A Model-Based Testing Approach Combining Passive Conformance Testing and Runtime Verification: Application to Web Service Compositions Deployed in Clouds

Sébastien Salva and Tien-Dung Cao

Abstract. This paper proposes a model-based testing approach which combines two monitoring methods, runtime verification and passive testing. Starting from ioSTS (input output Symbolic Transition System) models, this approach generates monitors to check whether an implementation is conforming to its specification and meets safety properties. This paper also tackles the trace extraction problem by reusing the notion of proxy to collect traces from environments whose access rights are restricted. Instead of using a classical proxy to collect traces, we propose to generate a formal model from the specification, called Proxy-monitor, which acts as a proxy and which can directly detect implementation errors. We apply and specialise this approach on Web service compositions deployed in PaaS environments.

Keywords: Passive Testing, Runtime Verification, Proxy, ioco, Web services, Clouds.

1 Introduction

Software testing is a large process, more and more considered by IT (Information technologies) companies, used to check the correctness or quality of software, that are notions required by end customers. In particular, Model-based Testing, which is the topic of this paper, is an approach where the system to test is formally described with specification models which express its functional behaviours. Beyond the use of formal techniques, these models offer the advantage to automate some (and eventually all) steps of the testing process. Usually, the latter is performed with active approaches: basically, test cases are constructed from the specification and are

Sébastien Salva
LIMOS CNRS UMR 6158, University of Auvergne, France
e-mail: sebastien.salva@udamail.fr

Tien-Dung Cao
School of Engineering, Tan Tao University, Vietnam
e-mail: dung.cao@ttu.edu.vn

R. Lee (Ed.): *SERA*, SCI 496, pp. 99–116.
DOI: 10.1007/978-3-319-00948-3_7

experimented on its implementation to check whether the implementation meets desirable behaviours w.r.t. a test relation which defines the confidence level of the test between the specification and implementations. Active testing may give rise to some inconvenient though, e.g., the repeated or abnormal disturbing the implementation.

Two other complementary approaches are employed to cover implementations over a longer period of time without disturbing them: passive testing and runtime verification. The former relies upon a monitor which passively observes the implementation reactions, without requiring pervasive testing environments. The sequences of observed events, called *traces*, are analysed to check whether they meet the specification. Runtime verification, originating from the Verification area, addresses the monitoring and analysis of system executions to check that strictly specified properties hold in every system states.

Both approaches share some important research directions, such as methodologies for checking test relations and properties, or trace extraction techniques. This paper explores these directions and describes a testing technique which combines the two previous approaches. The main contributions can be summarised threefold:

1. Combination of runtime verification and ioco passive testing: we propose to monitor an implementation against a set of safety properties which express that "nothing bad ever happens". These ones are known to be monitorable and can be used to express a very large set of properties, e.g., security vulnerabilities. We combine this monitoring approach with a previous work dealing with ioco passive testing [13]. Ioco [15] is a well-known conformance test relation which defines the conforming implementations by means of suspension traces (sequences of actions and quiescence). So, starting from an ioSTS (input output Symbolic Transition System) model, our method generates monitors to check whether an implementation is ioco-conforming to its specification and meets safety properties,
2. Trace extraction: to collect traces on a system in production, it is required to have the sufficient access rights on the implementation environment to install testing tools. More and more frequently over recent years, these environment access rights are restricted. For instance, Web server access rights are often strictly limited for security reasons. Another example concerns Clouds. Clouds, and typically PaaS (Platform as a service) layers are virtualised environments where Web services and applications are deployed. This virtualisation of resources combined with access restriction make difficult the trace extraction. We address this issue by using the notion of transparent proxy and by assuming that the implementation can be configured to pass through a proxy (usually the case for Web applications). But, instead of using a classical proxy to collect traces, we propose to generate a formal model from the specification, called Proxy-monitor, which acts as a proxy and which can directly detect implementation errors,
3. Analysis overhead: The proposed algorithms also offer the advantage of performing synchronous (receipt of an event, error detection, forward of the event to its recipient) or asynchronous analyses (receipt and forward of an event, error detection) whereas the use of a basic proxy allows asynchronous analysis only. The overhead, with both synchronous and asynchronous analyses, is measured and discussed in the experiment part.

The paper is structured as follows: initial notations and definitions are given in Section 2. Section 3 gives some definitions about runtime verification and ioco passive testing. The combination of both approaches is defined in Section 4. We apply, in Section 5, the concept of Proxy-monitor on Web service compositions deployed in Windows Azure which is the Cloud platform of Microsoft[1]. Finally, we review some related works in Section 6 and Section 7 concludes the paper.

2 Model Definition and Notations

In this paper, we focus on models called input/output Symbolic Transition Systems (ioSTS). An ioSTS is a kind of automata model, extended with two sets of variables, and with guards and assignments on transitions, which give the possibilities to model the system state and constraints on actions.

Below, we give the definition of an ioSTS extension, called ioSTS suspension which also expresses quiescence i.e., the authorised deadlocks observed from a location. 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 no output action may be observed. The guard of this new transition must return true for each value which does not allow firing a transition labelled by an output. More details about ioSTSs can be found in [9].

Definition 1 (ioSTS suspension). An ioSTS suspension is a tuple $< L, l0, V, V0, I, \Lambda, \rightarrow >$, where:

- L is the finite set of locations, with $l0$ the initial one,
- V is the finite set of internal variables, while I is the finite set of parameters. We denote D_v the domain in which a variable v takes values. The internal variables are initialised with the assignment V_0 on V, which is assumed to be unique,
- Λ is the finite set of symbols, partitioned by $\Lambda = \Lambda^I \cup \Lambda^O \cup \{!\delta\}$: Λ^I represents the set of input symbols, (Λ^O) the set of output symbols,
- $\rightarrow$ is the deterministic finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \in L$ to $l_j \in L$, denoted $l_i \xrightarrow{a(p),G,A} l_j$ is labelled by an action $a(p) \in \Lambda \times \mathscr{P}(I)$, with $a \in \Lambda$ and $p \subseteq I$ a finite set of interaction variables. G is a guard over $(p \cup V \cup T(p \cup V))$ which restricts the firing of the transition. $T(p \cup V)$ are boolean terms over $p \cup V$. Internal variables are updated with the assignment function A of the form $(x := A_x)_{x \in V}$ A_x is an expression over $V \cup p \cup T(p \cup V)$.

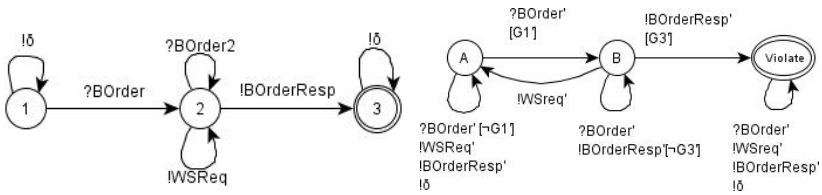
Web service compositions exhibit special properties relative to the service-oriented architecture (operations, partners, etc.). This is why we adapt the ioSTS action modelling. To represent the communication behaviours of Web service compositions with ioSTSs, we firstly assume that an action $a(p)$ expresses either the call of a Web service operation op with $a(p) = opReq(p)$, or the receipt of an operation response with $a(p) = opResp(p)$, or quiescence. The set of parameters p must gather also some specific variables:

[1] `http://www.windowsazure.com`

- the variable $from$ is equal to the calling partner and the variable to is equal to the called partner,
- Web services may engage in several concurrent interactions by means of several stateful instances called sessions, each one having its own state. For delivering incoming messages to the correct running session when several sessions are running concurrently, the usual technical solution is to add, in messages, a correlation value set which matches a part of the session state [1]. A correlation set is modelled with a parameter denoted *coor* in p.

The use of correlation sets also requires the following hypotheses which result from the correlation sets functioning. Particularly, the last one is required to correlate some successive operation calls with the same composite service instance:

Session identification: the specification is well-defined. When a message is received, it always correlates with at most one session.
Message correlation: except for the first operation call which starts a new composition instance, a message $opReq(p)$, expressing an operation call, must contain a correlation set $coor \subseteq p$ such that a non-empty subset $c \subseteq coor$ is composed of parameter values given in previous messages.



(a) An ioSTS suspension (b) A safety property

Fig. 1 ioSTS specifications

These notation are expressed in the straightforward example of Figure 1(a). For each symbol of Figure 1(a), Table 1 gives the corresponding action, guard and assignment. This specification describes the functioning of a *BookSeller* service. A client places an order composed of a list of books with *BookSeller* by supplying an ISBN list and the quantity of books ordered. *BookSeller* calls a service *Wholesaler* with *WholeSalerReq* to buy each book one by one. For one composition instance, we have two sessions of Web services connected together with correlations sets. Each session is identified with its own correlation set e.g., *BookSeller* with $c1 = \{account = "custid"\}$, and *Wholesaler* with $c2 = \{account = "custid", isbn = "2070541274"\}$. As these two correlation sets respect the Message correlation assumption, we can correlate the call of *Wholesaler* with one previous call of *BookSeller* even though several sessions are running in parallel.

An ioSTS is also associated to an ioLTS (Input/Output Labelled Transition System) to formulate its semantics. Intuitively, the ioLTS semantics is a valued automaton often infinite: the ioLTS states are labelled by internal variable values while

Table 1 Symbol table

Symbol	Action	Guard	Update
?BOrder	?BookOrderReq(List Books, quantity, account,from,to,corr)	$G1$=[from="Client" $\wedge$to="BR"$\wedge$ $corr = \{account\}$]	q:=quantity, b:=ListBooks, c1:=corr
?BOrder2	?BookOrderReq(List Books, quantity, account,from,to,corr)	$[\neg G1]$	
!BOrderResp	!BookOrderResp(resp, from, to, corr)	$G3$=[from="BR"$\wedge$ to= "Client"$\wedge$ resp="Order done"$\wedge$corr=c1]	
?R1	?BookOrderResp ?WholeSalerReq		
?R2	?BookOrderResp ?WholeSalerReq ?δ	[$\neq$G3] [$\neq$G2]	
!WSReq	!WholeSalerReq(isbn, from, to, corr)	$G2$=[isbn=b[q]$\wedge$ $q \geq 1\wedge$from= "BR"$\wedge$to= "WS"$\wedge$ $corr = \{a, isbn\}$]	$q := q-1$
?BOrderReq'	?BookOrderReq(List Books, quantity,account)	G1'=[quantity$\geq$ 1]	
!WSReq'	!WholeSalerReq(isbn)		
!BOrderResp'	!BookOrderResp(resp)	G3'=[end(resp)="done"]	
!BOrder [G1']	?BookOrderReq(List Books, quantity,)	[G1']	q:= quantity, b:=ListBooks, c1:= corr
?BOrderResp[$G3\wedge G3'$]	!BookOrderResp(resp)	$[G3 \wedge G3']$	
?BOrderResp[$\neg G3 \wedge G3'$]	!BookOrderResp(resp, from, to, corr)	$[\neg G3 \wedge G3']$	
?R3	?BookOrderResp ?WholeSalerReq ?δ	$[\neg G3 \wedge \neg G3']$ [$\neq$G2]	

transitions are labelled by actions and parameter values. The semantics of an ioSTS $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \rightarrow >$ is an ioLTS $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$ composed of valued states in $Q = L \times D_V$. $q_0 = (l0, V0)$ is the initial one, $\sum$ is the set of valued symbols and $\rightarrow$ is the transition relation. The complete definition of ioLTS semantics can be found in [9].

Runs and traces of ioSTS can be defined from their semantics:

Definition 2 (Runs and traces). For an ioSTS $\mathcal{S}$, 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. $Run_F(\mathcal{S}) = Run_F(||\mathcal{S}||)$ is the set of runs of $\mathcal{S}$ finished by a state in $F \times D_V \subseteq Q$ with F a set of locations of $\mathcal{S}$.

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

The parallel product is a classical state-machine operation used to produce a model representing the shared behaviours of two original automata. For ioSTSs, these ones are to be compatible:

Definition 3 (Compatible ioSTSs). An ioSTS $\mathcal{S}_1 = < L_1, l0_1, V_1, V0_1, I_1, \Lambda_1, \rightarrow_1 >$ is compatible with $\mathcal{S}_2 = < L_2, l0_2, V_2, V0_2, I_2, \Lambda_2, \rightarrow_2 >$ iff $V_1 \cap V_2 = \varnothing$, $\Lambda_1^I = \Lambda_2^I$, $\Lambda_1^O = \Lambda_2^O$ and $I_1 = I_2$.

Definition 4 (Parallel product $||$). The parallel product of two compatible ioSTSs $\mathcal{S}_1 =< L_1, l0_1, V_1,$
$V0_1, I_1, \Lambda_1, \rightarrow_1 >$ and $\mathcal{S}_2 = < L_2, l0_2, V_2, V0_2, I_2, \Lambda_2, \rightarrow_2 >$, denoted $\mathcal{S}_1||\mathcal{S}_2$, is the ioSTS $\mathcal{P} =< L_{\mathcal{P}}, l0_{\mathcal{P}}, V_{\mathcal{P}}, V0_{\mathcal{P}}, I_{\mathcal{P}}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}} >$ such that $V_{\mathcal{P}} = V_1 \cup V_2$, $V0_{\mathcal{P}} = V0_1 \wedge V0_2$, $I_{\mathcal{P}} = I_1 = I_2$, $L_{\mathcal{P}} = L_1 \times L_2$, $l0_{\mathcal{P}} = (l0_1, l0_2)$, $\Lambda_{\mathcal{P}} = \Lambda_1 = \Lambda_2$. The transition set $\rightarrow_{\mathcal{P}}$ is the smallest set satisfying the following inference rule:

$$\frac{l_1 \xrightarrow{a(p),G_1,A_1}_{\mathcal{S}_1} l_2, l_1' \xrightarrow{a(p),G_2,A_2}_{\mathcal{S}_2} l2'}{(l_1, l_1') \xrightarrow{a(p),G_1 \wedge G_2, A_1 \cup A_2}_{\mathcal{P}} (l_2, L_2')}$$

We end this Section with the definition of the ioSTS operation $refl$ which exchanges input and output actions of an ioSTS.

Definition 5 (Mirrored ioSTS and traces). Let $\mathcal{S}$ be an ioSTS. $refl(\mathcal{S}) =_{def} < L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{refl(\mathcal{S})}, \rightarrow_{\mathcal{S}} >$ where $\Lambda^I_{refl(\mathcal{S})} = \Lambda^O_{\mathcal{S}}$, $\Lambda^O_{refl(\mathcal{S})} = \Lambda^I_{\mathcal{S}}$.

We extend the $refl$ notation on trace sets. $refl : (\Sigma^*)^* \rightarrow (\Sigma^*)^*$ is 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).

3 Passive Testing with Proxy-Testers and Runtime Verification

To reason about conformance and property satisfiability, one assume that an implementation can be modelled with an ioLTS I. I is also assumed to have the same interface as the specification (actions with their parameters) and is input-enabled to accept any action.

For readability, the proofs of the propositions given below can be found in [14].

3.1 Verification of Safety Properties

The primary goal of runtime verification is to check whether an implementation I, from which traces can be observed, meets a set of properties expressed in trace predicate formalisms such as regular expressions, temporal logics or state machines. Given that we wish to merge the verification of safety properties with an ioSTS-based conformance, it sounds natural to also model them with a specific state machine model. We propose to take back the notion of observers [6] which capture the

negation of a safety property by means of final "bad" locations. Runs which lead to these locations represent behaviours which violate the property.

Definition 6 (Observer). An Observer is a deterministic ioSTS $\mathcal{O}$ composed of a non empty set of violation locations $Violate_{\mathcal{O}} \subset L_{\mathcal{O}}$. $\mathcal{O}$ must be both input and output-enabled, i.e. for each state $(l,v) \in L_{\mathcal{O}} \times D_{\mathcal{O}}$, and for each valued action $(a(p),\theta) \in \Lambda_{\mathcal{O}} \times D_p$, there exists $(l,v) \xrightarrow{a(p),\theta} (l',v') \in \rightarrow_{||\mathcal{O}||}$. Given an ioSTS $\mathcal{S}$, $Comp(\mathcal{S})$ stands for the set of compatible Observers of $\mathcal{S}$.

For the specification $\mathcal{S}$, an Observer $\mathcal{O}$ has to be input and output-enabled and compatible with $\mathcal{S}$. These assumptions are required to model a safety property which is violated by all the traces in $Traces_{Violate_{\mathcal{O}}}(\mathcal{O})$ and which is satisfied by all the traces in $(\sum_{||\mathcal{S}||})^* \setminus Traces_{Violate_{\mathcal{O}}}(\mathcal{O})$. Consequently, given an implementation I, one can say that I satisfies the Observer $\mathcal{O}$ if I does not yield any trace which also violates $\mathcal{O}$:

Definition 7 (Implementation satisfies Observer). Let $\mathcal{S}$ be an ioSTS and I an implementation. I satisfies the Observer $\mathcal{O} \in Comp(\Delta(\mathcal{S}))$, denoted $I \models \mathcal{O}$, if $Traces(\Delta(I)) \cap Traces_{Violate_{\mathcal{O}}}(\mathcal{O}) = \varnothing$.

Figures 1(b) and Table 1 illustrate an example of Observer for the specification of Figure 1(a). It means that "the receipt of an order confirmation ending with "done", without requesting *WholeSaler*, must never occurs".

Two Observers $\mathcal{O}_1$ and $\mathcal{O}_2$, describing two different safety properties, can be interpreted by the Observer $\mathcal{O}_1||\mathcal{O}_2$. In the remainder of the paper, we shall consider only one Observer, assuming that it may represent one or more safety properties.

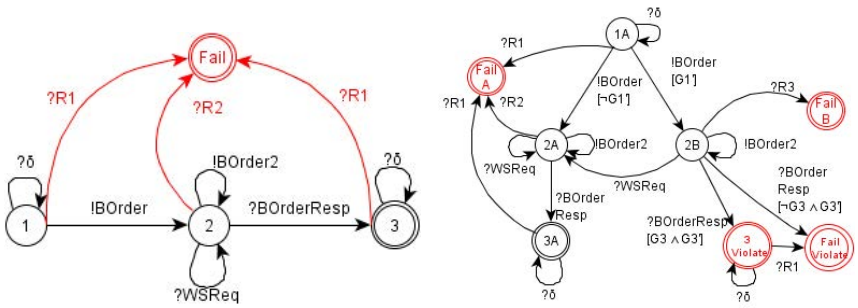
3.2 Ioco Testing with Proxy-Testers

In the paper, conformance is expressed with the relation *ioco* [15], which intuitively means that I is ioco-conforming to its specification $\mathcal{S}$ if, after each trace of the ioSTS suspension $\Delta(\mathcal{S})$, I only produces outputs (and quiescence) allowed by $\Delta(\mathcal{S})$. For ioSTSs, ioco is defined as:

Definition 8. 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 $Traces(\Delta(\mathcal{S})).(\sum^O \cup \{!\delta\}) \cap Traces(\Delta(I)) \subseteq Traces(\Delta(\mathcal{S}))$.

We have shown in our previous work [13] that *ioco* can be checked on implementations by means of a passive testing technique relying upon the concept of Proxy-tester. A Proxy-tester formally expresses the functioning of a transparent proxy, able to collect traces and to detect non-conformance without requiring to be set up in the same environment as the implementation one. We recall here some notions about Proxy-testers.

The Proxy-tester of a deterministic ioSTS $\mathcal{S}$ is derived from its Canonical tester $Can(\mathcal{S})$. This model is composed of the transitions of $\rightarrow_{\Delta(refl(\mathcal{S}))}$, i.e. the specification transitions labelled by mirrored actions (inputs become outputs and vice-versa).



(a) An ioSTS Canonical tester (b) An ioSTS Monitor

Fig. 2 Canonical tester and monitor examples

It is also enriched with transitions leading to a new location *Fail*, exhibiting the receipt of unspecified actions (expressing incorrect behaviours).

Instead of giving the definition of the Canonical tester, which can be found in [14], we illustrate in Figure 2(a) and Table 1 the Canonical tester of the ioSTS depicted in Figure 1(a). The specification actions are mirrored and, for instance, if we consider the location 2, new transitions to *Fail* are added to model the receipt of unspecified events (messages or quiescence).

The Proxy-tester of an ioSTS $\mathcal{S}$ corresponds to an augmented Canonical tester where all the transitions, except those leading to Fail, are *doubled* to express the receipt of an event and the forwarding to its addressee.

Definition 9 (Proxy-tester). The Proxy-tester of the ioSTS $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}} >$ is the ioSTS $Pr(Can(\mathcal{S}))$ where Pr is an ioSTS operation such that $Pr(Can(\mathcal{S})) =_{def} < L_{\mathcal{P}} \cup LF_{\mathcal{P}}, l0_{Can(\mathcal{S})}, V_{Can(\mathcal{S})} \cup \{side, pt\}, V0_{Can(\mathcal{S})} \cup \{side := \text{""}, pt := \text{""}\}, I_{Can(\mathcal{S})}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}} >$. $LF_{\mathcal{P}} = LF_{Can(\mathcal{S})} = \{Fail\}$ is the Fail location set. $L_{\mathcal{P}}$, $\Lambda_{\mathcal{P}}$ and $\rightarrow_{\mathcal{P}}$ are constructed with the following rules:

$$\frac{l_1 \xrightarrow{!a(p),G,A}_{Can(\mathcal{S})} l_2, l_2 \notin LF_{Can(\mathcal{S})}}{l_1 \xrightarrow{?a(p),G,A \cup \{p_t := p, side := \text{""}\}}_{\mathcal{P}} (l_1, l_2, a(p), G) \xrightarrow{!a(p),[p=p_t],\{(x:=x)_{x \in V_{Can(\mathcal{S})}}, side := \text{"Can"}, pt := pt\}}_{\mathcal{P}} l_2}$$

$$\frac{l_1 \xrightarrow{?a(p),G,A}_{Can(\mathcal{S})} l_2, l_2 \notin LF_{Can(\mathcal{S})}}{l_1 \xrightarrow{?a(p),G,A \cup \{p_t := p, side := \text{"Can"}\}}_{\mathcal{P}} (l_1, l_2, a(p), G) \xrightarrow{!a(p),[p=p_t],\{(x:=x)_{x \in V_{Can(\mathcal{S})}}, side := \text{""}, pt := pt\}}_{\mathcal{P}} l_2}$$

$$\frac{l_1 \xrightarrow{a(p),G,A}_{Can(\mathcal{S})} l_2, l_2 \in LF_{Can(\mathcal{S})}}{l_1 \xrightarrow{a(p),G,A \cup \{side := \text{"Can"}, pt := pt\}}_{\mathcal{P}} l_2}$$

Intuitively, the two first rules double the transitions whose terminal locations are not in the Fail location set LF to express the functioning of a transparent proxy. The first rule means that, for an event (action or quiescence) initially sent to the implementation, the Proxy-tester waits for this event and then forwards it. The two

transitions are separated by a unique location composed of the tuple $(l_1, l_2 a(p) G)$ to ensure that these two transitions, and only them, are successively fired. The last rule enriches the resulting ioSTS with transitions leading to Fail. A new internal variable, denoted *side*, is also added to keep track of the transitions provided by the Canonical tester (with the assignment side:="Can"). This distinction will be useful to define partial traces of Proxy-testers and to express conformance with them.

Previously, we have also intentionally enriched Proxy-tester transitions with an assignment on the variable *side*. The assignments $side =$ "*Can*" mark the transitions carrying actions provided by the Canonical-tester. These assignments help to extract partial runs and traces in Proxy-testers:

Definition 10 (Partial runs and traces). Let $\mathcal{P}$ be a Proxy-tester and $||\mathcal{P}|| = P =< Q_P, q0_P, \sum_P, \rightarrow_P >$ be its ioLTS semantics. We define $Side : Q_P \rightarrow D_{V_{\mathcal{P}}}$ the mapping which returns the valuation 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 $Run(\mathcal{P})$ be the set of runs of $\mathcal{P}$. We denote $Run^E(\mathcal{P})$ the set of partial runs derived from the projection $proj_{Q_P \sum_P Side^E(Q_P)}(Run(\mathcal{P}))$. It follows that $Traces^E(\mathcal{P})$ is the set of partial traces of (partial) runs in $Run^E(\mathcal{P})$.

With these notations, we have showed that *ioco* can be rephrased with Proxy-tester traces by [14]:

Proposition 1
I *ioco* $\mathcal{S} \Leftrightarrow Traces(\Delta(I)) \cap refl(Traces^{Can}_{Fail}(Pr(\ Can(\mathcal{S})))) = \varnothing$

4 Combining Runtime Verification and Proxy-Testing

4.1 Proxy-Tester and Observer Composition

Canonical testers are enough for detecting all the implementations that are not ioco-conforming to a given specification since they reflect incorrect behaviours in Fail states. Observers offer at least one similarity with Canonical testers since they describe undesired behaviours. This similarity tends to combine them to produce a model which could be used to detect both property violations and non-conformance. This product is called *Monitor*. It refines the original Canonical tester behaviours by separating the traces which violate safety properties among all the traces which may be observed from the implementation under test. A monitor is defined as:

Definition 11 (Monitor). Let $\Delta(\mathcal{S})$ be an ioSTS suspension and $\mathcal{O} \in Comp(\Delta(\mathcal{S}))$ be an Observer. The Monitor of the Canonical tester $Can(\mathcal{S})$ and of the Observer $\mathcal{O}$ is the ioSTS $\mathcal{M} = Can(\mathcal{S})||(refl(\mathcal{O})$.

As an example, the Monitor constructed from the previous Canonical tester (Figure 2(a)) and the Observer of Figure 1(b) is depicted in Figure 2(b) and Table 1. It contains different verdict locations: Fail received from the Canonical tester, Violate received from the Observer and a combination of both Fail/Violate which denotes non-conformance and the violation of the safety property. For example, the

trace "?*BookOrder*(,1,"*custid*") !*BookOrderResp*("*done*")" violates the Observer of Figure 1(b) because *WholeSaler* is not called. This trace reflects also an incorrect behaviour because the response received "done" is incorrect. We should have received "Order done".

The combination of Canonical tester locations with Observer ones leads to new locations labelled by local verdicts. We define these locations exhibiting verdicts by verdict location sets:

Definition 12 (Verdict location sets). Let $Can(\mathcal{S})$ be a Canonical tester and $\mathcal{O} \in Comp(\Delta(\mathcal{S}))$ be a compatible Observer with $\Delta(\mathcal{S})$. The parallel product $\mathcal{M} = Can(\mathcal{S})||refl(\mathcal{O})$ produces several sets of verdict locations defined as follows:

1. **VIOLATE** = $(L_{Can(\mathcal{S})} \setminus \{Fail\}) \times Violate_{\mathcal{O}}$,
2. **FAIL** = $\{Fail\} \times (L_{\mathcal{O}} \setminus Violate_{\mathcal{O}})$,
3. **FAIL/VIOLATE** = $\{(Fail, Violate_{\mathcal{O}})\}$.

In particular, we denote $LF_{\mathcal{M}} = FAIL \cup FAIL/VIOLATE$, the Fail location set of $\mathcal{M}$.

Monitors share many similarities with Canonical testers: they have a mirrored alphabet and a verdict location set LF. Typically, they are specialised Canonical testers recognising also property violations. To passively monitor an implementation, it sounds natural to apply the concept of Proxy-tester on Monitors. This gives a final model called Proxy-monitor:

Definition 13 (Proxy-monitor). Let $\mathcal{M}$ be a Monitor resulting from the parallel product $Can(\mathcal{S})||refl(\mathcal{O})$ with $\mathcal{S}$ an ioSTS and $\mathcal{O} \in Comp(\Delta(\mathcal{S}))$ an Observer compatible with the suspension of $\mathcal{S}$.

We call $Pr(\mathcal{M})$, the Proxy-monitor of $\mathcal{M}$.

Proxy-monitors are constructed as Proxy-testers except that the Fail location sets are different. For a Proxy-tester, there is only one Fail location, whereas a Proxy-monitor has a Fail location set $LF_{\mathcal{M}}$ equals to $FAIL \cup FAIL/VIOLATE$ since it stems from a composition between an Observer and a Canonical tester. Except this difference, transitions of the Monitor are still doubled in its Proxy-monitor.

Before focusing on test verdicts which can be obtained from Proxy-monitors, it remains to define formally the notion of passive monitoring of an implementation I by means of a Proxy-monitor. This product cannot be defined without modelling the external environment, e.g., the client side, which interacts with the implementation. We assume that this external environment can be also modelled with an ioLTS *Env* which can interact with I (hence $refl(Env)$ is compatible with I and $Traces(\Delta(Env))$ is composed of sequences in $refl((\sum_{\Delta(I)})^*)$).

Definition 14 (Monitoring of an implementation). Let $PM =< Q_{PM}, q0_{PM}, \sum_{PM}, \rightarrow_{PM} >$ be the ioLTS semantics of a Proxy-monitor $Pr(\mathcal{M})$ derived from an ioSTS $\mathcal{S}$ and an Observer $\mathcal{O} \in Comp(\Delta(\mathcal{S}))$. $QF_{PM} \subseteq Q_{PM} = LF_{Pr(\mathcal{M})} \times D_{V_{Pr(\mathcal{M})}}$ is its Fail state set. $I =< Q_I, q0_I, \sum_I \subseteq \sum_M, \rightarrow_I >$ is the implementation model, assumed compatible with $\mathcal{S}$ and $Env =< Q_{Env}, q0_{Env}, \sum_{Env} \subseteq \sum_P, \rightarrow_{Env} >$ is the ioSTS modelling the external environment, compatible with $refl(I)$.

The monitoring of I by $Pr(\mathcal{M})$ is expressed with the product $||_p(Env, PM, I) =< Q_{Env} \times Q_{PM} \times Q_I, q0_{Env} \times q0_{PM} \times q0_I, \sum_{PM}, \rightarrow_{||_p(Env,PM,I)} >$ where the transition relation $\rightarrow_{||_p(Env,PM,I)}$ is defined by the smallest set satisfying 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).

$$\frac{q_1 \xrightarrow{!a}_{\Delta(Env)} q_2, q_2'' \xrightarrow{?a}_{\Delta(I)} q_3'', q_1' \xrightarrow[",,"]{?a}_{PM} q_2' \xrightarrow["Can"]{!a}_{PM} q_3'}{q_1 q_1' q_2'' \xrightarrow[",,"]{?a}_{||_p(Env,PM,I)} q_2 q_2' q_2'' \xrightarrow["Can"]{!a}_{||_p(Env,PM,I)} q_2 q_3' q_3''}$$

$$\frac{q_2 \xrightarrow{?a}_{\Delta(Env)} q_3, q_1'' \xrightarrow{!a}_{\Delta(I)} q_2'', q_1' \xrightarrow["Can"]{?a}_{PM} q_2' \xrightarrow[",,"]{!a}_{PM} q_3'}{q_2 q_1' q_1'' \xrightarrow["Can"]{?a}_{||_p(Env,PM,I)} q_2 q_2' q_2'' \xrightarrow[",,"]{!a}_{||_p(Env,PM,I)} q_3 q_3' q_2''}$$

$$\frac{q_2 \xrightarrow{?\delta}_{\Delta(Env)} q_3, q_1'' \xrightarrow{!a}_{\Delta(I)} q_2'', q_1' \xrightarrow["Can"]{?a}_{PM} q_2', q_2' \in QF_{PM}}{q_2 q_1' q_1'' \xrightarrow["Can"]{?a}_{||_p(Env,PM,I)} q_2'}$$

The verdict list can now be drawn up from Definition 12. Concretely, the observed traces lead to a set of verdicts, extracted from the verdict location sets which indicate specification and/or safety property fulfilments or violations:

Proposition 2 (Test verdicts). *Consider an external environment Env, an implementation I monitored with a Proxy-monitor $Pr(\mathcal{M})$, itself derived from an ioSTS $\mathcal{S}$ and an Observer $\mathcal{O} \in Comp(\Delta(\mathcal{S}))$. Let $OT \subseteq Traces(||_p(Env, PM, I))$ be the observed trace set. If there exists $\sigma \in OT$ such that:*

1. *σ belongs to $Traces_{FAIL/VIOLATE}(||_p(Env, PM, I))$, then I does not satisfy the safety property and I is not ioco-conforming to $\mathcal{S}$,*
2. *σ belongs to $Traces_{FAIL}(||_p(Env, PM, I))$, then I is not ioco-conforming to $\mathcal{S}$,*
3. *σ belongs to $Traces_{VIOLATE}(||_p(Env, PM, I))$, then I does not satisfy the safety property.*

Intuitively, the sketch of proof of the above Proposition is based on some successive Trace set replacements. For example with 1), we have $Traces_{FAIL/VIOLATE}(||_p(Env, PM, I)) \neq \varnothing$. By considering successively Definition 11, Definition 13 and Definition 14, $Traces_{FAIL/VIOLATE}\ (||_p\ (Env, PM, I))$ can be replaced by $refl(Traces(\Delta(I))) \cap (Traces_{Fail}(Can(\mathcal{S})) \cap Traces_{Violate_{\mathcal{O}}}(refl(\mathcal{O}))) \neq \varnothing$. We deduce that $Traces^{Can}_{FAIL/VIOLATE}(||_p(Env, PM, I)) \neq \varnothing$ iff $refl(Traces(\Delta(I))) \cap Traces_{Fail}(Can(\mathcal{S})) \neq \varnothing(a)$ and iff $refl(Traces(\Delta(I))) \cap refl(Traces_{Violate_{\mathcal{O}}}(\mathcal{O})) \neq \varnothing(b)$. From (a), we have $\neg I$ ioco $\mathcal{S}$, from (b), we have $I \not\models \mathcal{O}$. The complete proof is given in [14].

5 Application to Web Service Composition Deployed in Clouds

We consider having a Web service composition deployed in a PaaS environment and we assume that each partner participating to the composition (Web services and clients) are configured to pass through a passive tester. The latter, whose architecture is depicted in Figure 3, is mainly based upon Proxy-monitors and aims to collect all the traces of Web service composition instances. To consider these instances and to detect non-conformance or violations of safety properties, several analyser instances, based upon a Proxy-monitor model, are executed in parallel. Any incoming message received from the same composition instance must be delivered to the same analyser instance: this step is performed by a module called *entry-point* which routes messages to the correct analyser instance by means of correlation sets.

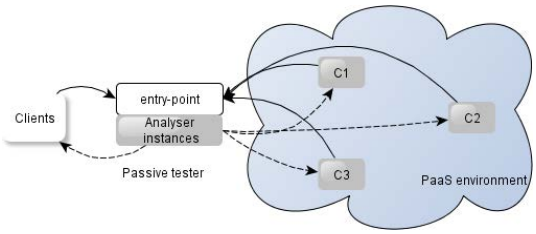


Fig. 3 The passive tester architecture

The entry-point functioning is given in Algorithm 1. The latter handles a set L of pairs (p_i, PV) with p_i an analyser instance identifier and PV the set of parameter values received in previous messages. For each received message, this set is used to correlate it with an existing composition instance in reference to the *Message correlation* hypothesis. Whenever a message $(e(p), \theta)$ is received, its correlation set c is extracted to check if an exiting analyser instance is running to accept it. This instance exists if L contains a pair (p_i, PV) such that a non-empty subset $c' \subseteq c$ is composed of values of PV (correlation hypothesis). In this case, the correlation set has been constructed from parameter values of messages received previously. If an instance is already running, the message is forwarded to it. Otherwise, (line 7), a new one is started. If an analyser instance p_i has returned a trace set (line 11), then the latter is stored in $Traces(Pr(\mathcal{M}))$ and the corresponding pair (p_i, PV) is removed from L.

Algorithm 2 describes the functioning of an analyser. Basically, it waits for an event (message or quiescence), covers Proxy-monitor transitions, and constructs traces to detect non conformance or property violations when a verdict location is reached. Algorithm 2 is based upon a forward checking approach: it starts from the initial state i.e., $(l0_{Pr(\mathcal{M})}, V0_{Pr(\mathcal{M})})$ and constructs a run denoted *Run*. Whenever an event $(e(p), \theta)$ is received (valued action or quiescence), with eventually θ a valuation over p (line 2), it looks for the next transitions which can be fired (line 5). Each transition must have the same start location as the one found in the final state (l, v) of the run *Run*, the same action as the received event $e(p)$ and its guard must be

Algorithm 1. Entry-point

input : Proxy-monitor $Pr(\mathcal{M})$
output: $Traces(Pr(\mathcal{M}))$

1 $L = \varnothing$;
2 **while** *message* $(e(p), \theta)$ **do**
3 extract the correlation set c in θ;
4 **if** $\exists (p_i, PV) \in L$ *such that* $c' \subseteq c$ *and* $c' \subseteq PV$ **then**
5 forward $(e(p), \theta)$ to p_i; $PV = PV \cup \theta$;
6 **else**
7 create a new $Pr(\mathcal{M})$ instance p_i;
8 $L = L \cup (p_i, \{\theta\})$; send $(e(p), \theta)$ to p_i;
9 **if** $\exists (p_i, PV) \in L$ *such that* p_i *has returned the trace set* T **then**
10 $Traces(Pr(\mathcal{M})) = Traces(Pr(\mathcal{M})) \cup T$;
11 $L = L \setminus \{(p_i, PV)\}$;

satisfied over the valuation $v \cup \theta$. If this transition reaches a verdict location (Definition 12) then the algorithm constructs a new *Run* (lines 8-11) and ends. Otherwise, the event $(e(p), \theta)$ is forwarded to the called partner with the next transition t_2 (lines 12 to 14). *Run* is completed with r' followed by the sent event and the reached state $q_{next2} = (l_{next2}, v'')$. Then, the algorithm waits for the next event. It ends when Fail and/or Violate is detected or when no new event is observed after a delay sufficient to detect several times quiescent states (set to ten times in the algorithm with qt). It returns the trace T derived from *Run*.

Algorithm 2 reflects exactly the definition of the monitoring of an implementation (Definition 14). It collects valued events and constructs traces of $||_p(Env, PM, I)$ by supposing that both I and Env are ioLTS suspensions. Lines (5-15) implement the rules of Definition 14. In particular, when a verdict location lv is reached (line 8 or 11), the analyser has constructed a run, from its initial state which belongs to $Run_V(||_p(Env, PM, I))$ with V a verdict location set. From this run, we obtain a trace of $Traces_V(||_p(Env, PM, I))$.

So, with Proposition 2, we can state the correctness of the algorithm with:

Proposition 3. *The algorithm has reached a location verdict in:*

- $FAIL/VIOLATE \Rightarrow Traces_{FAIL/VIOLATE}(||_p(Env, PM, I)) \neq \varnothing \Rightarrow I \not\models (\mathcal{O}, Violate_{\mathcal{O}})$ *and* $\neg(I \; ioco \; \mathcal{S})$,
- $FAIL \Rightarrow Traces_{FAIL}(||_p(Env, PM, I)) \neq \varnothing \Rightarrow \neg(I \; ioco \; \mathcal{S})$,
- $VIOLATE \Rightarrow Traces_{VIOLATE}(||_p(Env, PM, I)) \neq \varnothing \Rightarrow I \not\models (\mathcal{O}, Violate_{\mathcal{O}})$.

Both the previous algorithms perform a synchronous analysis. Algorithm 1 receives a message, transfers it to Algorithm 2, which constructs a run from Proxy-monitor transitions before eventually forwarding the message to its addressee. However, this analysis can be done asynchronously to reduce the checking overhead with slight modifications: as soon as Algorithm 1 receives a message, it can forward it directly. Then, the message can be also given to Algorithm 2 which constructs its run only.

Algorithm 2. Proxy-Monitor-based analyser algorithm

input : A Proxy-monitor $Pr(\mathcal{M})$
output: Trace

1 $Run := \{(q_0 = (l0_{Pr(\mathcal{M})}, V0_{Pr(\mathcal{M})}))\}$; $qt = 0$;
2 **while** *Event*$(e(p), \theta) \wedge$ *Fail is not detected* $\wedge qt < 10$ **do**
3 **if** $e(p) = !\delta$ **then**
4 $qt := qt + 1$;
5 **foreach** $t = l \xrightarrow{e(p),G,A} l_{next} \in \rightarrow_{Pr(\mathcal{M})}$ *such that Run ends with* (l, v) *and* $\theta \cup v \models G$ **do**
6 $q_{next} = (l_{next}, v' = A(v \cup \theta))$;
7 $r' = Run.(e(p), \theta).q_{next}$;
8 **if** $l_{next} \in VIOLATE \cup FAIL/VIOLATE$ **then**
9 Violation is detected; $Run := r'$;
10 **if** $l_{next} \in FAIL \cup FAIL/VIOLATE$ **then**
11 Fail is detected; $Run := r'$;
12 **if** $l_{next} \notin VIOLATE \cup FAIL/VIOLATE \cup FAIL$ **then**
13 Execute($t_2 = l_{next} \xrightarrow{!e(p),G_2,A_2}_{Pr(\mathcal{M})} l_{next2}$) ; `// forward` $(!e(p), \theta)$
14 $q_{next2} := (l_{next2}, A_2(\theta \cup v'))$; $Run := r'.(!e(p), \theta).q_{next2}$;
15 return the trace $T = \{proj_{\Sigma_{||Pr(\mathcal{M})||}}(Run)\}$;

5.1 Experimentation

We have implemented this approach in a tool called *CloudPaste* (Cloud PASsive TEsting [2]) to assess the feasibility of the approach. We experimented it with the Web service composition of Figure 1(a), developed with SOAP Web services in C# and deployed in Windows Azure. The Azure PaaS layer supports proxy configuration, i.e. services can be configured to pass through proxies that can be hosted inside or outside of the Cloud. The guard solving in Algorithm 2 is performed by the SMT (Satisfiability Modulo Theories) solver Z3 [3] that we have chosen since it offers good performance, takes several variable types and allows a direct use of arithmetic formulae. However, it does not support String variables. So, we extended the Z3 expression language with terms, which refer to the ioSTS definition, and in particular with String-based terms. A term stands for a function over internal variables and parameters which returns a Boolean. Basically, our tool takes Z3 expressions enriched with terms, terms are evaluated and replaced with Boolean values. Then, a Z3 script, composed of the internal variables, the parameters and the guard, is dynamically written before calling Z3. If the guard is satisfiable (not satisfiable), Z3 returns *sat* (*unsat* respectively). Z3 returns *unknown* when it cannot determine whether a formula is satisfiable or not.

[2] `http://sebastien.salva.free.fr/cloudpaste/cloudpaste.html`
[3] `http://z3.codeplex.com/`

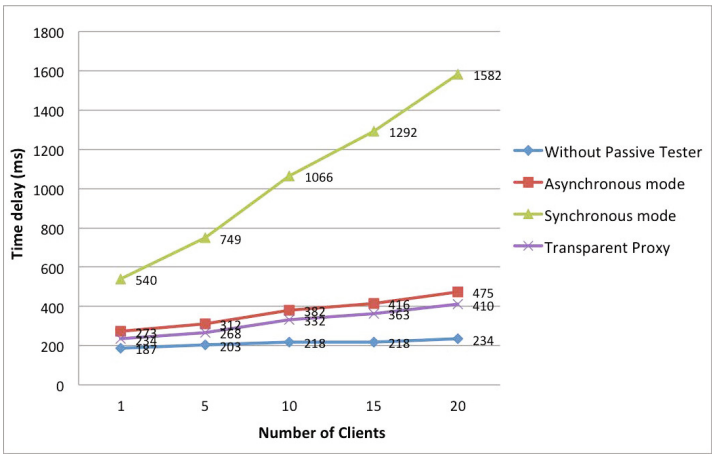


Fig. 4 Time processing measurements in Window Azure

We generated the Proxy-monitor from the ioSTS of Figure 1(a) combined with five safety properties with a tool generating Canonical testers and Proxy-monitors. The first property is the one described in Section 3.1. The other properties are based on security vulnerabilities. Client applications were simulated with at most 20 instances of Java applications performing one request to the *BookSeller* Web service with correct lists of two ISBNs. The passive tester was installed in a Windows server hosted in Azure. The detection of quiescence was implemented with a timeout set to 10s with respect to the HTTP timeout (usually set between 3 and 100 seconds).

Figure 4 depicts the average time processing of one client (milliseconds) when one up to twenty clients are running. The curves represent respectively the average time, without passive-tester, with the use of the transparent proxy *Charles* [4], with CloudPaste in asynchronous mode and in synchronous mode. In asynchronous mode, CloudPaste processes messages with a slightly higher time delay than Charles (with 20 clients, 102ms per message with Charles, 118ms per message with CloudPaste). This time delay is far lower than the quiescence timeout (and than the HTTP timeout as well). In synchronous mode, the checking overhead is higher with an average time of 135ms per message for 1 client and 395ms per message for 20 clients. This big difference results from the constraint solver calls and from the lack of optimisation of our code (Z3 is not yet called in parallel in CloudPaste). Nevertheless, in synchronous mode, the time processing is still lower than the timeout set to observe quiescence (the testing process can be done) and than the HTTP timeout (messages can be forwarded correctly). This mode is also particularly interesting since it offers the advantage to eventually implement recovery action calls, e.g., error compensation or implementation reset, when an error is detected. Error recovery is not possible with classical proxies or in asynchronous mode. These results tend

[4] `http://www.charlesproxy.com`

to show that our approach represents a good solution for testing and that it can be done in real-time.

6 Related Works

The works proposed in the literature either dealing with runtime verification, e.g., [3, 4, 8] or with passive testing, e.g., [11, 5, 2, 12] rely on three main methods for trace observation. Monitors or passive testers can be encapsulated within the implementation environment [4, 5], i.e. it is modified or completed with new test modules e.g., workflow engines. Traces can be also observed with probes, e.g., sniffer-based tools, deployed in the implementation environment [3, 11, 8, 12]. With these two methods, it is required to assume that the implementation environment access rights are granted and that it may be modified. This prerequisite condition cannot be always satisfied with any implementation environment. Installing a sniffer-based tool in a PaaS platform is not possible since services are geographically deployed in a dynamic manner and since the access and the modification of PaaS and IaaS layers are not authorised. The same issue is usually raised with Web servers: Web applications are tested by means of active methods with a testing server and are then deployed into another production server whose access rights are restricted for security reasons. Another possibility consists in adding directly probes into the system code [7] but this is occasionally considered only since it has the disadvantage of modifying the implementation behaviours for testing. Our work focuses on these issues, by proposing the use of the proxy concept for testing. A first naive solution would be to collect traces with a proxy e.g. SOAPUI[5], to eventually prune/modify them to obtain usual traces (those that would be collected directly from the implementation) and to analyse them with a specification to detect errors. Our proposal consists in generating automatically another model called *Proxy-tester* from a specification and to use a passive tester performing an analysis directly with Proxy-testers.

Few works have also focused on the combination of runtime verification with conformance testing [3, 6]. The latter consider active testing and therefore a combination of properties with classical test cases which are later actively executed on the system: in [3], test cases are derived from a model describing system inputs and properties on these inputs. Once test cases are executed, the resulting traces are analysed to ensure that the properties hold. Runtime verification and active testing have been also combined to check whether a system meets a desirable behaviour and conformance w.r.t. ioco [6]. In these previous works, the combination of active testing with runtime verification helps to choose, in the set of all possible test cases, only those expressing behaviours satisfying the given specification and safety properties. The other behaviours (those satisfying the specification but not the safety property and vice-versa) are not considered. Our proposal solves this issue by defining differently specifications and safety properties so that the resulting monitors could cover any behaviours passively over a long period of time.

[5] `www.soapui.org/`

7 Conclusion

We have proposed a testing approach combining ioco passive testing with runtime verification of safety properties. A monitor, called Proxy-monitor, is automatically generated from safety properties and specifications modelled with ioSTSs. Proxy-monitors are then used to detect whether the implementation is not ioco-conforming to its specification or if the former violates properties. Proxy-monitors are also based upon the notion of transparent proxy to ease the extraction of traces from environments in which testing tools cannot be deployed. Our approach can be applied on different types of communication software, e.g., Web service compositions, in condition that they could be configured to send messages through a proxy. In the experimentation part, we have also showed that the overhead obtained by the use of our approach remains reasonable and is much lower than the HTTP timeout.

In this paper, we have dealt with deterministic ioSTS specifications to rephrase *ioco*, like many testing approaches proposed in the literature. However, nondeterministic ioSTSs can be considered as well by apply determinization techniques [10]. In a future work, we could also consider nondeterministic ioSTSs with a weaker test relation than ioco to generate nondeterministic Proxy-testers. Another immediate line of future work concerns the enrichment of the experimentation with larger Web service compositions deployed in different Clouds, each having its own possibilities and restrictions.

References

1. Ws-bpel, Oasis Consortium (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
2. Andrés, C., Cambronero, M.E., Núñez, M.: Passive testing of web services. In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 56–70. Springer, Heidelberg (2011)
3. Arthoa, C., Barringerb, H., Goldbergc, A., Havelundc, K., Khurshidd, S., Lowrye, M., Pasareanuf, C., Rosug, G., Seng, K., Visserh, W., Washingtonh, R.: Combining test case generation and runtime verification. Theoretical Computer Science 336(2-3), 209–234 (2005)
4. Barringer, H., Gabbay, D., Rydeheard, D.: From runtime verification to evolvable systems. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 97–110. Springer, Heidelberg (2007)
5. Cavalli, A., Benameur, A., Mallouli, W., Li, K.: A Passive Testing Approach for Security Checking and its Practical Usage for Web Services Monitoring. In: NOTERE 2009 (2009)
6. Constant, C., Jéron, T., Marchand, H., Rusu, V.: Integrating formal verification and conformance testing for reactive systems. IEEE Trans. Softw. Eng. 33(8), 558–574 (2007), doi:10.1109/TSE.2007.70707
7. d'Amorim, M., Havelund, K.: Event-based runtime verification of java programs. In: Proceedings of the Third International Workshop on Dynamic Analysis, WODA 2005, pp. 1–7. ACM, New York (2005), doi:10.1145/1082983.1083249
8. Falcone, Y., Jaber, M., Nguyen, T.-H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 204–220. Springer, Heidelberg (2011)

9. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test Generation Based on Symbolic Specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)
10. Jéron, T., Marchand, H., Rusu, V.: Symbolic determinisation of extended automata. In: Navarro, G., Bertossi, L., Kohayakawa, Y. (eds.) TCS 2006. IFIP, vol. 209, pp. 197–212. Springer, Boston (2006)
11. Lee, D., Chen, D., Hao, R., Miller, R.E., Wu, J., Yin, X.: Network protocol system monitoring: a formal approach with passive testing. IEEE/ACM Trans. Netw. 14, 424–437 (2006)
12. Nguyen, H.N., Poizat, P., Zaidi, F.: Online verification of value-passing choreographies through property-oriented passive testing. In: Ninth IEEE International Symposium on High-Assurance Systems Engineering, pp. 106–113 (2012)
13. Salva, S.: Passive testing with proxy-testers. International Journal of Software Engineering and Its Applications (IJSEIA). Science & Engineering Research Support Society (SERSC) 5 (2011)
14. Salva, S.: A model-based testing approach combining passive testing and runtime verification. Tech. rep., LIMOS, LIMOS Research report RR13-04 (2013), `http://sebastien.salva.free.fr/useruploads/files/RR-13-04.pdf`
15. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)