0 = (l_0, V_0)$ is the initial state,
- $\sum = \{(a(p), \theta) \mid a(p) \in \Lambda \times \mathcal{P}(I), \theta \in D_p\}$ is the set of valued symbols. $\sum^I$ is the set of input actions and $\sum^O$ is the set of output ones,
- $\rightarrow$ is the transition relation $Q \times \Sigma \times Q$ deduced by the following rule:

$$\frac{l_1 \xrightarrow{a(p),G,A} l_2, \theta \in D_p, v \in D_V, v' \in D_V, v \cup \theta \models G, v' = A(v \cup \theta))}{(l_1, v) \xrightarrow{a(p),\theta} (l_2, v')}$$

This rule can be read as follows: for an ioSTS transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain an ioLTS transition $(l_1, v) \xrightarrow{a(p),\theta} (l_2, v')$ with v a valuation over the internal variable set, if there exists a valuation θ such that the guard G returns true with $v \cup \theta$. Once the transition is executed, the internal variables are assigned with v' derived from the assignment $A(v \cup \theta)$. An ioSTS suspension $\Delta(\mathcal{S})$ is also associated to its suspension ioLTS semantics by $||\Delta(\mathcal{S})|| = \Delta(||\mathcal{S}||)$.

Some behavioural properties can now be defined on ioSTS in terms of their semantics, in particular runs and traces.

Definition 4 (Runs and traces): For an ioSTS $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$, interpreted by its ioLTS 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 $F \times D_V \subseteq Q$ with F a location of L.

It follows that a trace of a run r is defined as the projection $proj_{\sum}(r)$ on actions. $Traces_F(\mathcal{S}) = Traces_F(||\mathcal{S}||)$ is the set of traces of runs finished by states in $F \times D_V$.

For instance, $?lgReq("incorrectkey")!lgResp("error")$ is a trace obtained from the banking system of Figure 1 when a connection was attempted with an incorrect key.

4 Passive testing with proxy-tester

4.1 Proxy-tester definition

This paper proposes an alternative passive testing method by defining a ioco proxy-tester, based on the notion of transparent proxy which corresponds to a passive intermediary between the external environment (client side) and the implementation.

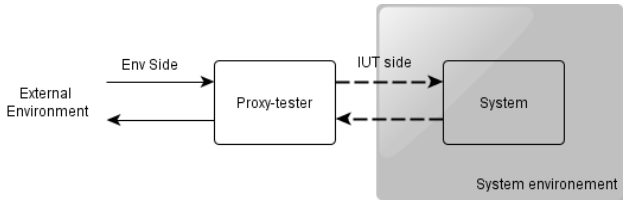

Fig. 2. The proxy-tester use

Figure 2 illustrates a test architecture example, composed of a proxy-tester. This one may be deployed in the implementation environment or outside of it, in condition that it may still interact with the implementation. Firstly, it must interact with the external environment as is described in the specification. Since it does not behave instead of the implementation, each action received from the external environment by the proxy-tester is forwarded to the implementation. It must be also able to receive actions from the latter which are then forwarded to the external environment, but which are also analyzed to check the implementation conformance.

Therefore, 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 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. It ensues that a proxy-tester must be a combination of the specification with a mirror specification blent with incorrect behaviours. This second part corresponds to a canonical tester of the specification.

The canonical tester of an ioSTS, gathers the specification transitions labelled by mirrored actions (inputs become outputs and vice versa) and transitions leading to a new location $Fail$, modelling the receipt of unspecified actions. Actually, canonical testers usually represent deterministic test cases where input and output actions are mirrored to stimulate the implementation waiting for an action and to observe its reactions. For our passive testing method, determinism is not a required hypothesis though. So, to avoid any ambiguity, we prefer using the term *ioSTS reflection*.

Definition 5 (ioSTS reflection): 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 an ioSTS and $\Delta(\mathcal{S})$ be its suspension. The ioSTS reflection for $\mathcal{S}$ is the ioSTS $REF(\mathcal{S}) =< L_{REF}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{REF}, \rightarrow_{REF} >$ such that:

- $\Lambda^I_{REF} = \Lambda^O_{\Delta(\mathcal{S})}$ and $\Lambda^O_{REF} = \Lambda^I_{\Delta(\mathcal{S})}$ (the alphabet is mirrored),
- $L_{REF}, \rightarrow_{REF}$ are defined by the rules.

These rules construct an ioSTS composed of transitions with mirrored actions and completed with transitions to a Fail location guarded by the negation of the union of guards of the same output action in outgoing transitions.

(keep transitions): $\mathcal{S} \quad \frac{t \in \rightarrow_{\Delta(\mathcal{S})}}{t \in \rightarrow_{REF}}$

(incorrect behaviour completion): $\frac{a \in \Lambda^O_{\mathcal{S}} \cup \{!\delta\}, l_1 \in L_{\mathcal{S}}, G_a = \bigwedge_{l_1 \xrightarrow{a(p),G,A} \Delta(\mathcal{S}) l} \neg G}{l_1 \xrightarrow{?a(p),G_a,A_a=(x:=x)_{x \in V}} {}_{REF} Fail}$

For example, the reflection of the specification, given in Figure 1, is illustrated in Figure 3. If we consider the location 2, new transitions to $Fail$ are added to model either the receipt of the unspecified response ?transferResp or quiescence.

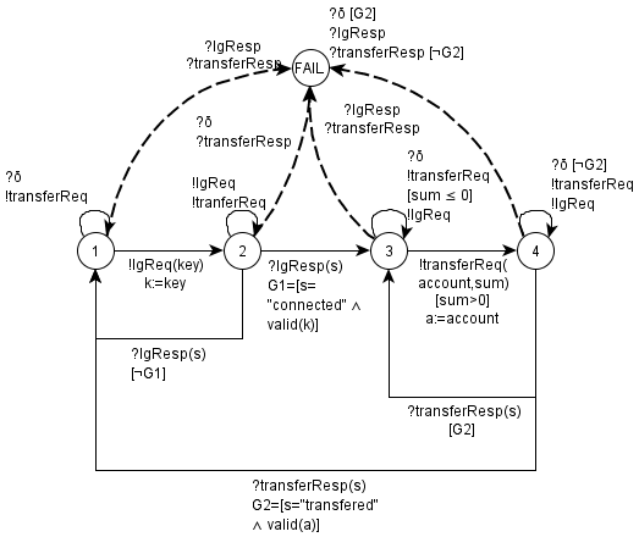

Fig. 3. An ioSTS reflection

Since $REF(\mathcal{S})$ is constructed by exchanging inputs and outputs symbols of its specification, its trace set is not composed to the traces of $\Delta(\mathcal{S})$ but of its mirrored traces. If we define $refl : (\sum^*)^* \rightarrow (\sum^*)^*$ the function which constructs a mirrored trace set from an initial one (for each trace, input symbols are exchanged with output ones and vice-versa), we can write that $Traces(\Delta(\mathcal{S})) \subseteq refl(Traces(REF(\mathcal{S})))$.

In the sequel, we will say that an ioSTS reflection recognizes incorrect behaviours of its initial ioSTS specification. More precisely, if we denote $NC\_Traces(\Delta(\mathcal{S})) = Traces(\Delta(\mathcal{S})).(\sum^O \cup \{!\delta\} \setminus Traces(\Delta(\mathcal{S}))$, the non-conformant traces of $\Delta(\mathcal{S})$, we can deduce directly from the rule (incorrect behaviour completion) that:

Proposition 6: The non-conformant trace set of $\Delta(\mathcal{S})$, denoted $NC\_Traces(\Delta(\mathcal{S}))$, is equal to $refl(Traces_{Fail}(REF(\mathcal{S})))$.

Now, we are ready to define the proxy-tester of an ioSTS $\mathcal{S}$ which is a combination of $\mathcal{S}$ with its ioSTS reflection. To identify without ambiguity the external environment side to the implementation one in proxy-testers, we separate variables, locations, guards and assignments of $\mathcal{S}$ and $REF(\mathcal{S})$ with a renaming function. For simplicity, we use the same function ϕ. Internal and interaction variables of $REF(\mathcal{S})$ are renamed with the function $\phi : V \cup I \rightarrow V' \cup I'$, $\phi(v) \rightarrow v'$. Locations are renamed by $\phi : L \rightarrow L'$, $\phi(l) \rightarrow l'$ and so on.

For an ioSTS $\mathcal{S}$, we also denote $\phi(\mathcal{S}) =< \phi(L_\mathcal{S}), \phi(l0_\mathcal{S}), \phi(V_\mathcal{S}), \phi(V0_\mathcal{S}), \phi(I_\mathcal{S}), \Lambda_S, \rightarrow_{\phi(\mathcal{S})} >$ where $\rightarrow_{\phi(\mathcal{S})}$ is the finite transition set composed of transitions $l'_1 \xrightarrow{a(p'),G',A'}_{\phi(\mathcal{S})} l'_2$ with $G' = \phi(G)$ a guard in $\mathcal{F}(p' \cup V' \cup T(p' \cup V'))$ and $A' = \phi(A)$ a variable assignment $A' : V' \times p' \times T(p' \cup V') \rightarrow V'$.

In the following proxy-tester definition, we also add, for each transition, an assignment of the variable denoted $side$, which helps to clearly identify the interactions with the external environment ($side := Env$) from the interactions with the implementation under test ($side := IUT$). We can now formalize the notion of proxy-tester:

Definition 7 (Proxy-tester): 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 reflection $\phi(REF(\mathcal{S}))$. $\mathcal{P}(\mathcal{S})$ is defined by an ioSTS $< L_\mathcal{P}, l0_\mathcal{P}, V_\mathcal{S} \cup \phi(V_\mathcal{S}) \cup \{side\}, V0_\mathcal{S} \cup \phi(V0_\mathcal{S}) \cup \{side := ""\}, I_\mathcal{S} \cup \phi(I_\mathcal{S}), \Lambda_{\Delta(\mathcal{S})} \cup \Lambda_{REF}, \rightarrow_\mathcal{P} >$ such that $L_\mathcal{P}, l0_\mathcal{P}$ and $\rightarrow_\mathcal{P}$ are constructed by the following inference rules:

(Env: to IUT)
$$l_1 \xrightarrow{?a(p),G,A}_{\Delta(\mathcal{S})} l_2, l_{1'} \xrightarrow{!a(p'),G',A'}_{REF} l_{2'},$$
$$l'_1 = \phi(l_1), G' = \phi(G), A' = \phi(A)$$
$$\vdash$$
$$(l_1 l_{1'}) \xrightarrow{?a(p),G,A(\{p_t:=p, side:=Env\})}_\mathcal{P} (l_2 l_{1'} ?aGA) \xrightarrow{!a(p'),[p'=\phi(p_t)],A'(\{side:=IUT\})}_\mathcal{P} (l_2 l_{2'}),$$

(IUT: to Env)
$$l_1 \xrightarrow{!a(p),G,A}_{\Delta(\mathcal{S})} l_2, l_{1'} \xrightarrow{?a(p'),G',A'}_{REF} l_{2'},$$
$$l'_1 = \phi(l_1), G' = \phi(G), A' = \phi(A)$$
$$\vdash$$
$$(l_1 l_{1'}) \xrightarrow{?a(p'),G',A'(\{p_t:=p', side:=IUT\})}_\mathcal{P} (l_1 l_{2'} ?aGA) \xrightarrow{!a(p),[p=\phi^{-1}(p_t)],A(\{side:=Env\})}_\mathcal{P} (l_2 l_{2'}),$$

(to: Fail)
$$l_{1'} \xrightarrow{b(p),G,A}_{REF} Fail, l_1 \in L_\mathcal{S}, l'_1 = \phi(l_1)$$
$$\vdash$$
$$(l_1 l_{1'}) \xrightarrow{b(p),G,A(\{side:=IUT\})}_\mathcal{P} Fail$$

Intuitively, the first rule (Env to IUT) combines a specification transition and an ioSTS reflection one carrying the same mirrored actions and guards to express that if an action is received from the external environment then it is spread to the implementation. The two transitions are separated by a unique location $(l_2 l_{1'} ?aGA)$. 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. The second rule (IUT to Env) similarly combines a specification transition and an ioSTS reflection one labelled by the same mirrored actions to express that if an action is received from the implementation then it is spread to the external environment. The last rule (to Fail) completes the resulting ioSTS with the transition leading to Fail of the ioSTS reflection.

As ioSTS specifications are assumed input-enabled and according to the previous definition, a proxy-tester accepts any output provided by the external environment. A proxy-tester is also constructed from the ioSTS reflection of the specification which is completed on the incorrect behaviour set: hence, both the ioSTS reflection and the proxy-tester accept any output provided by the implementation. Consequently, deadlocks can only occur in proxy-testers when one of its final states is reached, in particular one of its Fail states.

Figure 4 depicts the resulting proxy-tester obtained from the previous specification (Figure 1) and its reflection (Figure 3). 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 transitions for interactions with the implementation ($side := IUT$). We have also reduced the proxy-tester with left-right arrows to illustrate the composition of self-loop transitions. For instance, the left-right arrow given in location 33′ corresponds to the transitions depicted in Figure 5. Figure 4 clearly illustrates that the initial specification behaviours are kept and that the incorrect behaviours modelled in the ioSTS reflection are present as well.

The transitions representing interactions with the implementation are identified by means of the variable $side$. As a consequence, specific properties on runs and traces of the proxy-tester can also be defined. In particular, we can define partial runs and traces over the variable $side$.

Definition 8 (Partial runs and traces): Let $\mathcal{P}(\mathcal{S}) =< L_\mathcal{P}, l0_\mathcal{P}, V_\mathcal{P}, V0_\mathcal{P}, I_\mathcal{P}, \Lambda_\mathcal{P}, \rightarrow_\mathcal{P} >$ be a proxy-tester and $||\mathcal{P}(\mathcal{S})|| = P =< Q_P, q0_P, \sum_P, \rightarrow_P >$ be its ioLTS semantics. We define $Side : Q_P \rightarrow D_{V_\mathcal{P}}$ the mapping which returns the value of the $side$ variable of a state in Q_P. $Side_E(Q_P) \subseteq Q_P$ is the set of states $q \in Q_P$ such that $Side(q) = E$.

Let $RUN(\mathcal{P}(\mathcal{S}))$ be the set of runs of $\mathcal{P}(\mathcal{S})$. We denote $RUN^E(\mathcal{P}(\mathcal{S}))$ the set of partial runs derived from the projection $proj_{Q_P \sum_P Side_E(Q_P)}(RUN(\mathcal{P}(\mathcal{S})))$.

It follows that $Traces^E(\mathcal{P}(\mathcal{S}))$ is the set of partial traces of (partial) runs in $RUN^E(\mathcal{P}(\mathcal{S}))$.

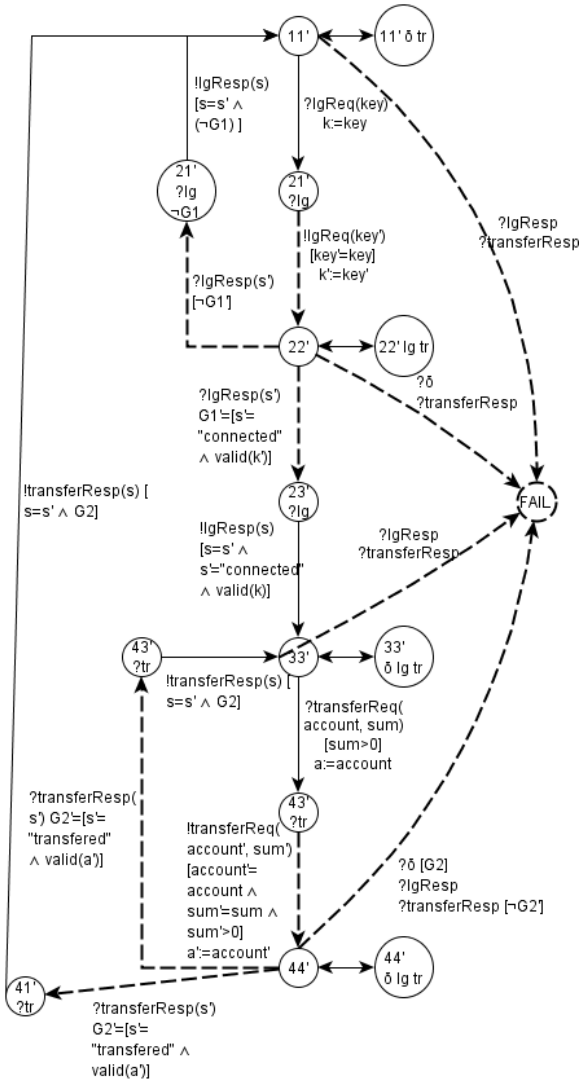

Fig. 4. A proxy-tester

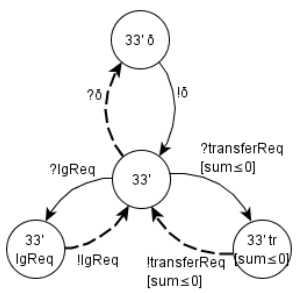

Fig. 5. A detailed part of the proxy-tester

For a proxy-tester $\mathcal{P}(\mathcal{S})$, we can now write $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ for representing the non-conformant trace set extracted by the proxy-tester from the implementation side only. For instance, in the proxy-tester of Figure 4, $!lgReq("incorrectkey")?\delta$ belongs to $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$.

With these notations, we can deduce an interesting trace property on proxy-testers. We can write that the incorrect behaviours expressed in the ioSTS reflection with $Traces_{Fail}(REF(\mathcal{S}))$ still exist in the proxy-tester and are expressed by the trace set $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$.

Proposition 9: Let $\mathcal{S}$ be a specification and $REF(\mathcal{S})$, $\mathcal{P}(\mathcal{S})$ be its reflection and its proxy-tester respectively. We have $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces_{Fail}(REF(\mathcal{S}))$.

The proof is given in Appendix A.

4.2 Passive conformance

To reason about conformance, the implementation under test is assumed to behave like its specification and is modelled by an ioLTS I. $\Delta(I)$ represents its suspension ioLTS. Formal testing methods usually define the confidence degree between an implementation I and its specification $\mathcal{S}$ by means of a test relation. In the paper, we consider the $ioco$ relation. In [13], $ioco$ is defined for ioSTSs by:

Definition 10: Let I be an implementation modelled by an ioLTS, and $\mathcal{S}$ be an ioSTS. I is ioco-conforming to $\mathcal{S}$, denoted I $ioco$ $\mathcal{S}$ iff $Traces(\Delta(\mathcal{S})).(\sum^O \cup \{!\delta\}) \cap Traces(\Delta(I)) \subseteq Traces(\Delta(\mathcal{S}))$.

Intuitively, I is ioco-conforming to its specification $\mathcal{S}$ if, after each trace of the STS suspension $\Delta(\mathcal{S})$, I only produces outputs (and quiescence) allowed by $\Delta(\mathcal{S})$. Even though this relation clearly defines the ioco conformance, it would be more interesting to express it by means of proxy-tester properties. So, we will reformulate $ioco$ below.

Firstly, if we denote $NC\_Traces(\Delta(\mathcal{S})) = Traces(\Delta(\mathcal{S})).(\sum^O \cup \{!\delta\} \setminus Traces(\Delta(\mathcal{S}))$, the ioco relation can also be written by [13]:

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

The ioSTS reflection definition (Definition 5) exhibits the fact that incorrect specification behaviours are recognized in its Fail states. The non-conformant trace set $NC\_Traces(\mathcal{S})$ is therefore equivalent to $refl(Traces_{Fail}(REF(\mathcal{S})))$ (Proposition 6). Moreover, we have stated previously that $Traces_{Fail}(REF(\mathcal{S})) = Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ (Proposition 9). Consequently, we also can write:

Proposition 11:

$$I \text{ ioco } \mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap refl(Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))) = \emptyset$$

Now that proxy-tester traces are explicitly given in the test relation, we can also say that I is ioco-conforming to its specification when implementation traces do not belong to the set of partial proxy tester traces, obtained from the IUT side, leading to Fail. However, we can go farther, in the ioco rephrasing, by focusing on the parallel execution of the proxy-tester and the implementation. This execution can be modelled by a parallel composition:

Definition 12 (Passive test execution): $P =< Q_P, q0_P, \sum_P, \rightarrow_P >$ is the ioLTS semantics of a proxy-tester $\mathcal{P}(\mathcal{S})$. $I =< Q_I, q0_I, \sum_I \subseteq \sum_P, \rightarrow_I >$ is the implementation model.

The passive testing of I can be modelled by the parallel composition $||(P,I) =< Q_P \times Q_I, q0_P \times q0_I, \sum_P, \rightarrow_{||(P,I)} >$ where the transition relation $\rightarrow_{||(P,I)}$ is defined by the following rules. For readability reason, we denote an ioLTS transition $q_1 \xrightarrow[E]{?a} q_2$ if $Side(q_2) = E$ (the variable $side$ is valued to E in q_2).

(to IUT):

$$\frac{q'_2 \xrightarrow{?a}_{\Delta(I)} q'_3, q_1 \xrightarrow[Env]{?a}_P q_2 \xrightarrow[IUT]{!a}_P q_3}{q_1 q'_2 \xrightarrow{?a}_{||(P,I)} q_2 q'_2 \xrightarrow{!a}_{||(P,I)} q_3 q'_3}$$

(from IUT):

$$\frac{q'_1 \xrightarrow{!a}_{\Delta(I)} q'_2, q_1 \xrightarrow[IUT]{?a}_P q_2 \xrightarrow[Env]{!a}_P q_3, q'_3 \neq Fail}{q_1 q'_1 \xrightarrow{?a}_{||(P,I)} q_2 q'_2 \xrightarrow{!a}_{||(P,I)} q_3 q'_2}$$

(to Fail):

$$\frac{q'_1 \xrightarrow{!a}_{\Delta(I)} q'_2, q_1 \xrightarrow[IUT]{?a}_P Fail}{q_1 q'_1 \xrightarrow{?a}_{||(P,I)} Fail}$$

The immediate deduction of the $\rightarrow_{||(P,I)}$ definition is that $Traces(\Delta(I)) \cap refl(Traces^{IUT}_{Fail}(P)) = Traces(\Delta(I)) \cap refl(Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})))$ is equivalent to $refl(Traces_{Fail}(||(P,I)))$ (rule (to Fail)). In other terms, the non-conformant traces of $\Delta(I)$ can be also found in $Traces_{Fail}(||(P,I))$. Consequently, $ioco$ can be also reformulated as:

Proposition 13:

$$\begin{aligned} I \text{ ioco } \mathcal{S} &\Leftrightarrow Traces(\Delta(I)) \cap refl(Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))) \\ &= \emptyset \\ &\Leftrightarrow Traces_{Fail}(||(P,I)) = \emptyset \end{aligned}$$

In the previous proposition, the $refl$ function can be removed in the last equivalence since if we have $refl(Traces_{Fail}(||(P,I))) = \emptyset$, we also have $Traces_{Fail}(||(P,I)) = \emptyset$ (the function $refl$ only yields mirrored traces).

From the resulting relation, we can also deduce that non conformance (I $\neg ioco$ $\mathcal{S}$) is detected when a trace of the parallel composition $||(P,I)$ leads to one of its Fail states. The proxy-tester execution algorithm can now be easily deduced: it aims to construct traces in $||(P,I)$ and to detect bugs when traces lead to one of its Fail states.

Preliminarily, we assume that the client traffic is routed to the proxy-tester and that the proxy-tester instance configuration is identical to the implementation one. For example, if the implementation is a multi-instance system (one implementation instance per client), the proxy-tester mode must be identical.

As in Proposition 13, the implementation is assumed to behave as an ioLTS suspension. The proxy-tester handles a set of instantiated locations (pairs which consist of a location and a valuation), each expressing one of its current states. A single instantiated location is not sufficient since both the proxy-tester and the implementation may be indeterministic and may cover different behaviours. During its execution, the proxy-tester receives valued actions from both the client side and the implementation under test. While receiving actions, for each instantiated location, it covers its transitions until it reaches a Fail state. In this case, the implementation is not ioco-conforming to the specification. Algorithm 1 details the proxy-tester functioning.

The proxy-tester algorithm is based on a forward checking approach. It starts from its initial instantiated location i.e., $(l0_{\mathcal{P}(\mathcal{S})}, V0_{\mathcal{P}(\mathcal{S})})$. Upon a received event $e(p), \theta$ with eventually θ a valuation over p (line 2), which is either an valued action or quiescence, for each instantiated location in IL, it looks for the next transitions which can be fired (line 5): each one must have the same start location as the instantiated location (L, W), the same action as the received event $e(p)$ and its guard must be satisfied over the current internal variable valuation W and the external variable valuation θ. If this transition leads to a Fail state then the proxy-tester algorithm returns Fail and ends(lines 6-7). Otherwise, the event $e(p)$ is forwarded to the right side (IUT or Environment) with the next proxy-tester transition t_2 depending on the $side$ value found in the assignment A_2 of t_2, denoted $side(t_2)$ for simplicity in the algorithm (line 9). The algorithm uses the boolean $sent$ to forward the event only one time even though several paths can be covered in the proxy-tester on account of indeterminism. The new instantiated location is computed (lines 15-18) depending of $side(t_2)$). It is composed of the last reached location l_{next2}, and of a new valuation derived from θ, W, and from the assignments A and A_2. Since the both sides Env and IUT are separated in the proxy-tester by means of the renaming function ϕ, we use the latter to rename θ before using it: if the event is received from the environment, we rename θ with ϕ in A_2 because the second transition models an interaction with the implementation. If the event is received from the implementation, we rename θ with ϕ^{-1} in A_2 because the second transition models an interaction with the external environment. Once, each instantiated location of IL is covered, the proxy-tester waits for the next event.

This algorithm passively covers the proxy-tester transitions while receiving valued actions from both the external environment and the implementation under test until a Fail state is reached. When Fail is detected, the proxy-tester has constructed a run, from the initial instantiated location. From this run, we obtain a trace of $Traces_{Fail}(||(Env, P, I))$ leading to a Fail state. So, we can state the correctness of the algorithm with:

Proposition 14: The algorithm has returned Fail $\Rightarrow Traces_{Fail}(||(Env, P, I)) \neq \emptyset \Rightarrow \neg(I \; ioco \; \mathcal{S})$.

ALGORITHM 1: Proxy-tester algorithm

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

// initialise the proxy-tester to its initial Configuration
$IL := \{(l0_{\mathcal{P}(\mathcal{S})}, V0_{\mathcal{P}(\mathcal{S})})\}$;
while *Event*$(e(p), \theta)$ **do**
  $IL' = \emptyset$; $sent = false$;
  // check for each instantiated location
  foreach $(L, W) \in IL$ **do**
    // possible next fireable transition
    foreach $t = L \xrightarrow{e(p),G,A} l_{next} \in \rightarrow_{\mathcal{P}(\mathcal{S})}$ *such that* $\theta \cup W \models G$ **do**
      if $l_{next} = Fail$ **then**
        return FAIL; END;
      else
        // event forwarded to the right side
        if $side(t_2 = l_{next} \xrightarrow{!e(p_2),G_2,A_2}_{\mathcal{P}(\mathcal{S})} l_{next2})$=*IUT* $\wedge sent = false$ **then**
          Execute($t_2 = l_{next} \xrightarrow{(!e(p_2),\phi(\theta)),G_2,A_2}_{\mathcal{P}(\mathcal{S})} l_{next2})$;
          // send $(!e(p'), \phi(\theta))$ to IUT
          $sent = true$;
        if $side(t_2 = l_{next} \xrightarrow{!e(p_2),G_2,A_2}_{\mathcal{P}(\mathcal{S})} l_{next2})$=*ENV* $\wedge sent = false$ **then**
          Execute($t_2 = l_{next} \xrightarrow{(!e(p_2),\phi^{-1}(\theta)),G_2,A_2}_{\mathcal{P}(\mathcal{S})} l_{next2})$;
          // send $(!e(p), \phi^{-1}(\theta))$ to Env
          $sent = true$;
        // computation of the next instantiated location
        if $side(t_2 = l_{next} \xrightarrow{!e(p_2),G_2,A_2}_{\mathcal{P}(\mathcal{S})} l_{next2})$=*IUT* **then**
          $Q_{next} := (l_{next2}, A2(A(\theta \cup W) \cup \phi(\theta)))$;
        else
          $Q_{next} := (l_{next2}, A2(A(\theta \cup W) \cup \phi^{-1}(\theta)))$;
        $IL' := IL' \cup \{Q_{next}\}$;
  $IL := IL'$;

5 Experimentation and Discussion

We initially experimented our method by setting a proxy-tester between clients and the Amazon E-commerce Web service [17] whose specification part is illustrated in Figure 7. For simplicity, we only considered the string variable type and we adopted the Hampi solver [18] to check the satisfiability of guards and to perform updates over strings.

We have implemented a preliminary tool which computes an ioSTS proxy-tester from an ioSTS specification. Then, we have coded an HTTP proxy-tester from Algorithm 1. Its architecture is given in Figure 6. It receives HTTP requests and responses from clients or implementations and forwards them as it is described in the previous algorithm. Clients, calling the proxy-tester, were simulated with 20 instances of the SOAPUI tool [19], whose purpose is to execute action sequences on Web services. Each client, executed in loop, had its own sequence of actions which can be summarized by: look for an item (book, movie, etc.), create a cart, add the item to the cart, empty the cart. Clients were also configured to connect through a proxy. There is no need to configure the Amazon service here since it responds only to clients, it does not establish connection to clients.

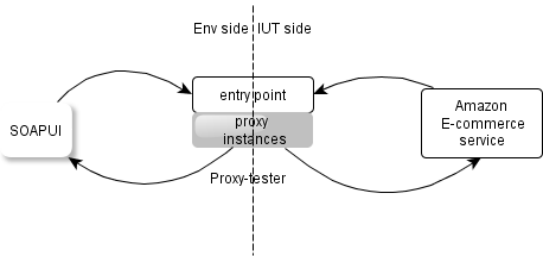

Fig. 6. The experimentation architecture

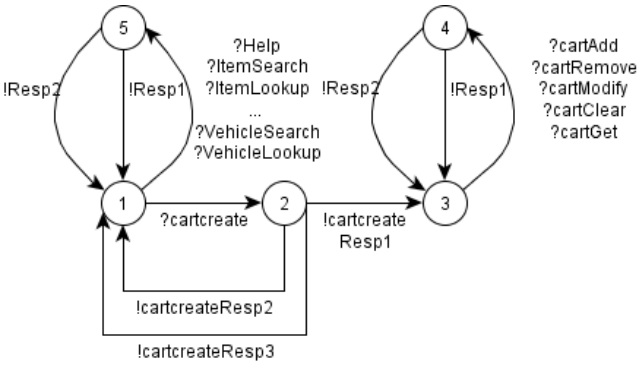

Fig. 7. A part of the Amazon E-commerce Web Service specification

The Amazon service may engage in several concurrent interactions by means of several instances, each one having its own state. As we had 20 clients calling 20 instances of the same service in the same time, the HTTP proxy-tester had to test passively these 20 instances in parallel. To do this, we have implemented a centralized entry point whose functioning is given in Algorithm 2. Each service instance is identified by an unique session identifier and is passively tested by an unique proxy-tester instance. Each pair (id, proxy-tester instance) is stored in a set denoted L. Whenever an HTTP message is received, the session id is extracted to check if an existing proxy-tester instance is running to test it. If such an instance already exists, the message is given to it, otherwise a new instance is created. If Fail is detected in a proxy-tester instance p, then the corresponding pair (id, p) is removed in L.

Since passive testing may reveal bugs sometimes af-

ALGORITHM 2: Proxy-tester entry point

```
L = ∅ is the set of pairs (id, p) composed of the session
identifier id and of the proxy-tester instance number p;
while HTTP message mess do
    id is the session id found in mess;
    if ∃(id, p) ∈ L then
    forward mess to p;
    else
        create a new proxy-tester instance p;
        L = L ∪ (id, p);
        forward mess to p;
    if ∃(id, p) ∈ L such that p has returned Fail then
    L = L \ {(id, p)};
```

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

Fig. 8. Statistics on the Amazon Web Service

ter a long period of time, we also prepared 15 specific client requests to try to detect incorrect behaviours of the Amazon service. These client requests were constructed from the experiment results of an earlier work, dealing with the Amazon service testing as well [16]. For instance, one of these requests is composed of the incorrect item type "Book" instead of "book". In this case, the service returns a wrong response composed of a message indicating that the customer id is false (which has nothing to do with the item name). For each of these requests, the proxy-tester has returned Fail, as expected.

We also performed this experimentation to check the feasibility of the method. Figure 8 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 ioSTS reflection whose transition number is finite as well. The ioSTS proxy-tester can be computed rapidly (at most some minutes).

This experimentation showed that proxy-testers offer definite advantages when used for testing Web services or Web applications, in comparison with sniffer-based tools, since privileges on the Amazon servers was not required and since it was not necessary to set a sniffer on each client as well.

6 Proxy-tester extension

As stated earlier, a proxy-tester represents an intermediary between client applications and the implementation that can be also seen like an upper layer which encompasses the implementation. This leads one to believe that proxy-testers could be extended to offer more features. Below, we propose a proxy-tester extension whose purpose is to improve the implementation functional quality by completing its proxy-tester. Concretely, this completion aims to manage faults and eventually to compensate them instead of the implementation. Then, we briefly present other possible extensions.

6.1 Fault management and compensation with proxy-testers

One of the passive testing benefits is to publish the implementation more rapidly while testing. In this case, the latter interacts with real client applications. Whenever an error is detected, it sounds more convenient to not stop the implementation abruptly, without warning the client side. Instead, we propose to complete the proxy-tester to produce a response when an error is detected and to eventually compensate (repair) this error.

6.1.1 Proxy-tester Fail location completion

With this straightforward step, two transitions are added to the proxy-tester after its Fail location: one to warn the client side that an error has occurred and another one to reset the proxy-tester and the implementation by supposing that any kind of reset function would be available for the latter.

For a proxy-tester $\mathcal{P}(\mathcal{S}) =< L_{\mathcal{P}}, l0_{\mathcal{P}}, V_{\mathcal{P}}, V0_{\mathcal{P}}, I_{\mathcal{P}}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}} >$, we define its Fail location completion by the STS operator Add_{Fail} with $Add_{Fail}\mathcal{P}(\mathcal{S}) =_{def} < L_{\mathcal{P}} \cup \{Reset\}, l0_{\mathcal{P}}, V_{\mathcal{P}}, V0_{\mathcal{P}}, I_{\mathcal{P}}, \Lambda_{\mathcal{P}} \cup \{!reset, !error\}, \rightarrow_{\mathcal{P}} \cup \{Fail \xrightarrow{!reset,,\{side:=IUT\}} Reset \xrightarrow{!error,,\{V_0 \cup V_0' \cup \{side:=Env\}\}} l0_{\mathcal{P}}\} >$,

The output action !error represents a generic message, sent to the client side. Pragmatically, its composition should depend on the implementation nature. For instance, if it is an object-oriented application, then the message could be an exception. In the same way, the reset action may represent a real reset message sent to the implementation, or a mechanism such as a waiting delay to invalidate an on-going session.

6.1.2 Fault compensation

We propose to add an error compensation mechanism to proxy-testers. Intuitively, when an error occurs in the implementation side, we propose to compensate it with the firing of new transitions that we call compensation transitions in the sequel. Once the error is compensated, the proxy-tester may return to one of its states e.g., its initial state.

To add compensation mechanisms in proxy-testers, we propose two solutions. The first one completes the proxy-tester with ioSTSs modelling compensation mechanisms. In the second approach, we suppose that there exist transitions, in the initial specification, which exhibit default behaviours. These are used in the proxy-tester construction to add compensation mechanisms.

- **Proxy-tester completion for error compensation:** for a proxy-tester $\mathcal{P}(\mathcal{S})$, we assume having a set of compensation mechanisms, denoted $Comp(\mathcal{P}(\mathcal{S}))$, gathering pairs $(t, \mathcal{C})$ where $t \in \rightarrow_{\mathcal{P}(\mathcal{S})}$ is a proxy-tester transition leading to Fail and $\mathcal{C} =< L_{\mathcal{C}}, l0_{\mathcal{C}}, V_{\mathcal{C}}, V0_{\mathcal{C}}, I_{\mathcal{C}}, \Lambda_{\mathcal{C}}, \rightarrow_{\mathcal{C}} >$ is an acyclic ioSTS such that $V_{\mathcal{P}} \subseteq V_{\mathcal{C}}$, $I_{\mathcal{P}} \subseteq I_{\mathcal{C}}$.
For a compensation mechanism $(t = l \xrightarrow{a(p),G,A} Fail, \mathcal{C}) \in Comp(\mathcal{P}(\mathcal{S}))$, $\mathcal{P}(\mathcal{S})$ is completed by means of the ioSTS operation *compensate* defined as $compensate(\mathcal{P}(\mathcal{S}), (t, \mathcal{C})) =_{def} \mathcal{P}_{comp} =< L_{\mathcal{P}} \cup L_{\mathcal{C}}, l0_{\mathcal{P}}, V_{\mathcal{P}} \cup V_{\mathcal{C}}, V0_{\mathcal{P}} \cup V0_{\mathcal{C}}, I_{\mathcal{P}} \cup I_{\mathcal{C}}, \Lambda_{\mathcal{P}} \cup \Lambda_{\mathcal{C}}, \rightarrow_{\mathcal{P}} >$ where $\rightarrow_{\mathcal{P}_{comp}}$ is given by the following rules:

$$\frac{l_1 \xrightarrow{a(p),G_a,A_a}_{\mathcal{P}} l_2, l_2 \neq Fail}{l_1 \xrightarrow{a(p),G_a,A_a}_{\mathcal{P}_{comp}} l_2} \qquad \frac{l_1 \xrightarrow{a(p),G_a,A_a}_{C} l_2, l_1 \neq l0_C}{l_1 \xrightarrow{a(p),G_a,A_a}_{\mathcal{P}_{comp}} l_2}$$

$$\frac{t=l \xrightarrow{a(p),G_a,A_a}_{\mathcal{P}} Fail, l0_C \xrightarrow{b(p),G_b,A_b}_{C} l_1}{l \xrightarrow{a(p),G_a,A_a}_{\mathcal{P}_{comp}} (Fail, a(p)G_aA_a) \xrightarrow{b(p),G_b,A_b}_{\mathcal{P}_{comp}} l_1}$$

- **Proxy-tester definition with error compensation mechanisms:** in this approach, we suppose that some transitions $l_1 \xrightarrow{!a(p),G,A} l_2$, modelling default output actions provided by the system, are marked in the specification. Intuitively, they are used in the proxy-tester construction to directly produce compensation transitions which will execute default actions when errors will occur. In other terms, the obtained proxy-tester will be composed of compensation transitions to send default valued actions to the external side and to eventually continue the proxy-tester execution as if there were no error.
To identify these transitions in the specification, we use the internal variable $marked$ and the assignment $marked := true$. For a specification $\mathcal{S}$, its proxy-tester $\mathcal{P}(\mathcal{S})$ can be constructed with the rules of Definition 7 by replacing the rule $toFail$ with these two following ones.
The first rule corresponds to the initial rule of Definition 7 used when there is no marked transition from the location l_1. The second rule adds a compensation mechanism to the proxy-tester by adding the transition to Fail of the ioSTS reflection (like in Definition 7), followed by a compensation transition. The latter is obtained from the marked transition of the specification which expresses the default output action which should be received from the implementation. According to Definition 7 (IUT to Env rule), the marked transition of the specification is also combined with its mirrored transition of the ioSTS reflection. This produces two transitions leading to $(l_2 l_2')$. The rule $(toFail_2)$ also constructs a compensation transition leading to $(l_2 l_2')$ which carries the same action as the marked transition. Its guard is different though: the action $!a(p)$ is not received from the implementation. Consequently, the proxy-tester needs to construct a valuation over the parameters p such that the guard of the default transition (G_a which allows the firing of a(p)) is satisfiable. The choice of the valuation is done by constraint solving over G_a and the variable set $V \cup p$, that is expressed by the $solve$ term.
Figure 9 illustrates an example of proxy-tester construction with compensation. We suppose that we have, in the specification of Figure 1, the transition $l_2 \xrightarrow{!lgResp(s),[s \neq "connected" \vee valid(k)=false],marked:=true} l_1$ marked as a default transition. In the ioSTS reflection (Figure 3) we have two transitions to Fail from l_2'. Therefore, the resulting proxy-tester has two compensation transitions. For each, the guard is $[s = solve(s \neq "connected" \vee valid(k) = false) \wedge (s \neq "connected" \vee valid(k) = false)]$ to compute a valuation satisfying the guard $s \neq "connected" \vee valid(k) = false$ and to return a valued action to the external side. The proxy-tester resumes its execution from the location 11', since the default transition leads to the location 1 in the specification.

(to: Fail) $l_{1'} \xrightarrow{b(p),G_b,A_b}_{REF} Fail, l_1 \in L_{\mathcal{S}}, l_1' = \phi(l_1)$, $l_1 \xrightarrow{!a(p),G_a,A_a \cup \{marked:=true\}} l_2 \notin \rightarrow_{\mathcal{S}}$ $\vdash$ $(l_1 l_{1'}) \xrightarrow{b(p),G_b,A_b(\{side:=IUT\})}_{\mathcal{P}} Fail$

(to: Fail2) $l_{1'} \xrightarrow{b(p),G_b,A_b}_{REF} Fail, l_1 \in L_{\mathcal{S}}, l_1' = \phi(l_1)$, $l_1 \xrightarrow{!a(p),G_a,A_a \cup \{marked:=true\}}_{\mathcal{S}} l_2$ $\vdash$ $(l_1 l_{1'}) \xrightarrow{b(p),G_b,A_b(\{side:=IUT\})}_{\mathcal{P}} (Fail, b(p) G_b A_b) \xrightarrow{!a(p),[p=solve(G_a) \wedge G_a],A_a(\{side:=Env\})}_{\mathcal{P}} (l_2 l_2')$

These completions for error compensation lead to the following remarks. Firstly, adding a compensation mechanism in proxy-testers ought to be considered with caution. We have not set any hypothesis on compensation transitions. However, it sounds more appropriate to add a compensation mechanism which resets the implementation, if possible. Indeed, once an error is detected, the current implementation state is unknown (at best, it may be estimated). Resetting the implementation ensures that its next state will be the initial one. An interesting perspective would be to give permission for proxy-testers to change the implementation state. This solution could be seen as dynamic partial repairing of implementations. But it requires strong hypotheses such as an intrusive access of the implementation and mechanisms to change the current state.

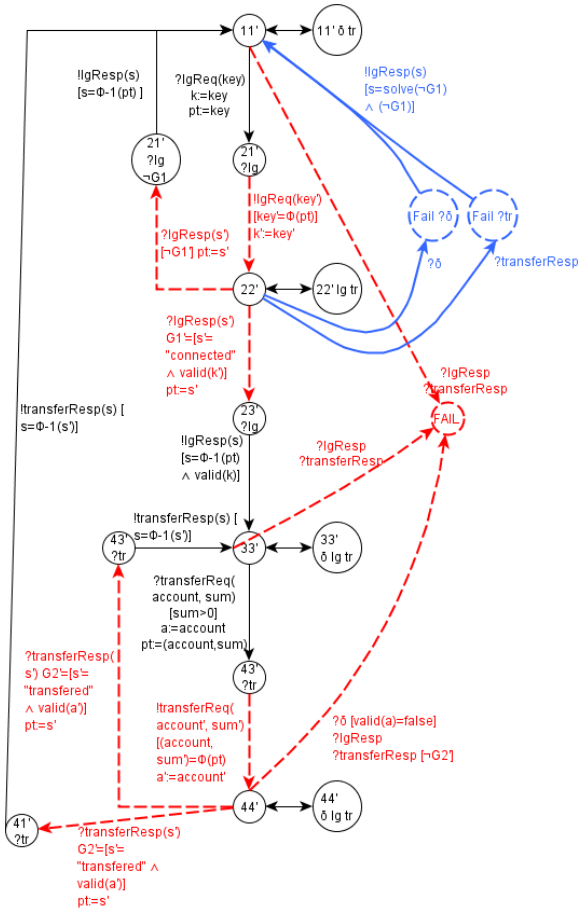

Fig. 9. A proxy-tester completed with fault management

The fault management completion transforms proxy-testers by adding new Fail locations and new transitions after these locations. The ioco relation is yet valid if it is written with the set of Fail locations, that we denote $FAIL$. Indeed, we have rephrased ioco by means of the set of traces $Traces_{Fail}(|||(Env, P, I))$ leading to Fail states. Proxy-testers still recognize the incorrect behaviours in their Fail states since the previous modification rules transform some transitions to Fail by transitions leading to locations in $FAIL$. So, the ioco relation can be rewritten with:

Proposition 15: Let $\mathcal{P}(\mathcal{S})$ be the proxy-tester of the ioSTS $\mathcal{S}$ and $FAIL \subseteq L_{\mathcal{P}(\mathcal{S})}$ be the set of its Fail locations. We have:

$$I \text{ ioco } \mathcal{S} \Leftrightarrow Traces_{FAIL}(|||(Env, P, I)) = \emptyset$$

6.2 Perspectives

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

- *Component-based system testing in environments whose access is restricted:* such systems could be passively tested by means of a set of proxy-testers interconnected to one another. If 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 synchronization of the proxy-testers and/or the recomposition of the partial traces collected from each component,
- *Combination with invariant-based methods:* it is still possible to extract implementation traces with proxy testers. So, passive testing methods which check the satisfiability 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 [6] can be executed to check the compliance 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 helps to protect a system from attacks received from the external environment. And on the other hand, proxy-testers could 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 ioSTSs: the first rules have to be synchronized with the transitions carrying the assertion $side := Env$ modelling interactions with the external environment, while security rules should be synchronized with the other transitions. The result corresponds to a specialized application firewall combined with a security passive testing tool,
- *System quality improvement:* as shown previously, proxy-testers can be also used to improve the functional quality of an implementation on the fly. Other criteria, than those presented above, could be studied. For instance, observability is one of the most important criteria in design and testing. It assesses the capability of inferring the internal state of an implementation by knowledge of its outputs. Proxy-testers could be used to improve the observability of an existing implementation by detecting a lack of observation and by sending an output action instead of the implementation.

7 Conclusion

Passive testing methods usually require sniffer-based tools, located in the same environment as the implementation, to extract traces. When the environment access is restricted for security or technical reasons, a sniffer-based tool cannot be set. We have proposed, in this paper, a proxy-oriented approach. We have formally defined a ioco proxy-tester, which represents an

intermediary between the external environment and the implementation under test. While observing the reactions of both the client and the implementation sides, it can detect if the implementation is not ioco-conforming to its specification. Besides, proxy-testers offer other possibilities: we have shown that they can be used to improve the implementation functional quality.

An immediate line of future work is to extend the notion of proxy-tester for both security testing and protection. Firstly, we intend to define security rules or test patterns with ioSTSs for protecting and for testing. Then, these rules must be synchronized with a proxy-tester to achieve an intermediary application which could filter out the client requests and check whether the responses meet the security requirements. A test relation, expressed with these rules, has to be also defined.

APPENDIX A
PROOF OF PROPOSITION 9

Let $\mathcal{S}$ be a specification, $\phi(REF(\mathcal{S}))$ its reflection, and $\mathcal{P}(\mathcal{S})$ its proxy-tester. We have $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces_{Fail}(REF(\mathcal{S}))$.

For an ioSTS $\mathcal{S}$, we denote $l_0 \xRightarrow{a_0 a_1 ... a_{n-1}} l_n$ if there exists a path $l_0 \xrightarrow{a_0(p_0),G_0,A_0} l_1 ... l_{n-1} \xrightarrow{a_{n-1}(p_{n-1}),G_{n-1},A_{n-1}} l_n$.

* $Traces_{Fail}(REF(\mathcal{S})) \subseteq Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$.

Let $path = l'_0 \xrightarrow{a_0(p'_0),G'_0,A'_0} l'_1 \ ... \ l'_{n-1} \xrightarrow{a_{n-1}(p_{n-1}),G'_{n-1},A'_{n-1}} Fail$ be a path of $REF(\mathcal{S})$ from its initial location to Fail.

(1)$\exists path_2 = l_0 l'_0 \xRightarrow{?a_0 ! a_0} l_1 l'_1 ... l_{n-1} l'_{n-1} \xRightarrow{?a_{n-1} ! a_{n-1}} Fail \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^{2n-1}$.

Proof of (1): We recall that ϕ is a function, $l'_1 = \phi(l_1)$.

According to the proxy-tester definition, we have:

- (A1) if $a_i (0 \leq i \leq n-2) \in \Lambda^I_{REF}$, we have $t'_i = l'_i \xrightarrow{?a_i,G'_i,A'_i} l_{i+1'} \in \rightarrow_{\phi(REF(\mathcal{S}))}$ and $t_i = l_i \xrightarrow{!a_i,G_i,A_i} l_{i+1} \in \rightarrow_{\mathcal{S}}$ (ioSTS definition). The rule (IUT to Env) produces a single path $(l_i l'_i) \xrightarrow{?a_i(p'_i),G'_i,A'_i(\{side:=(IUT)\})} (l_i l'_{i+1} a_i(p_i) G_i A_i) \xrightarrow{!a_i(p_i),[p_i=p'_i \wedge G_i],A_i(\{side:=(Env)\})} (l_{i+1} l i+1') \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^2$. If we have $t_j \neq t_i$ (with a different location, or a different guard, or a different assignment or a different action) the intermediary location will be different. If $t_j = t_i$ we still obtain a single path,
- (B1) if $a_i (0 \leq i \leq n-2) \in \Lambda^O_{REF}$, we have $t'_i = l'_i \xrightarrow{!a_i,G'_i,A'_i} l_{i+1'} \in \rightarrow_{\phi(REF(\mathcal{S}))}$ and $t_i = l_i \xrightarrow{?a_i,G_i,A_i} l_{i+1} \in \rightarrow_{\mathcal{S}}$ (ioSTS definition). The rule (Env to IUT) also produces a single path $(l_i l'_i) \xrightarrow{?a_i(p_i),G_i,A_i(\{side:=(Env)\})} (l_{i+1} l'_i a_i G_i A_i) \xrightarrow{!a_i(p'_i),[p'_i=p_i \wedge G'_i],A'_i(\{side:=(IUT)\})} (l_{i+1} l'_{i+1}) \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^2$,
- (C1) for $l'_{n-1} \xrightarrow{a_{n-1},G'_{n-1},A'_{n-1}} Fail$, we also have a single path $l_{n-1} l'_{n-1} \xrightarrow{a_{n-1},G'_{n-1},A'_{n-1}} Fail$ (rule IUT to Fail of the proxy tester construction).

From (A1)(B1)(C1), we deduce that there exists a path $l_0 l'_0 \xRightarrow{?a_0 ! a_0} l_1 l'_1 ... l_{n-1} l'_{n-1} \xRightarrow{?a_{n-1} ! a_{n-1}} Fail \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^{2n-1} = path_2$ □

Let $R = (l'_0, V'_0)(a_0(p'_0), \theta'_0)(l'_1, V'_1) ... (l'_{n-1}, V'_{n-1})(a_{n-1}(p'_{n-1}), \theta'_{n-1}) Fail \in RUN_{Fail}(\phi(REF(\mathcal{S})))$ be a run extracted from $path$, and $T = (a_0(p'_0), \theta'_0) ... (a_{n-1}(p'_{n-1}), \theta'_{n-1})$ the corresponding trace in $Traces_{Fail}(\phi(REF(\mathcal{S})))$.

(2) $\exists$ trace $T_2 \in Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ with $T = T_2$.

Proof of (2):

A run of an ioSTS is a path of the ioLTS semantics. According to (1), $path2$ is the path of the proxy-tester derived from $path$. Firstly, there exists a run $R_2 = q_0 \alpha_0 q_{01} \beta_0 q_1 ... q_i \alpha_i q_{ii+1} \beta_i q_{i+1} ... q_{n-1} \alpha_{n-1} Fail \in RUN_{Fail}(\mathcal{P}(\mathcal{S}))$ such that $\alpha_{n-1} = (a_{n-1}(p'_{n-1}), \theta'_{n-1})$, $\alpha_i = (a_i(p'_i), \theta'_i)$ if $Side(q_{ii+1}) = IUT$, or $\beta_i = (a_i(p'_i), \theta'_i)$ if $Side(q_{i+1}) = IUT$.

Consider $(l'_i, V'_i)(a_i(p'_i), \theta'_i)(l'_{i+1}, V'_{i+1})(0 \leq i \leq n-2)$ the partial run obtained from the transition $l'_i \xrightarrow{a_i(p'_i),G'_i,A'_i} l_{i+1'}$ of $path$. If we apply the ioLTS semantics transformation rule on $path_2$ (Definition 3) there exists a partial run composed of the same valuation θ'_i:

- (A) if $a_i \in \Lambda^I_{REF}$, and if we apply the ioSTS semantics transformation rule on $(l_i l'_i) \xrightarrow{?a_i(p'_i),G'_i,A'_i(\{side:=(IUT)\})} (l_i l'_{i+1} a_i G_i A_i) \xrightarrow{!a_i(p_i),[p_i=p'_i \wedge G_i],A_i(\{side:=(Env)\})} (l_{i+1} l i+1')$ (see A1), there exists $q_i \alpha_i q_{ii+1} \beta_i q_{i+1} = (l_i l'_i, V_i \cup V'_i \cup \{side := IUT||Env\})$ $(?a_i(p'_i), \theta'_i)$ $((l_i l'_{i+1}, V_i \cup V'_{i+1} \cup \{side := IUT\}))$ $(!a_i(\phi^{-1}(p'_i)), \phi^{-1}(\theta'_i))$ $(l_{i+1} l'_{i+1}, V_{i+1}) \cup V'_{i+1} \cup \{side := \{Env\}\})$,
- (B) if $a_i \in \Lambda^O_{REF}$ and if we apply the ioSTS semantics transformation rule on $(l_i l'_i) \xrightarrow{?a_i(p_i),G_i,A_i(\{side:=(Env)\})} (l_{i+1} l'_i a_i G_i A_i) \xrightarrow{!a_i(p'_i),[p'_i=p_i \wedge G'_i],A'_i(\{side:=(IUT)\})} (l_{i+1} l'_{i+1})$ (see B1), there exists $q_i \alpha_i q_{ii+1} \beta_i q_{i+1} = (l_i l'_i, V_i \cup V'_i \cup \{side := Env||IUT\})$ $(?a_i(\phi^{-1}(p'_i)), \phi^{-1}(\theta'_i))$ $(l_{i+1} l'_i, \phi^{-1}(V'_{i+1}) \cup V'_i \cup \{side := Env\})$ $(!a_i(p'_i), \theta'_i)$ $(l_{i+1} l'_{i+1}, V_{i+1} \cup V'_{i+1} \cup \{side := \{IUT\}\})$

(C) for $i = n-1$, there exists a partial run composed of θ'_{n-1} with $q_{n-1} \alpha_{n-1} Fail = (l_{n-1} l'_{n-1}, V'_{n-1} \cup V'_{n-1} \cup \{side := Env||IUT\})$ $a_{n-1}(p'_{n-1}), \theta'_{n-1}(Fail, V_n \cup \{side := IUT\})$ extracted from (c1).

From (A),(B) and (C), we deduce that there exists a run $R_2 = q_0 \alpha_0 q_{01} \beta_0 q_1 \ ... \ q_i \alpha_i q_{ii+1} \beta_i \ q_{i+1} \ ... \ q_{n-1} \alpha_{n-1} Fail \in RUN_{Fail}(\mathcal{P}(\mathcal{S}))$ such that $\alpha_{n-1} = (a_{n-1}(p'_{n-1}), \theta'_{n-1})$, $\alpha_i = (a_i(p'_i), \theta'_i)$ if $Side(q_{ii+1}) = IUT$, or $\beta_i = (a_i(p'_i), \theta'_i)$ if $Side(q_{i+1}) = IUT$.

The projection $proj_{Q_P \sum_P Side_{IUT}(Q_P)}(R_2)$ is, by definition, the partial trace $T_2 \in Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$ of R_2, and $T_2 = (a_0(p'_0), \theta'_0)(a_1(p'_1), \theta'_1)...(a_{n-1}(p'_{n-1}), \theta'_{n-1})$. We deduce that $T_2 = T$. □

From (2), we have for any trace $T \in Traces_{Fail}$ $(\phi(REF(\mathcal{S})))$, there exists a trace $T_2 \in Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$. We deduce that $Traces_{Fail}$ $(\phi(REF(\mathcal{S}))) \subseteq Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$.

* $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) \subseteq Traces_{Fail}(REF(\mathcal{S}))$.

We can follow the same reasoning.

Let $path_2 = l_0 l'_0 \xRightarrow{?a_0 ! a_0} l_1 l'_1 ... l_{n-1} l'_{n-1} \xRightarrow{?a_{n-1}} Fail \in (\rightarrow_{\mathcal{P}(\mathcal{S})})^{2n-1}$

(1) $\exists path = l'_0 \xrightarrow{a_0(p'_0), G'_0, A'_0} l'_1$ ... $l'_{n-1} \xrightarrow{a_{n-1}(p_{n-1}), G'_{n-1}, A'_{n-1}} Fail \in (\rightarrow_{REF(\mathcal{S})})^n$

Proof of (1): According to the proxy-tester definition, we have:

- (A1) if $l_i l'_i \xRightarrow{?a_i ! a_i} l_{i+1} l_{i+1'} = (l_i l'_i) \xrightarrow{?a_i(p'_i), G'_i, A'_i(\{side:=(IUT)\})} (l_i l'_{i+1} a_i(p_i) G_i A_i) \xrightarrow{!a_i(p_i), [p_i = p'_i \wedge G_i], A_i(\{side:=(Env)\})} (l_{i+1} l i + 1')$, these transitions can only be obtained from the rule (IUT to Env) with the transitions $l'_i \xrightarrow{?a_i, G'_i, A'_i} l_{i+1'} \in \rightarrow_{\phi(REF(\mathcal{S}))}$ and $t_i = l_i \xrightarrow{!a_i, G_i, A_i} l_{i+1} \in \rightarrow_{\mathcal{S}}$,
- (B1) if $l_i l'_i \xRightarrow{?a_i ! a_i} l_{i+1} l_{i+1'} = (l_i l'_i) \xrightarrow{?a_i(p_i), G_i, A_i(\{side:=(Env)\})} (l_{i+1} l'_i a_i G_i A_i) \xrightarrow{!a_i(p'_i), [p'_i = p_i \wedge G'_i], A'_i(\{side:=(IUT\})} (l_{i+1} l'_{i+1})$, these transitions can only be obtained from the rule (Env to IUT) with the transitions $l'_i \xrightarrow{!a_i, G'_i, A'_i} l_{i+1'} \in \rightarrow_{\phi(REF(\mathcal{S}))}$ and $l_i \xrightarrow{?a_i, G_i, A_i} l_{i+1} \in \rightarrow_{\mathcal{S}}$,
- (C1)for $l_{n-1} l'_{n-1} \xRightarrow{?a_{n-1}} Fail = (l_{n-1} l'_{n-1}) \xrightarrow{a_{n-1}, G'_{n-1}, A'_{n-1}} Fail$, it can only be obtained from the rule (IUT to Fail) from $l'_{n-1} \xrightarrow{a_{n-1}, G'_{n-1}, A'_{n-1}} Fail \in \rightarrow_{\phi(REF(\mathcal{S}))}$

According to (A1)(B1)(C1), there exists a path $l'_0 \xrightarrow{a_0(p'_0), G'_0, A'_0} l'_1 ... l'_{n-1} \xrightarrow{a_{n-1}(p'_{n-1}), G'_{n-1}, A'_{n-1}} Fail = path \in (\rightarrow_{REF(\mathcal{S})})^n$

□

Now, if we apply the same reasoning as (A)(B)(C) on the traces of $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S}))$, we also deduce that $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) \subseteq Traces_{Fail}(REF(\mathcal{S}))$.

As $\phi(REF(\mathcal{S}))$ corresponds to $REF(\mathcal{S})$ with renamed locations and variables, we can write $Traces^{IUT}_{Fail}(\mathcal{P}(\mathcal{S})) = Traces_{Fail}(REF(\mathcal{S}))$ since traces are composed of valued actions without location nor variable.

REFERENCES

[1] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software - Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[2] E. Bayse, A. Cavalli, M. Nunez, and F. Zaidi, "A passive testing approach based on invariants: application to the wap," in *Computer Networks*. Elsevier Science, 2005, vol. 48, pp. 247–266.

[3] A. Cavalli, C. Gervy, and S. Prokopenko, "New approaches for passive testing using an extended finite state machine specification," *Information and Software Technology*, vol. 45, no. 12, pp. 837 – 852, 2003, testing and Validation of Communication Software. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584903000636

[4] A. Cavalli, S. Maag, and E. M. de Oca, "A passive conformance testing approach for a manet routing protocol," in *Proceedings of the 2009 ACM symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 207–211. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529326

[5] G. Morales, S. Maag, A. R. Cavalli, W. Mallouli, E. M. de Oca, and B. Wehbi, "Timed extended invariants for the passive testing of web services." in *ICWS'10*, 2010, pp. 592–599.

[6] A. Cavalli, A. Benameur, W. Mallouli, and K. Li, "A Passive Testing Approach for Security Checking and its Practical Usage for Web Services Monitoring," in *NOTERE 2009*, Jun. 2009. [Online]. Available: http://liris.cnrs.fr/publis/?id=4207

[7] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John, "Passive testing and applications to network management," in *Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, ser. ICNP '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 113–. [Online]. Available: http://portal.acm.org/citation.cfm?id=850935.852443

[8] R. Miller and K. Arisha, "On fault location in networks by passive testing," in *IPCCC'2000, Phoenix, USA*, ser. ICNP '97, 2000, pp. 281–287.

[9] D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin, "Network protocol system monitoring: a formal approach with passive testing," *IEEE/ACM Trans. Netw.*, vol. 14, pp. 424–437, April 2006. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.872572

[10] B. Alcalde, A. R. Cavalli, D. Chen, D. Khuu, and D. Lee, "Network protocol system passive testing for fault management: A backward checking approach," in *FORTE*, ser. Lecture Notes in Computer Science, D. de Frutos-Escrig and M. Núñez, Eds., vol. 3235. Springer, 2004, pp. 150–166.

[11] S. Salva, "Passive testing with proxy tester," in *International Journal of Software Engineering and Its Applications (IJSEIA)*, vol. 5, no. 4. Science & Engineering Research Support soCiety (SERSC), 10 2011, pp. 1–16.

[12] L. Frantzen, J. Tretmans, and T. Willemse, "Test Generation Based on Symbolic Specifications," in *Formal Approaches to Software Testing – FATES 2004*, ser. Lecture Notes in Computer Science, J. Grabowski and B. Nielsen, Eds., no. 3395. Springer, 2005, pp. 1–15. [Online]. Available: http://www.cs.ru.nl/~lf/publications/FTW05.pdf

[13] V. Rusu, H. Marchand, and T. Jéron, "Automatic verification and conformance testing for validating safety properties of reactive systems," in *Formal Methods 2005 (FM05)*, ser. LNCS, J. Fitzgerald, A. Tarlecki, and I. Hayes, Eds. Springer, July 2005.

[14] L. Frantzen, J. Tretmans, and R. de Vries, "Towards model-based testing of web services," in *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, A. Bertolino and A. Polini, Eds., Palermo, Sicily, ITALY, June 9th 2006, pp. 67–82.

[15] ir. H.M. Bijl van der, D. A. Rensink, and D. G. Tretmans, "Component based testing with ioco," 2003. [Online]. Available: http://doc.utwente.nl/41390/

[16] S. Salva and I. Rabhi, "Stateful web service robustness," in *ICIW '10: Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 167–173.

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

[18] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2009, pp. 105–116.

[19] Eviware, "Soapui," 2011, http://www.soapui.org/.