An approach for testing passively Web service compositions in Clouds

Sébastien Salva
LIMOS CNRS UMR 6158, PRES Clermont University,
Aubièrè, FRANCE

Abstract—This paper proposes a formal passive testing approach for Web service compositions deployed in Clouds. It addresses an issue lifted by the PaaS layer of Clouds, which represents virtualised environments where compositions can be deployed. In these latter, sniffer-based modules, which are the heart of passive testing methods, cannot be installed for technical reasons. We propose a new approach based on the notion of transparent proxy. We define a new model, called proxy-tester, which is used to observe the functional behaviours of the composition under test but also to check if it is ioco-conforming to its specification. We also provide a solution and algorithms for testing several composition instances in parallel.

Keywords: Web service composition; ioSTS; passive testing; proxy tester.

1. Introduction

Conformance testing is now a well-established activity, in model-based development, which aims at trying to find defects in systems or software by analysing the observed functional behaviours of an implementation and by comparing them with those of its specification. Two kind of approaches may be considered for testing. On the one hand, active methods can be applied to experiment an implementation with a set of predefined test cases, constructed from the specification, and to conclude whether a test relation is satisfied. For instance, ioco [1] is a standard conformance test relation which expresses the set of correct implementations by means of suspension traces (sequences of actions and quiescence).

On the other hand, passive testing, which is the topic of the paper, is another approach which does not actively experiment the implementation under test, but which passively observes its reactions over a long period of time without the need of a pervasive test environment. This approach offers definite advantages in comparison to active methods e.g., to publish more rapidly a system or to not inadvertently disturb it while testing. The passive tester is composed of a kind of sniffer-based module which is supposed to observe both the stimuli sent to the implementation and its reactions in the environment where it is running. Then, the resulting traces are used to check the satisfiability of a test relation or of properties called invariants [2]. In literature, all the proposed works, dealing with passive testing, rely on a sniffer-based module to extract traces [2], [3], [4]. This module must be installed in the implementation environment and may require modifications of the latter. However, at the same time, some current trends in Computer Science, such as Cloud computing, propose to replace physical environments, such as servers, by virtualised environments composed of resources whose locations and details are not known. In these environments, a sniffer-based analyser cannot be installed, consequently the current passive methods cannot be applied.

This paper addresses this issue by focusing on the passive testing of Web service compositions deployed on Clouds, and more precisely on PaaS (Platform as a Service) which is the layer that supports service deployment [5]. In the context of PaaS environments, we propose a model-based passive testing method which relies on the notion of transparent proxy to observe traces. With this concept, no code modification is required, it is only necessary to configure Web services and clients to pass through a proxy. For Web service compositions described with ioSTSs (input/output Symbolic Transition Systems [6]), we firstly define another model called proxy-tester. Intuitively, this model describes the functioning of a dedicated kind of proxy of one specification: it represents an intermediary between the different partners taking part to the composition (Web services and clients). So, once executed with an algorithm provided in the paper, it helps to follow the functioning of one Web service composition instance and to forward any received message to the right partner. Thanks to this model, we also show that composition traces can be extracted while testing and that these traces can be also used to check whether the composition under test is ioco-conforming to its specification.

This paper focuses on another issue briefly mentioned previously. Web service compositions deployed in PaaS can be invoked by several clients at the same time. Consequently, several composition instances can be executed concurrently. We consider these composition instances in the paper and provide original algorithms to construct traces in parallel.

This paper is structured as follows: Section 2 defines the Web service composition modelling. Section 3 describes our passive testing method by defining the ioco proxy-tester of one specification. In Section 4, we details the passive tester functioning, which is suitable to test several instances of
the same composition in parallel. Finally, we conclude in Section 5.

2. Model Definition and notations

To express formally the functional behaviours of Web service compositions, we focus on models called input/output Symbolic Transition Systems (ioSTS). An ioSTS is a kind of automata model which is extended with a set of variables and with transition guards and assignments, giving the possibilities to model the system state and constraints on actions. With ioSTSs, action set is separated 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.

Below, we give the definition of an ioSTS extension, called ioSTS suspension which also expresses quiescence i.e., the absence of observation from a location. Quiescence is modelled by a new symbol \( \delta \) and an augmented ioSTS denoted \( \delta(\text{ioSTS}) \). For an ioSTS \( S \), \( \delta(S) \) is obtained by adding a self-loop labelled by \( \delta \) for each location where quiescence may be observed. The guard of this new transition must return true for each value which does not allow firing a transition labelled by an output.

**Definition 1 (ioSTS suspension)** An ioSTS suspension is a tuple \( < L, l_0, V, V_0, I, \Lambda, \rightarrow, > \), where:

- \( L \) is the finite set of locations, with \( l_0 \) the initial one,
- \( V \) is the finite set of internal variables, while \( I \) is the finite set of interaction ones. We denote \( D_v \) the domain in which a variable \( v \) takes values. The internal variables are initialised with the assignment \( V_0 \) on \( V \), assumed to be unique,
- \( \Lambda \) is the finite set of symbols, partitioned by \( \Lambda = \Lambda^I \cup \Lambda^O \cup \{\delta\} \); \( \Lambda^I \) represents the set of input symbols, \( \Lambda^O \) the set of output symbols,
- \( \rightarrow \) is the (potentially non 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 \( a(p) \in \Lambda \times \mathcal{P}(I) \), with \( a \in \Lambda \) and \( p \subseteq I \) a finite set of interaction variables \( p = \{p_1, ..., p_k\} \). \( G \) 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 \( (x \leftarrow A_x)_{x \in V} \), \( A_x \) 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 (restrict) the ioSTS action modelling:

**Messages model:** to represent the communication behaviours of Web service compositions with ioSTSs, we firstly assume that an action \( a(p) \) expresses a message i.e., the call of a Web service operation \( \text{op} (a(p) = \text{opResp}(p)) \), or the receipt of an operation response \( a(p) = \text{opResp}(p) \), or quiescence. The set of parameters \( p \) must gather also some specific variables:

- 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, correlation values which match a part of the session state [7], [8]. So when a session calls another partner, the message must be composed of a set of values called correlation set which identifies the session. We model a correlation set in a message \( a(p) \) with a parameter, denoted \( \text{coor} \in p \).

The use of correlation sets with ioSTSs also implies to set the following hypotheses on actions:

- **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 \( \text{opResp}(p) \), expressing an operation call, must contain a correlation set \( \text{coor} \subseteq p \) such that a subset \( c \subseteq \text{coor} \) of the correlation set is composed of parameter values given in previous messages.

The first hypothesis results from the correlation sets functioning. The last one is given to coordinate the successive operation calls together so that we could follow the functioning of one composition instance without ambiguity by observing the messages and the correlation sets exchanged between sessions, while testing.

These notation are expressed in the straightforward example of Figures 1 and 2. This specification describes a composition of two Web services: Shoppingservice represents an interface which allows a customer to look for the availability of books with isbn (International Standard Book Number). This service calls StockService to check the availability and the current stock of one book.

For one composition instance, we have two sessions of services connected together with correlations sets. Each service session is identified with its own correlation set e.g., Shoppingservice with \( c_1 = \{\text{account} = \"custid\"\} \), StockService with \( c_2 = \{\text{account} = \"custid\", \text{isbn} = \"2070541274\"\} \). As these two correlation sets respect the Message correlation assumption, we can correlate the call of StockService with one previous call of Shoppingservice 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 corresponds to a valued automaton without symbolic variable, which is often infinite:
the ioLTS states are labelled by internal variable values while
transitions are labelled by actions and interaction variable
(parameter) values. The semantics of an ioSTS $S = < L, I, \Lambda, \rightarrow >$ is an ioLTS $|S| = < Q, q_0, \sum, \rightarrow >$ composed valued states in $Q = L \times D_V$, $q_0 = (l_0, V_0)$ 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 [6].

Intuitively, for an ioSTS transition $l_1 \xrightarrow{a(p), G, A} l_2$, we obtain an ioLTS transition $(l_1, v) \xrightarrow{a(p), G} (l_2, v')$ with $v$ a set of values over the internal variable set, if there exists a parameter value set $\theta$ such that the guard $G$ evaluates to true with $v \cup \theta$. Once the transition is executed, the internal variables are assigned with $v'$ derived from the assignment $A(v \cup \theta)$. Finally, runs and traces of ioLTS can be defined from their semantics:

**Definition 2 (Runs and traces) For an ioSTS $S$, interpreted by its ioLTS semantics $|S| = < Q, q_0, \sum, \rightarrow >$, a run $q_0, q_1, \ldots, q_n, q_{n+1}$ is an alternate sequence of states and valued actions. \( \text{RUN}_F(S) = \text{RUN}_F(|S|) \) is the set of runs of $S$ finished by a state in $F \times D_V \subseteq Q$ with $F$ a location of $S$.

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

3. Ioco passive testing with proxy-testers

This Section covers the theoretical aspects of ioco-proxy testing. Instead of using a classical proxy for observing messages, we formalise below the notion of proxy-tester of an ioSTS specification. This model will help to observe traces but will be also used to directly check whether a composition implementation is ioco-conforming to its specification.

3.1 Proxy-tester definition

A proxy-tester corresponds to a passive intermediary between the partners of one composition (Web services and clients) which must observe any behavioural action (messages or quiescence). To act as an intermediary between partners, a proxy-tester must exhibit different behaviours: to receive client requests or to forward received messages to the composition, it must behave as in the specification. It must also collect, by means of input actions, the observable messages (output actions) produced by any partner. This can be expressed with a mirror specification (inputs are replaced with outputs and vice-versa). While receiving actions, we propose that it also recognises the correct messages and the incorrect ones to detect failures. As a consequence, a proxy-tester must be a combination of the specification with a mirror specification augmented with the potential incorrect behaviours of the composition. This second part corresponds to a non-conformance observer of the specification, also called canonical tester [9].

The non-conformance observer of an ioSTS, denoted NCObserve, gathers the specification transitions labelled by mirrored actions (inputs become outputs and vice versa) and transitions leading to a new location Fail, exhibiting the receipt of unspecified actions. Transitions to a Fail location are guarded by the negation of the union of guards of the same output action in outgoing transitions. Due to its extent and generality, we do not provide here the definition of the NCObserver of an ioSTS which can be found in [10]. Instead, we illustrate the NCObserver of the previous specification in Figures 3 and 4. Inputs are replaced with outputs and vice-versa. Incorrect behaviours are also added with new transitions to Fail. For instance, if we consider the location 2, new transitions to Fail are added to model the receipt of unspecified actions (messages or quiescence).

As stated earlier, a proxy-tester corresponds to a kind of fuse of an ioSTS with its NCObserver over the transition set. To express this combination without ambiguity, we initially separate locations of $S$ and $\text{NCO}(S)$ with a renaming function $\phi$. Locations are renamed by $\phi: L \rightarrow L'$, $\phi(l) \rightarrow l'$. For an ioSTS $S$, we also denote $\phi(S) = < L_S, \phi(l_S), V_S, V_0S, J_S, \Lambda_S, \rightarrow_{\phi(S)} >$.

Now, we are ready to define the proxy-tester of an ioSTS $S$:
**Fig. 3:** An ioSTS NObserver

**Definition 3 (Proxy-tester)** The proxy-tester \( P(S) \) of the specification \( S =< L_S, \overline{0}_S, V_S, V_0S, I_S, \Lambda_S, \rightarrow_S \) is a combination of \( \Delta(S) \) with its NObserver \( \phi(NCO(S)) \). \( P(S) \) is defined by an ioSTS \( < L_P, \overline{0}_P, V_P \cup \{ \text{side} \}, V_0S \cup \{ \text{side} := \prime \prime \}, I_S, \Delta(S) \cup \Lambda_{NCO}, \rightarrow_P \) such that \( L_P, \overline{0}_P \) and \( \rightarrow_P \) are constructed by the following inference rules.

![Symbol table](image)

Fig. 4: Symbol table

Intuitively, the first rule combines a specification transition and a NObserver one carrying the same mirrored actions and guards to express that if an action is received from the client environment then it is forwarded to the Web service composition. The two transitions are separated by a unique location \( (l_2 l_4, ?aGA) \). The second rule (IUT to Partner) similarly combines a specification transition and an NObserver one labelled by the same mirrored actions to express that if an action is received from the Web service composition then it is forwarded to right partner (Web service or client). Transitions labelled by \( \delta \), modelling quiescence, are also combined: so if quiescence is detected from the implementation, quiescence is also observed from the client environment. The last rule (to Fail) completes the resulting ioSTS with the transition leading to Fail of the NObserver. In each rule, a new internal variable, denoted `side`, is also added to keep track of the transitions provided by the ioSTS NObserver (with the assignment `side := "NCO"`). This distinction will be useful to define partial traces of proxy-testers.

Figure 5 depicts the resulting proxy-tester obtained from the previous specification (Figure 1) and its NObserver (Figure 3). For sake of readability, the `side` variable is replaced with solid and dashed transitions: dashed transitions stand for labelled by the assignment `side := "NCO"`. Figure 5 clearly illustrates that the initial specification behaviours are kept and that the incorrect behaviours modelled in the NObserver are present as well.

In the proxy-tester definition, transitions carrying actions provided by NCobservers are emphasised by means of the variable `side`. Specific properties on runs and traces of the proxy-tester can be deduced from this property. In particular, we can define partial runs and traces over the variable `side`.

**Definition 4 (Partial runs and traces)** Let \( P(S) =< L_S, \overline{0}_S, V_P, V_0P, I_P, \Lambda_P, \rightarrow_P \) be a proxy-tester and \( ||P(S)|| = P =< Q_P, \rightarrow_P, \sum_P, \rightarrow_P \) be its ioLTS semantics. We define \( \text{Side} : Q_P \rightarrow D_{\overline{0}_P} \), the mapping which returns the value of the side variable of a state in \( Q_P \). \( \text{Side}_E(Q_P) \subseteq Q_P \) is the set of states \( q \in Q_P \) such that \( \text{Side}(q) = E \).

Let \( \text{RUN}(P(S)) \) be the set of runs of \( P(S) \). We denote \( \text{RUN}^E(P(S)) \) the set of partial runs derived from the projection \( \text{proj}_{Q_P, \sum_P, \text{Side}_E(Q_P)}(\text{RUN}(P(S))) \).

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

For a proxy-tester \( P(S) \), we can now write \( \text{Traces}_{\text{Fail}}^{NCO}(P(S)) \) for representing the partial traces leading to Fail derived from the NObserver part. With these notations, we can deduce an interesting trace property on proxy-testers. We can write that the incorrect behaviours expressed in the NObserver with \( \text{Traces}_{\text{Fail}}^{NCO}(\phi(NCO(S))) \) can be still captured in the proxy-tester with the trace set \( \text{Traces}_{\text{Fail}}^{NCO}(P(S)) \).
Proposition 5 Let S be a specification and NCO(S), P(S) be its NCOserver and its proxy-tester respectively. We have Traces_{NCO}^{P(S)}(S) = Traces_{NCO}^{P(S)}(S).

The proof is given in an extended version of the paper in [11].

3.2 Ioco testing with proxy-testers

Formal testing methods often define the confidence degree between an implementation I and its specification S by means of a test relation. To reason about conformance and to formalise a test relation, the implementation under test is assumed to behave like its specification and is modelled by an ioLTS I. \( \Delta(I) \) represents its suspension ioLTS.

In the paper, we consider the ioco relation, which is defined as a partial inclusion of suspension traces of the implementation in those of the specification [6]. In [10], ioco has been also defined for ioSTS by making explicit the non-conformant trace set:

Definition 6 Let I be an implementation modelled by an ioLTS, and S be an ioSTS. I is ioco-conforming to S, denoted I ioco S if \( \text{Traces}(\Delta(I)) \cap NC\_Traces(\Delta(S)) = \emptyset \) with \( NC\_Traces(\Delta(S)) = Traces(\Delta(S)) \cup \Omega \setminus Traces(\Delta(S)) \).

To check if I is ioco-conforming to its specification, we can collect traces of I with proxy-testers and compare them with the specification one. However, since we have formalised proxy-testers, we can also rephrase ioco with proxy-tester traces. So, we reformulate ioco below.

Firstly, a NCoServer exhibits the incorrect behaviours of the specification with traces leading to its Fail states. However, NCO(S) is constructed by exchanging inputs and outputs symbols of its specification. If we define \( \text{refl} : (\sum^*)^* \rightarrow (\sum^*)^* \) 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:

Proposition 7 Let S be an ioSTS. The non-conformant trace set of \( \Delta(S) \), denoted NC_Traces(\( \Delta(S) \)), is equal to \( \text{refl}(\text{Traces}_{\text{Fail}}(\text{NCO}(S))) \).

We have also asserted previously that \( \text{Traces}_{\text{Fail}}(\text{NCO}(S)) = \text{Traces}_{\text{Fail}}(\text{P}(S)) \) (Proposition 5). Consequently, ioco can be formulated with Propositions 5 and 7 as:

\[ I \text{ ioco } S \Leftrightarrow \text{Traces}(\Delta(I)) \cap \text{refl}(\text{Traces}_{\text{NCO}}(\text{P}(S))) = \emptyset \]

So defined, ioco means that I is ioco-conforming to its specification when implementation traces do not belong to the set of partial proxy-tester traces leading to Fail, obtained from the NCoServer part. However, we can go farther, in the ioco rephrasing, by taking into account the parallel execution of the client environment, the proxy-tester and the Web service composition implementation. This execution can be defined by a parallel composition:

Definition 8 (Passive test execution) Let \( P = \langle Q_P, q_0P, \sum_{P}, \rightarrow_{P} \rangle \) be the ioLTS semantics of a proxy-tester P(S), and \( I = \langle Q_I, q_0I, \sum_{I}, \rightarrow_{I} \rangle \) be the implementation model. We assume that the client environment can be modelled with an ioLTS Env \( = \langle Q_{Env}, q_0_{Env}, \sum_{Env} \subseteq \sum_{P}, \rightarrow_{Env} \rangle \). The passive testing of I is defined by the parallel composition \( (Q_{Env}, P, I) \) \( = \langle Q_{Env} \times Q_P \times Q_I, q_0_{Env} \times q_0_P \times q_0_I, \sum_{Env} \subseteq \sum_{P}, \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \rangle \) given by the following rules. For readability reason, we denote an ioLTS transition \( q_1 \xrightarrow{\gamma_a} q_2 \) if Side\((q_2) = E \) (the variable side is valued to E in q2):

- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
- \( q_1 \xrightarrow{\gamma_a} \Delta(Env) \rightarrow_{||{(Env,P,I)}|\rangle \rightarrow \text{tr} ||{(Env,P,I)}|\rangle \) \( q_2 \)
The immediate deduction of the \( \rightarrow_{\text{env,P,I}} \) definition is that \( \text{Traces}(\Delta(I)) \cap \text{refl}(\text{Traces}^{\text{NCO}}_{\text{fail}}(P)) = \text{Traces}_{\text{fail}}(\Delta(I)) \cap \text{refl}(\text{Traces}^{\text{NCO}}_{\text{fail}}(\mathcal{P}(S))) \) is equivalent to \( \text{refl}(\text{Traces}_{\text{fail}}(||(\text{Env},P,I))) \) (third rule). In other terms, the non-conformant traces of \( \Delta(I) \) can be also found in \( \text{Traces}_{\text{fail}}(||(\text{Env},P,I)) \). Therefore, ioco can be finally also reformulated as:

\[ I \text{ ioco } \Leftrightarrow \text{Traces}(\Delta(I)) \cap \text{refl}(\text{Traces}^{\text{NCO}}_{\text{fail}}(\mathcal{P}(S))) = \emptyset \Leftrightarrow \text{Traces}_{\text{fail}}(||(\text{Env},P,I)) = \emptyset \]

The refl function can be removed in the last equivalence since if the set \( \text{refl}(\text{Traces}_{\text{fail}}(||(\text{Env},P,I))) \) is empty, then \( \text{Traces}_{\text{fail}}(||(\text{Env},P,I)) \) is also empty (the function \( \text{refl} \) only yields mirrored trace sets).

4. Passive tester functioning

A straightforward consequence of ioco is that non-conformance \( (I \nleftrightarrow \text{ioco } \mathcal{S}) \) is detected when a trace of the parallel composition \( ||(\text{Env},P,I) \) leads to one of its Fail states. From this assertion, we can also deduce the intuition of the proxy-tester functioning. It can be summarised by these steps: wait for an action (message or quiescence), cover some proxy-tester transitions when an action is received and construct traces, detect non-conformance when one of the proxy-tester Fail states is reached or continue.

Nevertheless, service compositions, deployed in PaaS, can be invoked concurrently by several client applications. This implies that a tester must also cope with several composition instances gathering several sessions in parallel. To extract traces from these composition instances, several proxy-tester instances, running in parallel, are also required. All of these will be managed by a unique entity that we call passive tester. Its architecture is given in Figure 6. The passive tester aims to cover, in parallel, the behaviours of several composition instances to collect traces. Incoming messages must be delivered to the correct proxy-tester instance by the passive tester. This step is performed by an entry point which routes messages by means of correlation sets.

![Figure 6: The passive tester architecture](image)

The entry point functioning is given in Algorithm 1. Each composition instance must be passively tested by a unique proxy-tester instance. Therefore, Algorithm 1 handles a set \( L \) of pairs \((pi, PV)\) with \( pi \) a proxy-tester instance and \( PV \) the set of parameter values received with messages. When a new message is received, this set is used to correlate it with an existing composition instance in reference to the Message correlation hypothesis. As stated in Section 2, the latter helps to correlate messages by assuming that a part of the correlation set of a message is composed of parameter values of messages received previously. Whenever a message \((e(p), \theta)\) is received, its correlation set \( c \) is extracted to check if a proxy-tester instance is running to accept it. This instance exists if \( L \) contains a pair \((pi, PV)\) such that a subset \( e' \subseteq c \) is composed of values of \( PV \). In this case, the correlation set has been constructed from parameter values of messages received previously. If one instance is already running, the message is forwarded to it. Otherwise, (line 7), a new one is started. If a proxy-tester instance \( pi \) returns a trace set (line 11), then the latter is stored in \( \text{Traces}(\mathcal{P}(S)) \) and the corresponding pair \((pi, PV)\) is removed from \( L \).

Algorithm 1: Proxy-tester entry point

```
input : Proxy-tester \mathcal{P}(S)
output: Traces(\mathcal{P}(S))

1 \hspace{1em} L = \emptyset;
2 \hspace{1em} \textbf{while} message \((e(p), \theta)\) \textbf{do}
3 \hspace{2em} \text{extract the correlation set } c \text{ in } \theta;
4 \hspace{2em} \text{if } \exists (pi, PV) \in L \text{ such that } e' \subseteq c \text{ and } e' \subseteq PV \text{ then}
5 \hspace{3em} \text{forward } (e(p), \theta) \text{ to } pi; \text{ PV} = PV \cup \theta;
6 \hspace{2em} \text{else}
7 \hspace{3em} \text{create a new proxy-tester instance } pi;
8 \hspace{3em} \text{forward } (e(p), \theta) \text{ to } pi;
9 \hspace{2em} \text{if } \exists (pi, PV) \in L \text{ such that } pi \text{ has returned the trace set } T \text{ then}
10 \hspace{3em} \text{Traces}(\mathcal{P}(S)) = \text{Traces}(\mathcal{P}(S)) \cup T;
11 \hspace{2em} L = L \setminus \{(pi, PV)\};
```

The proxy-tester algorithm, which aims to test passively one composition instance, is given in Algorithm 2. Both the client environment and the implementation are assumed to behave as ioLTS suspensions. The proxy-tester algorithm handles a set of runs denoted \( \text{RUNS} \). A single run is not sufficient since both the proxy-tester and the implementation may be indeterministic and may cover different behaviours. The proxy-tester algorithm is based on a forward checking approach. It starts from its initial state \( i.e., (I_0(\mathcal{P}(S)), V_0(\mathcal{P}(S))) \). Upon a received action \((e(p), \theta)\) which is either an valued action or quiescence (line 2), it looks for the next transitions which can be fired for each run \( r \) in \( \text{RUNS} \) (line 5). Each transition must have the same start location as the one found in the final state \((l, v)\) of \( r \), the same action as the received action \( e(p) \) and its guard must evaluate to true over the current internal variable value set \( v \) and the parameter values \( \theta \). If this transition leads to a Fail state then the proxy-tester algorithm adds the resulting run \( r' \) to \( \text{RUNS} \) (lines 8-11). Otherwise, the valued action \((e(p), \theta)\) is forwarded to the called partner with the next proxy-tester transition \( t_2 \) (lines 12 to 17). The new run \( r'' \) is composed of \( r' \) followed by...
the sent action and the reached state  \( q_{next_2} = (l_{next_2}, v'') \).

Once, each run of \( RUNS \) is covered, the proxy-tester waits for the next action.

Algorithm 2: Proxy-tester algorithm

```plaintext
input: A proxy-tester \( \mathcal{P}(S) \)
output: Trace set
1 \( RUNS := \langle \langle \emptyset \rangle = (l_{0}\mathcal{P}(S), V0_{\mathcal{P}(S)}) \rangle; \)
2 while Action\((e(p), \emptyset)\) do
3 \( r' = \emptyset; \)
4 foreach \( r = q_{0}, q_{1}, q_{l} \in RUNS \) with \( q_{i} = (l, v) \) do
5 \( r' = r; (e(p), \emptyset).q_{next}; \)
6 if \( r' = \emptyset \) then Fail
7 \( RUNS = (RUNS \setminus r) \cup r'; \)
8 else
9 \( \text{Execute}(t_2 = l_{next} \xrightarrow{e(p),G,A} l_{next} \in \mathcal{P}(S) \text{ such that } G \text{ evaluates to true over } \emptyset \cup v) \);
10 \( q_{next} = (l_{next}, v' = A(v \cup \emptyset)); \)
11 \( r'' = r'.(e(p), \emptyset).q_{next_2}; \)
12 \( RUNS = (RUNS \setminus r) \cup r''; \)
16 return the trace set \( T = \text{proj}_2(RUNS); \)
```

Algorithm 2 reflects exactly the parallel execution definition of Section 3.1. It actually constructs traces of \( \langle(Env, P, I) \rangle \) by supposing that \( Env \) and \( I \) are ioLTS suspensions. In particular, when a location Fail is reached (line 8), the proxy-tester has constructed a run, from its initial state which belongs to \( RUNS_{Fail} \). From this run, we obtain a trace of \( \text{Traces}_{Fail} \). So, we can state the correctness of the algorithm with:

**Proposition 9** The algorithm has detected \( \text{Fail} \) \( \Rightarrow \text{Traces}_{Fail}(\langle(Env, P, I)\rangle) \neq \emptyset \) \( \Rightarrow \neg (I \text{ ioco } S) \).

These algorithms are currently under development for two well-known Paas, Windows Azure and Google AppEngine. With Windows Azure, the entry point is implemented as a transparent proxy which behaves as it is described in Algorithm 1. No modification of composition codes is required. Web services and clients need to be configured to pass through an external proxy only. In summary, this development part is not raising particular issues. It is quite different with Google AppEngine since the use of proxy is prohibited. In this Paas, we intend to implement the entry point of the passive tester as a Java Filter Servlet. This kind of application can filter the messages exchanged between Web applications and clients. All these implementations and experimentation will be proposed in future works.

5. Conclusion

We have proposed an original approach for passive testing Web service compositions in PaaS environments. Our approach is based upon the notion of transparent proxy and is able to construct implementations traces from several composition instances deployed in virtualised environments, without requiring any modification of code. This approach also offers the advantage of checking the satisfiability of the ioco relation. An immediate line of future work is to implement the proposed passive tester and to experiment existing compositions for different Clouds, each having its own possibilities and restrictions.

References