A Model-based testing approach combining passive testing and runtime verification. Application to Web service composition testing in Clouds

Sébastien Salva

Research Report LIMOS/RR-13-04

10 avril 2013

1. LIMOS, UMR 6158, PRES Clermont-Ferrand University, FRANCE, sebastien.salva@udamail.fr
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 ioco-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 basic proxy to collect traces, we propose to generate a formal model from the specification, called Proxy-monitor, which can act 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, ioSTS, ioco, Clouds, PaaS layers
1 Introduction

Software testing is more and more considered as an essential activity in Software engineering by IT (Information technologies) companies. Software testing is a large process that is 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, this testing process is performed with active approaches: basically, test cases are constructed from the specification and are 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 [LS09].

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 [Sch00]. We combine this monitoring approach with a previous work delating with ioco passive testing [Sal12]. Ioco [Tre96] 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 proper-
ties,

2. Trace extraction: to collect traces, it is required to have an open testing environment where tools, workflow engines or frameworks can be installed. More and more frequently over recent years, the real implementation environment access is restricted. For instance, Web server accesses are often strictly limited for security reasons. And these restrictions prevent from installing monitors to collect traces. 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, whose locations and details are not known, 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 and basic proxy to collect traces, we propose to generate a formal model from the specification, called Proxy-monitor, which can act as a proxy and which can directly detect implementation errors,

3. 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 analysis (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 analysis, 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 which lead to the Proxy-monitor model is defined in Section 4. We apply the concept of Proxy-monitor on Web service compositions in Section 5. We review some related works on passive testing and runtime verification in Section 6. Finally, Section 7 concludes the paper.

2 Model Definition and notations

To describe 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 two sets of variables and with guards and assignments on transitions, giving the possibilities to model the system states and constraints on actions. The fact of using symbolic variables helps to describe infinite state transition systems in a finite manner. These potentially infinite behaviours can be expressed by the semantics of an ioSTS, given in
terms of input/output Labelled Transition Systems (ioLTS). This model offers the advantage of reusing the iooco theory [Tre96] which defines conformance as a partial inclusion of observable behaviours (suspension traces) of the implementation in those of the specification.

2.1 The IOSTS model

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 \( v: Y \rightarrow D \). 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 \( v: T(X) \rightarrow D \). The set of the first order formula \( G \) over \( Y \subseteq X \cup T(X) \) is denoted \( F(Y) \). \( v \models G \) denotes a formula \( G \) which evaluates to true with respect to the valuation \( v \).

With ioSTSs, the action set is separated with inputs beginning by ? to express actions expected by the system, and with outputs beginning by ! to express actions produced 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) A deterministic Input Output Symbolic Transition System (ioSTS) 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 parameter ones. The internal variables are initialised with the valuation \( V_0 \in D_V \), which is assumed to be unique,
- \( \Lambda \) is the finite set of symbolic actions \( a(p) \), with \( p = (p_1, \ldots, p_k) \) a finite set of parameter variables, called the signature of \( a \), in \( I^k \) (\( k \in \mathbb{N} \)). The signature of \( a \) is assumed unique: so, if \( a(p) \in \Lambda \), then \( a(p') \), with \( p' \neq p \) does not belong to \( \Lambda \).
- \( \Lambda \) is partitioned by \( \Lambda = \Lambda^I \cup \Lambda^O \): \( \Lambda^I \) (\( \Lambda^O \)) represents the set of input actions beginning with ? (the set of output actions beginning with ! respectively).
- \( \rightarrow \) 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 \( l_i \stackrel{a(p), G, A}{\rightarrow} l_j \), is labelled by an (input or output) action \( a(p) \in \Lambda \), \( G \in \mathcal{F}(p \cup V \cup T(p \cup V)) \) is a guard which restricts the firing of the transition. Once the transition is fired, internal variables are updated with a set \( A \) of assignments of the form \((v := Av)_{v \in V}\) such that for each variable \( v \), \( Av \) is an expression on \( V \cup p \cup T(p \cup V) \).
The distinction made between internal and interaction variables is convenient to clearly express the state of the system (internal variables) and to model complex actions composed with communication parameters (interaction variables).

The ioSTS suspension is an immediate ioSTS extension, which also expresses quiescence i.e., the absence of observation from a location. The ioSTS suspension offers the advantage of expressing when it is allowed to have a system in a deadlock state and of detecting, with conformance testing, unauthorised deadlocks in the system under test. Usually, quiescence is observed on implementations with timers: after each sending of action to the implementation, a timer is reset. If it expires, then quiescence is observed. Timers are assumed initialised with a duration sufficiently large to ensure that any output action, provided by the implementation, can be observed.

Quiescence is modelled by a new symbol !δ and an augmented ioSTS denoted \( \Delta(\text{ioSTS}) \). For an ioSTS \( S \), \( \Delta(S) \) is obtained by adding a self-loop labelled by !δ for each location where quiescence may be observed. This is defined by:

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

\[
\begin{array}{c}
 l_1 \in L, a \in \Lambda^0, G_a = \bigvee_{l_2} (\exists v \in D_p).G \\
 l_1 \vdash_{\Delta(S)} !\delta, A' \quad \quad l_1, G' = \bigwedge_{a \in \Lambda^0} \neg G_a, A' = (x := x)_{x \in V}
\end{array}
\]

**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 \( op \) (\( a(p) = op\text{Req}(p) \)), or the receipt of an operation response (\( a(p) = op\text{Resp}(p) \)), 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, the usual technical solution is to add, in messages, correlation values which match a part of the session state [MC11]. So, messages 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 coor \( \in p \).
The use of correlation sets with ioST斯 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 $opReq(p)$, expressing an operation call, must contain a correlation set $coor \subseteq p$ such that a subset $c \subseteq coor$ 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 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 correlation sets. Each session is identified with its own correlation set e.g., BookSeller with $c_{1}=\{\text{account} = "custid"\}$, and Wholesaler with $c_{2} = \{\text{account} = "custid", isbn="2070541274"\}$. 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 corresponds to a valued automaton: the ioLTS states are labelled by internal variable valuations while transitions are labelled by actions and interaction variable valuations.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Message</th>
<th>Guard</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>!BOrder</td>
<td>!BookOrderReq(List Books, quantity, account, from, to, corr)</td>
<td>G1=[from=&quot;Client&quot; ∧ to=&quot;BR&quot; ∧ corr = {account}]</td>
<td>q:=quantity, b:=ListBooks, c1:=corr</td>
</tr>
<tr>
<td>!BOrder2</td>
<td>!BookOrderReq(List Books, quantity, account, from, to, corr)</td>
<td>¬G1</td>
<td></td>
</tr>
<tr>
<td>!WSReq</td>
<td>!WholeSalerReq(isbn, from, to, corr)</td>
<td>G2=[isbn=b[q]∧ q ≥ 1 ∧ from=&quot;BR&quot; ∧ to=&quot;WS&quot; ∧ corr = {a.isbn}]</td>
<td>q := q − 1</td>
</tr>
<tr>
<td>!BOrder Resp</td>
<td>!BookOrderResp(resp, from, to, corr)</td>
<td>G3=[from=&quot;BR&quot; ∧ to=&quot;Client&quot; ∧ resp=&quot;Order done&quot; ∧ corr=c1]</td>
<td></td>
</tr>
<tr>
<td>?R1</td>
<td>?BookOrderResp ?WholeSalerReq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?R2</td>
<td>?BookOrderResp ?WholeSalerReq ¬G3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!BOrder Req'</td>
<td>!BookOrderReq(List Books, quantity, account)</td>
<td>G1’=[quantity ≥ 1]</td>
<td></td>
</tr>
<tr>
<td>!BOrder Resp'</td>
<td>!BookOrderResp(resp)</td>
<td>G3’=[end(resp)=&quot;done&quot;]</td>
<td></td>
</tr>
<tr>
<td>!BOrder [G1’]</td>
<td>!BookOrderReq(List Books, quantity,)</td>
<td>G1’</td>
<td></td>
</tr>
<tr>
<td>!BOrder Resp(G3∧ G3')</td>
<td>!BookOrderResp(resp)</td>
<td>G3’∧G3’</td>
<td></td>
</tr>
<tr>
<td>!BOrder Resp(¬G3∧ G3')</td>
<td>!BookOrderResp(resp, from, to, corr)</td>
<td>¬G3∧G3’</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2:** Symbol table
**Definition 3 (ioLTS semantics)** The semantics of an ioSTS $S = <L, l_0, V, V_0, I, \Lambda, \rightarrow>$ is the ioLTS $|S| = <Q, q_0, \Sigma, \rightarrow>$ where:
- $Q = L \times D_V$ is the set of states,
- $q_0 = (l_0, V_0)$ is the initial state,
- $\Sigma = \{(a(p), \theta) | a(p) \in \Lambda, \theta \in D_\theta\}$ is the set of valued symbols. $\Sigma^I$ is the set of input actions and $\Sigma^O$ is the set of output ones,
- $\rightarrow$ is the transition relation $Q \times \Sigma \times Q$ deduced by the following rule:

$$\frac{l_1 \leftarrow a(p).G_A} {l_2, \theta \in D_\theta, v \in D_V, v' \in D_V, v \cup \theta = G, v' = A(v, \theta)}$$

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 $\theta$ 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 $A$ over $v \cup \theta$. An ioSTS suspension $\Delta(S)$ is also associated to its ioLTS semantics suspension by $||\Delta(S)|| = \Delta(||S||)$.

Runs and traces of ioSTSs, which represent executions and action sequences, can also be derived from the ioLTS semantics:

**Definition 4 (Runs and traces)** For an ioSTS $S = <L, l_0, V, V_0, I, \Lambda, \rightarrow>$, interpreted by its ioLTS semantics $|S| = <Q, q_0, \Sigma, \rightarrow>$, a run $q_0 \alpha_0...\alpha_{n-1} q_n$ is an alternate sequence of states and valued actions. $RUN(S) = RUN(|S|)$ is the set of runs found in $|S|$. $RUN_F(S)$ is the set of runs of $S$ finished by a state in $F \times D_V \subseteq Q$ with $F$ a location set in $L$.

It follows that a trace of a run $r$ is defined as the projection $\text{proj}_r(q)$ on actions. $\text{Traces}_F(S) = \text{Traces}_F(|S|)$ 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 5 (Compatible ioSTSs)** An ioSTS $S_1 = <L_1, l_1^0, V_1, V_1^0, I_1, \Lambda_1, \rightarrow_1>$ is compatible with $S_2 = <L_2, l_2^0, V_2, V_2^0, I_2, \Lambda_2, \rightarrow_2>$ iff $V_1 \cap V_2 = \emptyset$, $\Lambda_1^I = \Lambda_2^I$, $\Lambda_1^O = \Lambda_2^O$ and $I_1 = I_2$.

**Definition 6 (Parallel product $|$)** The parallel product of two compatible ioSTSs $S_1 = <L_1, l_1^0, V_1, V_1^0, I_1, \Lambda_1, \rightarrow_1>$ and $S_2 = <L_2, l_2^0, V_2, V_2^0, I_2, \Lambda_2, \rightarrow_2>$, denoted $S_1 | S_2$, is the ioSTS $P = <L_p, l_p^0, V_p, V_p^0, I_p, \Lambda_p, \rightarrow_p>$ such that $V_p = V_1 \cup V_2$, $I_p = I_1 \cup I_2$, $\Lambda_p = \Lambda_1 \cup \Lambda_2$. 


\[ V_P^0 = V_1^0 \land V_2^0, \ I_P = I_1 = I_2, \ L_P = L_1 \times L_2, \ \iota_P^0 = (\iota_1^0, \iota_2^0), \ \Lambda_P = \Lambda_1 = \Lambda_2. \] The transition set \( \to_P \) is the smallest set satisfying the following rule:

\[
\frac{I_l \xrightarrow{a(p), G_1, A_1} S_1 I_l'}{S_1 I_l'} \quad \frac{I_l \xrightarrow{a(p), G_2, A_2} S_2 I_l'}{S_2 I_l'} \quad \frac{I_l \xrightarrow{a(p), G_1 \land G_2, A_1 \cup A_2} P (I_l L_2')}{P (I_l L_2')}
\]

The traces of a parallel product of two ioSTSs are given by the intersection of the trace sets of the two ioSTSs.

**Lemma 1 (Parallel product traces)** \( \text{Traces}_{F_1 \times F_2} (S_1 || S_2) = \text{Traces}_{F_1} (S_1) \cap \text{Traces}_{F_2} (S_2) \) with \( F_1 \subseteq L_{S_1}, \ F_2 \subseteq L_{S_2} \)

We end this Section with the definition of the ioSTS operation \( \text{refl} \) which exchanges input and output actions of an ioSTS.

**Definition 7 (Mirrored ioSTS and traces)** Let \( S \) be an ioSTS. \( \text{refl} (S) = \text{def} < L_{S}, I_{S}^0, V_{S}, \iota_{S}^0, \Lambda_{S}, \to_{S} > \) where \( \Lambda_{\text{refl}} (S) = \Lambda_{S}^0, \ \Lambda_{\text{refl}} (S) = \Lambda_{S}^I \).

We extend the \( \text{refl} \) notation on trace sets. \( \text{refl} : (\Sigma^*)^* \to (\Sigma^*)^* \) 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 model-based testing, one assume that the functional behaviours of the implementation can be modelled with an ioLTS \( I \) which is unknown and which provides exactly the same observations as the implementation. This classical assumption is required to formally define violations or fulfilment of implementations against properties or specifications. \( I \) is also assumed to have the same interface as the specification (actions with their parameters) and is input-enabled, i.e., it accepts any output actions.

#### 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 [CW02, CJMR07] 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 8 (Observer)** An Observer is a deterministic ioSTS $O$ composed of a non empty set of violation locations $\text{Violate}_O \subset L_O$. $O$ must be both input and output-enabled, i.e. for each state $(l,v) \in L_O \times D_O$, and for each valued action $(a(p),\theta) \in A_O \times D_p$, it exists $(l,v) \xrightarrow{a(p).\theta} (l',v') \in \rightarrow_{||O||}$.

Given an ioSTS $S$, $\text{Comp}(S)$ stands for the set of compatible Observers of $S$.

We shall also say that $\text{Traces}_{\text{Violate}_O}(O)$ gathers all the traces in $\text{Traces}(O)$ which violate $O$.

For a specification $S$, the ioSTS Observer $O$ has to be input and output-enabled and compatible with the specification $S$. These assumptions are required to model a safety property which is violated by all the traces in $\text{Traces}_{\text{Violate}_O}(O)$ and which is satisfied by all the traces in $(\sum_{||S||})^* \setminus \text{Traces}_{\text{Violate}_O}(O)$.

Given an implementation $I$, we say that $I$ satisfies the Observer $O$ if $I$ does not yield any trace which also violates $O$:

**Definition 9 (Implementation satisfies Observer)** Let $S$ be an ioSTS and $I$ an implementation. $I$ satisfies the Observer $O \in \text{Comp}(\Delta(S))$, denoted $I \models O$, if $\text{Traces}(\Delta(I)) \cap \text{Traces}_{\text{Violate}_O}(O) = \emptyset$.

Figures 3 and 2 illustrate an example of Observer that expresses a safety property for the specification of Figure 1. It means that "the receipt of an order confirmation ending with "done", without requesting WholeSaler, must never occurs".

**Figure 3:** A safety property

Two Observers $O_1$ and $O_2$, describing two different safety properties can be interpreted by the Observer $O_1 \parallel O_2$ composed of a Violation location set $\text{Violate}_{O_1} \times$
A location in $\text{Violate}_{O_1} \times \text{Violate}_{O_2}$ denotes either the violation of the first property, or of the second one or of both. 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, we consider the *ioco* test relation [Tre96], which intuitively means that $I$ is ioco-conforming to its specification $S$ if, after each trace of the ioSTS suspension $\Delta(S)$, $I$ only produces outputs (and quiescence) allowed by $\Delta(S)$. For ioSTSs, ioco is defined as:

**Definition 10** Let $I$ be an implementation modelled by an ioLTS, and $S$ be an ioSTS. $I$ is ioco-conforming to $S$, denoted $I$ ioco $S$ iff $\text{Traces}(\Delta(S)) \cap (\Sigma^O \cup \{!\delta\}) \subseteq \text{Traces}(\Delta(I))$. We have shown in our previous work [Sal12] 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. For instance, used with PaaS environments, it offers the advantage of collecting any message exchanged between clients and Web applications without requiring to modify the PaaS environment, the service codes and without installing a sniffer-based tool. Web applications only have to be configured to pass through a proxy. We rewrite here the notion of Proxy-tester to ease the combination of proxy-testing with runtime verification in the next section.

To collect the events observed from an implementation and to detect non-conformance, a Proxy-tester is constructed from a specific model called Canonical tester, itself derived from a specification. A Canonical tester is an ioSTS which gathers the specification transitions labelled by mirrored actions (inputs become outputs and vice-versa) and transitions leading to a new location $\text{Fail}$, exhibiting the receipt of unspecified actions. For an ioSTS $S$, the Canonical tester of $S$ corresponds to $\text{refl}(S)$ completed on its input actions. It is defined as:

**Definition 11 (ioSTS Canonical Tester)** Let $S = < L_S, I_0, V_S, V_0, I_S, A_S, \rightarrow_S >$ be an ioSTS and $\Delta(S)$ be its suspension. The Canonical tester of $S$ is the ioSTS $\text{Can}(S) = < L_S \cup \text{LF}_{\text{Can}(S)}, I_0, V_S, V_0, I_S, A_{\text{refl}(S)}, \rightarrow_{\text{Can}(S)} >$ such that $\text{LF}_{\text{Can}(S)} = \{\text{Fail}\}$ is the Fail location set composed here of the Fail location. $\rightarrow_{\text{Can}(S)}$ is defined by the rules:
As an example, the Canonical tester of the ioSTS depicted in Figure 1 is illustrated in Figures 4 and 2. If we consider the location 2, new transitions to \textit{Fail} are added to model the receipt of unspecified events (messages or quiescence).

![Diagram of an ioSTS Canonical tester](image)

Figure 4: An ioSTS Canonical tester

Canonical testers are composed of transitions exhibiting the receipt of unspecified actions and also recognise non-conformant behaviours of specifications in \textit{Fail} states. Ioco can be rewritten with Canonical traces by considering that $\text{Can}(S)$ is constructed by exchanging inputs and outputs symbols of its specification:

**Proposition 1**

$I \text{ ioco } S \iff \text{Traces}(\Delta(I)) \cap \text{refl}(\text{Traces}_{\text{Fail}}(\text{Can}(S))) = \emptyset$

Now, we are ready to define the Proxy-tester of an ioSTS $S$. It corresponds to a Canonical tester where all the transitions, except those leading to \textit{Fail}, are \textit{doubled} to express the receipt of an event and its forwarding to its real addressee.

**Definition 12 (Proxy-tester)** The Proxy-tester of the ioSTS $S = < L_S, l_0_S, V_S, V_0_S, I_S, A_S, \rightarrow_S >$ is the ioSTS $Pr(\text{Can}(S))$ where $Pr$ is an ioSTS operation such that $Pr(\text{Can}(S)) = \text{def } < l_0_S \cup LF_p, V_{\text{Can}(S)}, V_0_{\text{Can}(S)} \cup \{ \text{side}, pt \}, V_0_{\text{Can}(S)} \cup \{ \text{side := "}", pt := "}\}, I_{\text{Can}(S)}, A_p, \rightarrow_p >$. $LF_p = LF_{\text{Can}(S)} = \{ \text{Fail} \}$ is the \textit{Fail} location set.
\( L_p, \Lambda_p \) and \( \rightarrow_p \) are constructed with the following rules:

\[
I_1 \xrightarrow{!a(p), G, A} Can(S) I_2, I_2 \notin LF_{Can(S)}
\]

\[
\vdash I_1 \xrightarrow{?a(p), G, A \cup \{pt:=p, side:=\text{""}\}} \rightarrow_p (I_1, I_2, a(p), G)
\]

\[
!a(p), [p=p_t], \{ (x:=x) \in V_{Can(S)}, side:=\text{"Can"}, pt:=pt \} \xrightarrow{\rightarrow_p} I_2
\]

\[
I_1 \xrightarrow{?a(p), G, A} Can(S) I_2, I_2 \notin LF_{Can(S)}
\]

\[
\vdash I_1 \xrightarrow{?a(p), G, A \cup \{pt:=p, side:=\text{"Can"}\}} \rightarrow_p (I_1, I_2, a(p), G)
\]

\[
!a(p), [p=p_t], \{ (x:=x) \in V_{Can(S)}, side:=\text{""}, pt:=pt \} \xrightarrow{\rightarrow_p} I_2
\]

\[
I_1 \xrightarrow{a(p), G, A} Can(S) I_2, I_2 \in LF_{Can(S)}
\]

\[
\vdash I_1 \xrightarrow{a(p), G, A \cup \{side:=\text{"Can"}, pt:=pt\}} \rightarrow_p I_2
\]

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 \( (I_1, I_2, a(p), G) \) 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 \( side:=\text{"Can"} \)). This distinction will be useful to define partial traces of Proxy-testers and to express conformance with them.

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 Canonical tester of the specification which is completed on the incorrect behaviour set: hence, both the Canonical tester and 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 5 depicts the resulting Proxy-tester obtained from the previous specification (Figure 1) and its Canonical tester (Figure 4). For readability, the \( side \)
variable is replaced with solid and dashed transitions: dashed ones stand for transitions labelled by the assignment($side := Can$). Figure 5 clearly illustrates that the initial specification behaviours are kept and that the incorrect behaviours modelled in the Canonical tester are present as well.

![Diagram](image)

Figure 5: A Proxy-tester

The following proposition, which is a immediate consequence of the definition of the operation $Pr$, shows that the Proxy-tester of one ioSTS, is unique.

**Proposition 2** Let $S$ be an ioSTS. $Pr^{-1}$ is the inverse operation of $Pr$. We have $Pr^{-1}(Pr(S)) = S$

The definition of $Pr^{-1}$ (with the proof of the previous Proposition) is given in Appendices.

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

**Definition 13 (Partial runs and traces)** Let $\mathcal{P}$ be a Proxy-tester and $||\mathcal{P}|| = P = < Q_P, q_0, \Sigma_p, \rightarrow_P >$ be its ioLTS semantics. We define Side : $Q_P \rightarrow D_{V_P}$ 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 $\text{RUN}(\mathcal{P})$ be the set of runs of $\mathcal{P}$. We denote $\text{RUN}^E(\mathcal{P})$ the set of partial runs derived from the projection $\text{proj}_{Q_P, \Sigma_p \text{Side}^E(Q_P)}(\text{RUN}(\mathcal{P}))$.

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

For a Proxy-tester $\mathcal{P}$, we can now write $\text{Traces}^{\text{Can}}_{\text{Fail}}(\mathcal{P})$ for representing the partial traces leading to Fail derived from the Canonical tester part.

For instance, in the Proxy-tester of Figure 5, ![BookOrder("2070541274"), 1, "custid"]?δ belongs to $\text{Traces}^{\text{Can}}_{\text{Fail}}(\mathcal{P})$.

With these notations, we can write that the behaviours expressed in the Canonical tester with $\text{Traces}(\text{Can}(S))$ still exist in its Proxy-tester and are expressed by the trace set $\text{Traces}^{\text{Can}}_{\text{Fail}}(\text{Pr}(\text{Can}(S)))$.

**Proposition 3** Let $S$ be an ioSTS. We have

$\text{Traces}^{\text{Can}}(\text{Pr}(\text{Can}(S))) = \text{Traces}(\text{Pr}^{-1}(\text{Pr}(\text{Can}(S)))) = \text{Traces}(\text{Can}(S))$.

In particular, $\text{Traces}^{\text{Can}}_{\text{Fail}}(\text{Pr}(\text{Can}(S))) = \text{Traces}^{\text{Can}}_{\text{Fail}}(\text{Can}(S))$.

With the previous Propositions, $ioco$ can now be rephrased by:

**Proposition 4**

$I \ ioco \ S \iff \text{Traces}(\Delta(I)) \cap \text{refl}(\text{Traces}^{\text{Can}}_{\text{Fail}}(\text{Pr}(\text{Can}(S)))) = \emptyset$

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 Canonical part. So defined, one can deduce that when Fail is reached while the Proxy-tester execution then $ioco$ does not hold.

### 4 Combining runtime verification and proxy-testing

This section presents a method that gathers passive testing and runtime verification. Intuitively, the method relies upon the concept of Proxy-testing previously defined. To monitor an implementation against a set of safety properties, the latter will be injected into Proxy-testers to form a model called Proxy-monitor. This ioSTS reflects three concepts: proxy functioning with collect of traces, detection of incorrect behaviours and detection of property violations.
4.1 Proxy-tester and Observer composition

Proposition 1 reveals that Canonical testers are enough for detecting all the implementations that are not ioco-conforming to a given specification. 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. The result of the product of one Canonical tester $\text{Can}(S)$ with an Observer $O$, 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 14 (Monitor)** Let $\Delta(S)$ be an ioSTS suspension and $O \in \text{Comp}(\Delta(S))$ be an Observer. The Monitor of the Canonical tester $\text{Can}(S)$ and of the Observer $O$ is the ioSTS $M = \text{Can}(S)||\text{refl}(O)$.

As an example, the Monitor constructed from the previous Canonical tester (Figure 4) and the Observer of Figure 3 is depicted in Figures 6 and 2. It contains different verdict locations: Fail received from the Canonical tester, Violate received from the Observer and a combination of both Fail/Violate. (Fail,Violate) denotes non-conformance and the safety property violation. For example, the trace "?BookOrder(1,"custid") !BookOrderResp("done")" violates the Observer of Figure 3 because WholeSalers is not called. This trace reflects also an incorrect behaviour because the response "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 15 (Verdict location sets)** Let $\text{Can}(S)$ be a Canonical tester and $O \in \text{Comp}(\Delta(S))$ be a compatible Observer with $\Delta(S)$. The parallel product $M = \text{Can}(S)||\text{refl}(O)$ produces several sets of verdict locations defined as follows:

1. $\text{VIOLATE} = (L_{\text{Can}(S)} \setminus \{\text{Fail}\}) \times \text{Violate}_O$.
2. $\text{FAIL} = \{\text{Fail}\} \times (L_O \setminus \text{Violate}_O)$.
3. $\text{FAIL/VIOLATE} = \{(\text{Fail}, \text{Violate}_O)\}$.

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

As for the parallel product $||$ (Definition 6), the traces of a Monitor $M = \text{Can}(S)||\text{refl}(O)$ can be expressed with the traces of the composed ioSTFs.
Lemma 2 (Monitor traces) Let $\mathcal{M}$ be a Monitor constructed from a Canonical tester $\text{Can}(S)$ and a compatible Observer $\emptyset \in \text{Comp}(\Delta(S))$. We have,

$$\text{Traces}_{F1 \times F2}(\mathcal{M}) = (\text{Traces}_{F1}(\text{Can}(S)) \cap \text{Traces}_{F2}(\text{refl}(\emptyset)))$$

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

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

Proxy-monitors are constructed as Proxy-testers except that Fail location sets are different. For a Proxy-tester, there is only one Fail location, whereas a Proxy-monitor has a Fail location set $LF_M$ equals to $FAIL \cup FAIL/VIOLEATE$ 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.

As an example, Figure 7 depicts the Proxy-monitor of the Monitor illustrated in Figure 6. For readability, the side variable is replaced with solid and dashed
transitions: dashed ones stand for transitions labelled by the assignment \( \text{side} := \text{Can} \). This Proxy-monitor reflects the functioning of a transparent proxy and the monitoring of the implementation. Events collected by the Proxy-monitor are forwarded to their recipients. The monitoring is stopped once a location in \( \text{FAIL} \cup \text{FAIL/VIOlate} \) is reached. When a location in \( \text{VIOlate} \) is reached, the monitoring still continues though. This choice was made in the previous definitions on account of the nature of safety properties. The latter can be composed of specification properties or not. When a violation of a safety property is detected, the implementation is not necessarily incorrect. So, we chose to not stop the Proxy-monitor (hence the message forwarding to the implementation) abruptly.

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 with mirrored actions. We assume that this external environment

Figure 7: A Proxy-monitor
can be also modelled with an ioLTS suspension $Env$ such that $\text{refl}(Env)$ is compatible with $I$ and $\text{Traces}(\Delta(Env))$ is composed of sequences in $\text{refl}(\langle \sum_{\Delta(I)} \rangle^*)$.

**Definition 17 (Implementation monitoring)** Let $PM = \langle Q_{PM}, q_{0_{PM}} \rangle$.  
$\sum_{PM} \rightarrow_{PM}$ be the ioLTS semantics of a Proxy-monitor $Pr(M)$ derived from an ioSTS $S$ and an Observer $\emptyset \in \text{Comp}(\Delta(S))$. $Q_{FM} \subseteq Q_{PM} = LF_{Pr(M)} \times D_{V_{Pr(M)}}$ is its Fail state set. $I = \langle Q_I, q_{0_I}, \sum_I \subseteq \sum_M, \rightarrow_I \rangle$ is the implementation model, assumed compatible with $S$ and $Env = \langle Q_{Env}, q_{0_{Env}}, \sum_{Env} \subseteq \sum_P, \rightarrow_{Env} \rangle$ is the ioLTS modelling the external environment compatible with $\text{refl}(I)$.

The monitoring of $I$ by $Pr(M)$ is expressed with the product $|p|(Env, PM, I) = \langle Q_{Env} \times Q_{PM} \times Q_I, q_{0_{Env}} \times q_{0_{PM}} \times q_{0_I}, \sum_{PM} \rightarrow |p|(Env, PM, I) \rangle$ 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 $q_1 \xrightarrow{a_{\text{Env}}} q_2$ if Side($q_2$) = $E$ (the variable side is valued to $E$ in $q_2$).

<table>
<thead>
<tr>
<th>Transition</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1 \xrightarrow{a_{\Delta(Env)}} q_2 \xrightarrow{a_{\Delta(I)}} q_3 \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_3$</td>
</tr>
<tr>
<td>$q_1 q_2'' \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_2''$</td>
</tr>
<tr>
<td>$q_2 \xrightarrow{a_{\Delta(Env)}} q_3 \xrightarrow{a_{\Delta(I)}} q_4 \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_4$</td>
</tr>
<tr>
<td>$q_2 q_4'' \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_4''$</td>
</tr>
<tr>
<td>$q_2 \xrightarrow{a_{\Delta(Env)}} q_3 \xrightarrow{a_{\Delta(I)}} q_4 \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_4$</td>
</tr>
<tr>
<td>$q_2 q_4'' \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_4''$</td>
</tr>
<tr>
<td>$q_2 \xrightarrow{a_{\Delta(Env)}} q_3 \xrightarrow{a_{\Delta(I)}} q_4 \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_4$</td>
</tr>
<tr>
<td>$q_2 q_4'' \xrightarrow{a_{\text{Can}}}$</td>
<td>$\sum_{PM} \rightarrow_{PM} q_4''$</td>
</tr>
</tbody>
</table>

One can deduce from the $\rightarrow |p|(Env, PM, I)$ definition that:

**Proposition 5** We consider the notations of Definition 17. We have $\text{Traces}^\text{Can}(|p|(Env, PM, I)) = \text{refl}(\text{Traces}(\Delta(I))) \cap \text{Traces}^\text{Can}(PM) = \text{refl}(\text{Traces}(\Delta(I))) \cap \text{Traces}^\text{Can}(Pr(M))$

The verdict list can now be drawn up from Definition 15. 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 6 (Test verdicts)** Consider an external environment $Env$, an implementation $I$ monitored with a Proxy-monitor $Pr(M)$, itself derived from an ioSTS $S$ and an Observer $\emptyset \in \text{Comp}(\Delta(S))$. Let $OT \subseteq \text{Traces}(|p|(Env, PM, I))$ be the observed trace set. If it exists $\sigma \in OT$ such that:

1. $\sigma$ belongs to $\text{Traces}_{\text{FAIL/VIOLATE}}(|p|(Env, PM, I))$, then $I$ does not satisfy the safety property and $I$ is not ioSTS-conforming to $S$,  

19
2. $\sigma$ belongs to $\text{Traces}_{\text{FAIL}}(\|p(\text{Env}, \text{PM}, I))$, then $I$ is not ioco-conforming to $S$. No violation of the safety property were detected on $I$.

3. $\sigma$ belongs to $\text{Traces}_{\text{VIOLATE}}(\|p(\text{Env}, \text{PM}, I))$, then $I$ does not satisfy the safety property. Non-conformance between $I$ and $S$ were not detected.

5 Application to Web service composition deployed in Clouds

![Figure 8: The passive tester architecture](image)

Here, we consider having a Web service composition deployed in a PaaS environment. The framework, whose architecture is depicted in Figure 8, aims to collect all the traces of Web service composition instances. We assume that each partner participating to the composition (Web services and clients) are configured to pass through the passive tester which is mainly based upon Proxy-monitors and which acts as a proxy. To collect traces from several instances and to detect non-conformance or violations of safety properties, several Proxy-monitor instances are executed in parallel. Any incoming message received from the same composition instance must be delivered to the same Proxy-monitor instance: this step is performed by a module called *entry point* which routes messages to Proxy-monitor instances by means of correlation sets.

The entry point functioning is given in Algorithm 1. The latter handles a set $L$ of pairs $(p_i, PV)$ with $p_i$ a Proxy-monitor instance identifier and $PV$ the set of parameter values received in previous 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. Whenever a message $(e(p), \theta)$ is received, its correlation set $c$ is extracted to check if a Proxy-monitor instance is running to accept it. This instance exists if $L$ contains a pair $(p_i, PV)$ such that a 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 one instance is already running, the message is forwarded to it. Otherwise, (line 7), a new one is started. If a monitor instance $p_i$ returns a trace set (line 11), then the latter is stored in $Traces(Pr(M))$ and the corresponding pair $(p_i, PV)$ is removed from $L$.

**Algorithm 1:** Entry point

```
input : Proxy-monitor $Pr(M)$
output: $Traces(Pr(M))$
1 $L = \emptyset$;
2 while message $(e(p), \theta)$ do
3     extract the correlation set $c$ in $\theta$;
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 Proxy-monitor instance $p_i$;
8         $L = L \cup (p_i, \{\theta\})$; forward $(e(p), \theta)$ to $p_i$;
9     if $\exists (p_i, PV) \in L$ such that $p_i$ has returned the trace set $T$ then
10        $Traces(Pr(M)) = Traces(Pr(M)) \cup T$;
11        $L = L \setminus \{(p_i, PV)\}$;
```

Algorithm 2 describes the functioning of one Proxy-monitor instance. Basically, it aims to wait for an event (message or quiescence), to cover the Proxy-monitor transitions, to construct traces and 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 its initial state i.e., $(l_0Pr(M), V0Pr(M))$ and constructs runs stored in the $RUNS$ set. Whenever an event $(e(p), \theta)$ is received (valued action or quiescence), with eventually $\theta$ a valuation over $p$ (line 3), it looks for the next transitions which can be fired for each run $r$ in $RUNS$ (line 7). Each transition must have the same start location as the one found in the final state $(l, v)$ of the run $r$, the same action as the received event $e(p)$ and its guard must be satisfied over the valuation $v \cup \theta$. If this transition reaches a verdict location (Definition 15) then the algorithm adds the resulting run $r'$ to $RUNS$ (lines 11-16). Otherwise, the event $(e(p), \theta)$ is forwarded to the called partner with the next transition $t_2$ (lines 17 to 21). The new run $r''$ is composed of $r'$ followed by the sent event and the reached state $q_{next2} = (l_{next2}, v'')$. Then, the algorithm waits for the next event. The algorithm ends when Fail is detect 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 set \( T \) derived from \( RUNS \).

**Algorithm 2: Proxy-Monitor algorithm**

<table>
<thead>
<tr>
<th>input</th>
<th>A Proxy-monitor ( Pr(M) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Trace set</td>
</tr>
</tbody>
</table>

1. \( RUNS := \{(q_0 = (l_{0Pr(M)}, V_{0Pr(M)}))\}; \)
2. \( qt = 0; \)
3. while Event(\( e(p), \theta \)) \& Fail is not detected \& \( qt < 10 \) do
   4. \( r' = \emptyset; \)
   5. if \( e(p) = !\delta \) then
      6. \( qt := qt + 1; \)
   7. foreach \( r = q_0a_0...q_i \in RUNS \) with \( q_i = (l, v) \) do
      8. foreach \( t = l \xrightarrow{e(p),G} l' \) \( \in Pr(M) \) such that \( \theta \cup v = G \) do
         9. \( q_{next} = (l_{next}, v' = A(v \cup \theta)); \)
         10. \( r' = r.(e(p), \theta).q_{next}; \)
         11. if \( l_{next} \in VIOLATE \cup FAIL/VIOLATE \) then
             12. Violation is detected;
             13. \( RUNS = (RUNS \setminus r) \cup r'; \)
         14. if \( l_{next} \in FAIL \cup FAIL/VIOLATE \) then
             15. Fail is detected;
             16. \( RUNS = (RUNS \setminus r) \cup r'; \)
         17. if \( l_{next} \notin VIOLATE \cup FAIL/VIOLATE \cup FAIL \) then
             18. Execute(\( t_2 = l_{next} \xrightarrow{e(p),G_2} l_{next2}\)); \( \quad \) // forward (\( !e(p), \theta \))
             19. \( q_{next2} := (l_{next2}, A_2(\theta \cup v')); \)
             20. \( r'' = r'.(e(p), \theta).q_{next2}; \)
             21. \( RUNS = (RUNS \setminus r) \cup r''; \)
   22. return the trace set \( T = proj_{\Sigma_{Pr(M)}}(RUNS); \)

Algorithm 2 reflects exactly the monitoring of an implementation definition
(Definition 17). It collect valued events and constructs traces of \( ||p(Env, PM, I) \)
by supposing that both \( I \) and \( Env \) are ioLTS suspensions. Lines (7-21) implement
the rules of Definition 17. In particular, when a verdict location \( lv \) is reached (line
11 or 14), the Proxy-monitor has constructed a run, from its initial state which
belongs to \( RUNV(||p(Env, PM, I)) \) with \( V \) a verdict location set. From this run,
we obtain a trace of \( Traces_V(||p(Env, PM, I)) \).
So, with Proposition 6, we can state the correctness of the algorithm with:

**Proposition 7** The algorithm has reached a location verdict in:

- $\text{FAIL/VIOLATE} \Rightarrow \text{Traces}_{\text{FAIL/VIOLATE}}(\parallel p (\text{Env}, \text{PM}, I)) \neq \emptyset \Rightarrow \text{I} \not\models (\emptyset, \text{Violate}_0)$ and $\neg (\text{I} \ioco \emptyset),$
- $\text{FAIL} \Rightarrow \text{Traces}_{\text{FAIL}}(\parallel p (\text{Env, PM, I})) \neq \emptyset \Rightarrow \neg (\text{I} \ioco \emptyset),$
- $\text{VIOLATE} \Rightarrow \text{Traces}_{\text{VIOLATE}}(\parallel p (\text{Env, PM, I})) \neq \emptyset \Rightarrow \text{I} \not\models (\emptyset, \text{Violate}_0).$

Both the previous algorithms perform a synchronous analysis. Algorithm 1 receives a message, transfers it to Algorithm 2, which analyses Proxy-Monitor transitions and states before eventually forwarding the message to its recipient. 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 executes only its behaviour analysis. Nevertheless, the synchronous method is still particularly interesting if we intend to add recovery actions (and in a more general way runtime enforcements).

### 6 Related works

Several research works dealing with runtime verification or passive testing have been proposed recently in the literature. We briefly compare here some of them with the present work.

Different safety property formalisms can be found in the literature, e.g., temporal logics [ABG+05], automata or similar formalisms [FJN+11, CJMR07]. In most runtime verification approaches, violations of safety properties are detected by monitors whose functioning can be summarised by: maintain a checker state from system observations and produce a verdict [HR02, BGHS04, ABG+05, FJN+11].

An approach dedicated to runtime verification of component-based systems is proposed in [FJN+11]. Instead of considering one general model for describing the composition as in this paper, each composite component has its own ioLTS model. The specialised BIP framework is used to compose them later. The composition monitoring is performed with a classical runtime verification framework (monitor generation, trace extraction and analysis).

On the other hand, passive testing also aims to monitor systems, but offers slightly different features since this technique usually serves to detect defects continuously in the system under test in reference to its specification. Passive testing is often used to check whether a system under test conforms to its specification by means of a forward checking algorithm [MA00, LCH+06]: implementation reactions are given on the fly to an algorithm which detects incorrect behaviours.
by covering the specification transitions with these reactions. In this field, Lee and al. propose a passive testing method dedicated to wired protocols [LCH+06]. Forward checking algorithms may be improved with backward checking [ACC+04]: this approach performs two steps. Firstly, the specification is covered with a given trace backward. With this step, the algorithm tries to find the possible starting states of the trace, which lead to the current one. Then, it analyses the past of this set of starting states, also in a backward manner, seeking for states in which the variables are determined. When such states are reached, a decision is taken on the validity of the studied paths. Passive testing can be also employed to check invariant satisfiability [BCNZ05, ACNn11] where invariants, represent properties which are always true. This method is very similar to runtime verification.

Few works have focused on the combination of runtime verification with conformance testing [ABG+05, LS09, CJMR07]. The latter consider active testing and therefore a combination of properties with classical test cases which are later actively executed on the system. In [ABG+05], 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 expressed by ioco [CJMR07]. In the 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.

To collect traces, three main possibilities have been proposed. Monitors or passive tester can be encapsulated within the system [BGR07, CBML09], can be composed of probes, deployed in the system environment [DSS+05, PMCR08, FJN+11], or by adding probes directly into the system code [dH05]. These solutions bring several disadvantages such as risks of adding bugs in the implementation environment and/or require an open access to deploy tools. Guaranteeing this last hypothesis is more and more difficult for security or technical reasons. Our work mainly focuses on these issues, by proposing the use of Proxy-monitors. We also show that the resulting algorithms can be easily modified to propose either synchronous or asynchronous analysis.
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 for security or technical reasons. 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 this paper, we have dealt with deterministic ioSTS specifications, like many testing approaches proposed in the literature. A direct solution for considering nondeterministic ioSTSs is to apply determinization techniques [JMR06] on them. In a future work, we could also consider nondeterministic ioSTSs and a weaker test relation than ioco to generate nondeterministic Proxy-testers.

A Proof of Proposition 1

$I \text{ioco} \ S \iff \text{Traces}(\Delta(I)) \cap \text{refl}(\text{Traces}_{\text{Fail}_{\text{Can}}}(S)) = \emptyset$

Proof $I \text{ioco} \ S \iff \text{Traces}(\Delta(S)).(\Sigma^O \cup \{!\delta\}) \cap \text{Traces}(\Delta(I)) \subseteq \text{Traces}(\Delta(S))$ (Definition 10)

$I \text{ioco} \ S \iff (\text{Traces}(\Delta(S)).(\Sigma^O \cup \{!\delta\}) \setminus \text{Traces}(\Delta(S))) \cap \text{Traces}(\Delta(I)) = \emptyset$

$(\text{Traces}(\Delta(S)).(\Sigma^O \cup \{!\delta\}) \setminus \text{Traces}(\Delta(S)))$ corresponds to the trace set of $\text{refl}(\text{Can}(S))$ (Rules of Definition 11)

$(\text{Traces}(\Delta(S)).(\Sigma^O \cup \{!\delta\}) \setminus \text{Traces}(\Delta(S))) = \text{refl}(\text{Traces}_{\text{Fail}_{\text{Can}}}(S))$  

B Proof of Proposition 2

Let $S$ be an ioSTS. We have $Pr^{-1}(Pr(S)) = S$

We begin with the definition of $Pr^{-1}$.

Definition 18 Let $S$ be an ioSTS and $Pr(S)$ its Proxy-tester. $Pr^{-1}$ is the ioSTS operation such that $Pr^{-1}(Pr(S)) =_{\text{def}} S_2 = \langle L_{S_2}, l_{0_{Pr(S)}}, V_{Pr(S)} \setminus \{\text{side, pt}\}, V_0_{Pr(S)} \setminus \{\text{side := "", pt := ""}\}, I_{Pr(S)}, \Lambda_{S_2}, \rightarrow_{S_2} \rangle$ such that $L_{S_2}, \Lambda_{S_2}$ and $\rightarrow_{S_2}$ are constructed with the following rules:
Lemma 3 (Bijective operations \(Pr\) and \(Pr^{-1}\)) Let \(S\) be an ioSTS. \(Pr\) produces one ioSTS \(Pr(S)\) from \(S\) and \(Pr^{-1}(S)\) stems from one ioSTS \(S\).

\(Pr^{-1}\) produces one ioSTS \(S\) from a ioSTS \(Pr(S)\) and \(S\) stems from one ioSTS \(Pr(S)\).

Proof

IoSTS Hypothesis (1). We recall that ioSTSs are deterministic, so for a transition \(I_1 \xrightarrow{a(p),GA} I_j \in \rightarrow S\), it does not exist \(I_1 \xrightarrow{a(p),GA_2} I_j \in \rightarrow S\) such that \(A_2 \neq A\) (Definition 2).

\(Pr\) is an injective operation.

A transition \(I_1 \xrightarrow{a(p),GA} S I_2\), with \(I_2 \neq \text{Fail}\) is transformed, with \(Pr(S)\), into two transitions \(I_1 \xrightarrow{a(p),GA,\{p:=p,side:="Can"\}} I_2,\{a(p),G\} \xrightarrow{a(p),GA,\{p:=p,side:="Can"\}} Pr(S) I_2\) and only those (Rule 1, Definition 12). The location \((I_1, I_2, a(p), G)\) is unique for \(I_1 \xrightarrow{a(p),GA} S I_2\) according to Hypothesis (1). This is similar with a transition \(I_1 \xrightarrow{a(p),GA} S I_2\), with \(I_2 \neq \text{Fail}\) (Rule 1, Definition 12). A transition \(I_1 \xrightarrow{a(p),GA} S \text{Fail}\) is transformed into one.
transition \( l_1 \xrightarrow{a(p),GA} Pr(S) \) \( Fail \). By iteration on the transition set, we deduce that \( Pr \) is injective.

\( Pr \) is a surjective operation.

Let us consider that we have \( \mathcal{P}(S) \) and that it may results from two different ioSTS \( S_1 \) and \( S_2 \). If \( l_1 \xrightarrow{a(p),GA} Pr(S) \) \( Fail \) belongs to \( \rightarrow_{P(S)} \) then \( l_1 \xrightarrow{a(p),GA} Pr(S) \) \( Fail \) also belongs to \( \rightarrow_S \) and \( \rightarrow_{S_2} \).

Since \( S_1 \neq S_2 \), there exist (part 1) \( t1.t2 = l_1 \xrightarrow{?a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( (l_1, l_2, a(p), G) \xrightarrow{\mathfrak{I}(p),\{p = p_i\}, \{(x, x) \in \text{Can(S)} \text{ side := "Can"}, pt := pt\}} Pr(S) \) \( l_2 \) (Rule 1 Definition 12) or (part 2) \( t1.t2 = l_1 \xrightarrow{?a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( (l_1, l_2, a(p), G) \xrightarrow{\mathfrak{I}(p),\{p = p_i\}, \{(x, x) \in \text{Can(S)} \text{ side := "m", pt := pt\}} Pr(S) \) \( l_2 \) (Rule 2 Definition 12), such that

\( l_1' = b(p), Gb, Ab \xrightarrow{\mathcal{P}(S)} S_1 l_2' \) is transformed into \( t1.t2 \) and \( l_1'' = c(p), Gc, Ac \xrightarrow{\mathcal{P}(S)} S_2 l_2'' \) is transformed into \( t1.t2 \).

(part 1): the first rule of Definition 12 shows that \( l_1 = l_1' = l_2', l_2 = l_2'' = l_2'' = a(p) = b(p) = c(p), G = Gb = Gc \) and \( A = Ab = Ac \) (Hypothesis (1)). \( l_1 \xrightarrow{b(p), Gb, Ab} S_1 \)

\( l_2 = l_2'' = c(p), Gc, Ac \xrightarrow{S_2} S_2 l_2'' \). This contradicts the initial hypothesis.

(part 2): the reasoning is similar with the second rule of Definition 12.

Consequently \( S_1 = S_2 \) and \( Pr \) is surjective.

Now, we prove that we have \( Pr^{-1}(Pr(S) = S) \): Proof With the notations of Definitions 12 and 18, we have to prove that \( S_2 = Pr^{-1}(Pr(S) = S) \).

- \( L_{S_2} = L_S \).

\( (L_S \subseteq L_{S_2}) \).

Consider \( l_1 \xrightarrow{a(p),GA} S \) \( l_2 \), with \( l_2 \neq \text{Fail} \). The two first rules of Definitions 12 produce two unique transitions \( l_1 \xrightarrow{?a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( (l_1, l_2, a(p), G) \)

\( \xrightarrow{?a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \). \( \{l_1, l_2\} \subseteq L_{Pr(S)} \). With \( Pr^{-1} \), \( l_1 \xrightarrow{?a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( (l_1, l_2, a(p), G) \)

\( \xrightarrow{?a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \) are transformed into a unique transition \( l_1 \xrightarrow{a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \) with one of the two first rules of Definition 18. \( \{l_1, l_2\} \subseteq L_{Pr^{-1}(Pr(S))} \).

Now, Consider \( l_1 \xrightarrow{a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \) \( \xrightarrow{?a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \) \( \neq \text{Fail} \). \( l_1 \xrightarrow{a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \) \( \neq \text{Fail} \) \( \subseteq L_{Pr^{-1}(Pr(S))} \). By iteration on the transition set of \( S, L_S \subseteq L_{S_2} \).

\( (L_S \subseteq L_{S_2}) \).

Any transition \( l_1 \xrightarrow{a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \) with \( l_2 \neq \text{Fail} \) stems from the unique transitions \( l_1 \xrightarrow{a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_1 \xrightarrow{a(p),GA \cup \{p_i := p, side := "m", pt := pt\}} Pr(S) \) \( l_2 \) (Definition 18
and Lemma 3). \( \{l_1, l_2\} \subseteq L_{Pr(S)} \). \( l_1 \xrightarrow{a(p), G.A^2}_{P_{r^{-1}(S_2)}} (l_1, l_2, a(p), G) \)

\( a(p), G.A^3 \xrightarrow{pr^{-1}(S_2)} l_2 \) come from the unique transition \( l_1 \xrightarrow{a(p), G.A^4}_{Pr(pr^{-1}(S_2))} l_2 \) with \( Pr(Pr^{-1}(S_2)) \) ((Definition 12 and Lemma 3)). \( \{l_1, l_2\} \subseteq L_S \). If \( l_1 \xrightarrow{a(p), G.A} F ail \in \rightarrow S_2 \) then \( \{l_1, F ail\} \subseteq \rightarrow S \) (Definitions 12 and 18). By iteration, \( L_{S_2} \subseteq L_S \). Consequently, \( L_{S_2} = L_S \).

- \( l_0 S = l_0 S \)
The initial location of \( Pr(S) \) is \( l_0 S \) (Definition 12) and the initial location of \( S_2 \) is \( l_0 _{Pr(S)} \). Consequently \( l_0 S = l_0 S \).

- \( V_{S_2} = V_S \).
\( V_{Pr(S)} = V_S \cup \{side, pt\} \) (Definition 12) and \( V_{S_2} = V_{Pr(S)} \setminus \{side, pt\} \) (Definition 18). Consequently \( V_S = V_{S_2} \).

- \( V_{0 S_2} = V_{0 S} \).
Similar to \( V_{S_2} = V_S \).

- \( I_S = I_S \).
\( I_S = I_{Pr(S)} \) (Definition 12) and \( I_{S_2} = I_{Pr(S)} \) (Definition 18). Consequently, \( I_S = I_{S_2} \).

- \( \Lambda_{S_2} = \Lambda_S \).
\( \Lambda_S \subseteq \Lambda_{S_2} \)
Consider the action \( !a(p) \in \Lambda_S \) and any transition \( l_1 \xrightarrow{!a(p), G.A}_{S} l_2 \), \( l_2 \neq F ail \)
labelled by \( !a(p) \). \( Pr(S) \) transforms it into \( l_1 \xrightarrow{?a(p), G.A^2}_{Pr(S)} (l_1, l_2, a(p), G) \)
\( \xrightarrow{?a(p), G.A^3}_{Pr(S)} l_2 \) (Rule 1 of Definition 12). \( \{!a(p)\} \in \Lambda_{Pr(S)} \). With \( Pr^{-1}(Pr(S)) \), \( l_1 \xrightarrow{!a(p), G.A^2}_{Pr(S)} (l_1, l_2, a(p), G) \)
\( \xrightarrow{!a(p), G.A^3}_{Pr(S)} l_2 \) gives \( l_1 \xrightarrow{!a(p), G.A^4}_{S_{pr^{-1}(S_2)}} l_2 \) (Rule 1 of Definition 18). \( \{!a(p)\} \in \Lambda_{S_2} \). The reasoning is similar with an action \(?a(p) \in \Lambda_S \) except that the second rules of Definition 12 and Definition 18 has to be considered. For both input and output actions, carried by a transition \( l_1 \xrightarrow{!a(p), G.A}_{S} F ail \), \( l_1 \xrightarrow{!a(p), G.A}_{F ail} F ail \in \rightarrow S_{pr^{-1}(S_2)} \)
and \( l_1 \xrightarrow{!a(p), G.A}_{F ail} F ail \in \rightarrow S_2 \) (Definitions 12 and 18). Consequently, \( \Lambda_S \subseteq \Lambda_{S_2} \).

\( \Lambda_{S_2} \subseteq \Lambda_S \).
Consider the action \( !a(p) \in \Lambda_S \) and any transition \( l_1 \xrightarrow{!a(p), G.A}_{S} l_2 \), \( l_2 \neq F ail \)
labelled by $!a(p)$, $l_1 \xrightarrow{!a(p),G_A} S_2 l_2$ stems from the transitions $l_1$

\[ \xrightarrow{!a(p),G_A \theta} l_1, l_2, a(p), G \xrightarrow{!a(p),G_A \theta} l_2 \] (Rule 1 of Definition 12 and Lemma 3). \{!a(p)\} $\in \Lambda_{Pr(S)}$. $l_1 \xrightarrow{!a(p),G_A \theta} S_2 l_2$ can stem from the transition $l_1 \xrightarrow{!a(p),G_A \theta} S_2 l_2$. (Rule 1 of Definition 18 and Lemma 3). \{!a(p)\} $\in \Lambda_S$. As previously, for both input and output actions, carried by a transition $l_1 \xrightarrow{!a(p),G_A \theta} S_2 l_2$, $l_1 \xrightarrow{!a(p),G_A \theta} \text{Fail}$, $l_1 \xrightarrow{!a(p),G_A \theta} \text{Fail} \in \text{rightarrow}_{Pr(S)}$ and $l_1 \xrightarrow{!a(p),G_A \theta} \text{Fail} \in \text{rightarrow}_{S_2}$ (Definitions 12 and 18). Consequently, $\Lambda_{S_2} \subseteq \Lambda_S$ and $\Lambda_{S_2} = \Lambda_S$.

\[ \rightarrow_{S_2} \subseteq \rightarrow_{S} \]

We shall prove this equality by considering the underlying semantics level of ioSTSs and by proving that $\rightarrow_{||S_2||} = \rightarrow_{||S||}$.

Firstly, we prove that if a transition $t$ belongs to $\rightarrow_{||S||}$ then it belongs to $\rightarrow_{||S_2||}$ with the following lemma:

**Lemma 4** If $(l_1, v_1) \xrightarrow{a(p), \theta} (l_2, v_2) \in \rightarrow_{||S||}$ then $(l_1, v_1) \xrightarrow{a(p), \theta} (l_2, v_2) \in \rightarrow_{||S_2||}$.

**Proof** $(l_1, v_1) \xrightarrow{a(p), \theta} (l_2, v_2)$ stems from the ioSTS transition $l_1 \xrightarrow{a(p),G_A \theta} S_2 l_2$ such that $v_1 \cup \theta = G$ and $v_2 = A(v_1 \cup \theta)$ (Definition 3). If $a(p)$ is an output action $!a(p)$ and $l_2 \neq \text{Fail}$, the ioSTS transition is transformed into the unique sequence $l_1 \xrightarrow{!a(p),G_A \cup \{ p_t := p, \text{side} := "Can", pt := pt \}} \rightarrow_{Pr(S)} (l_1, l_2, a(p), G)$

\[ \xrightarrow{!a(p),G_A \cup \{ p_t := p, \text{side} := "Can", pt := pt \}} \rightarrow_{Pr(S)} (l_1, l_2, a(p), G) \]

$(l_1, v_1 \cup \{ pt := p, \text{side} := "Can", \text{side} := "" \}) \xrightarrow{!a(p), \theta} S_2 l_2$. Then, the two ioSTS transitions $l_1 \xrightarrow{!a(p),G_A \cup \{ p_t := p, \text{side} := "Can", pt := pt \}} S_2 l_2$ are also transformed with $Pr^{-1}$ into the unique transition $l_1 \xrightarrow{!a(p),G_A \theta} S_2 l_2$ by removing in $V_{Pr(S)}$ the variables $pt$ and $\text{side}$ (Rule 2, Definition 18). By applying this transformation on $(l_1, v_1 \cup \{ pt := p, \text{side} := "Can", \text{side} := "" \}) \xrightarrow{!a(p), \theta} S_2 l_2$. By applying this transformation on $(l_1, v_1 \cup \{ pt := p, \text{side} := "Can", \text{side} := "" \}) \xrightarrow{!a(p), \theta} S_2 l_2$. By applying this transformation on $(l_1, v_1 \cup \{ pt := p, \text{side} := "Can", \text{side} := "" \}) \xrightarrow{!a(p), \theta} S_2 l_2$. By applying this transformation on $(l_1, v_1 \cup \{ pt := p, \text{side} := "Can", \text{side} := "" \}) \xrightarrow{!a(p), \theta} S_2 l_2$.
"Can’\})

\[ a(p, \theta) \xrightarrow{||S||} (l_2, v_2) \]

The same reasoning can be applied if \( a(p) \) is an input action and \( a(p) \neq \text{Fail} \) with the second rules of Definitions 12 and 18. If \( l_2 = \text{Fail} \), the \( \text{ioSTS} \) transition is unchanged with \( Pr \) and \( Pr^{-1} \), so

\[ (l_1, v_1) \xrightarrow{a(p, \theta)} (\text{Fail}, v_2) \]

belongs to both \( \rightarrow ||S|| \) and \( \rightarrow ||S2|| \).

Now, we prove the following lemma:

\[ \text{Lemma 5 If } (l_0, v_0) \xrightarrow{a(p_1, \theta_1)} (l_1, v_1) \ldots (l_{n-1}, v_{n-1}) \xrightarrow{a(n_{p_n}, \theta_n)} (l_n, v_n) \in (\rightarrow ||S||)^n \]

We have previously shown that \( l_0 = l_0 ||S2 \) and \( V_0 = V_0 ||S2 \). Consequently, \( ||S|| \) and \( ||S2|| \) have the same initial state \((l_0 ||S2 \), \( V_0 ||S2 \) \) (1).

Hypothesis (a): Let \( \sigma' = (a_1(p_1), \theta_1) \ldots (a_{n-1}(p_{n-1}), \theta_{n-1}) \). We assume that if \( \text{seq'} = (l_0, v_0) \xrightarrow{a_1(p_1, \theta_1)} (l_1, v_1) \ldots (l_{n-2}, v_{n-2}) \xrightarrow{a_{n-1}(p_{n-1}, \theta_{n-1})} (l_{n-1}, v_{n-1}) \in (\rightarrow ||S||)^{n-1} \) then \( \text{seq'} \in (\rightarrow ||S2||)^{n-1} \).

We consider \( \sigma = \sigma' (a_n(p_n), \theta_n) \). If \( \text{seq} = (l_0, v_0) \xrightarrow{a_1(p_1, \theta_1)} (l_1, v_1) \ldots (l_{n-1}, v_{n-1}) \xrightarrow{a_n(p_n), \theta_n} (l_n, v_n) \in (\rightarrow ||S||)^n \) then \( \text{seq} \in (\rightarrow ||S2||)^n \).

With (1) and Hypothesis (a), we have for \( \sigma' \), \( \text{seq'} \in (\rightarrow ||S||)^{n-1} \) and \( \text{seq'} \in (\rightarrow ||S2||)^{n-1} \). Next, the statement "if \( (l_{n-1}, v_{n-1}) \xrightarrow{a_n(p_n), \theta_n} (l_n, v_n) \in (\rightarrow ||S||)^n \) then \( (l_{n-1}, v_{n-1}) \xrightarrow{a_n(p_n), \theta_n} (l_n, v_n) \in (\rightarrow ||S2||)^n \) holds with Lemma 4.

If the same reasoning on Hypothesis (1) is applied iteratively with Lemma 4, Lemma 5 holds.

With Lemma 5, we deduce that \( \rightarrow ||S|| \subseteq \rightarrow ||S2|| \).

\[ \rightarrow ||S2|| \subseteq \rightarrow ||S|| \]

is proved by considering the same reasoning by switching \( ||S|| \) and \( ||S2|| \).

As a consequence, Proposition 2 holds.

\[ \text{C Proof of Proposition 3} \]

\[ \text{Proposition 8 Let } S \text{ be an ioSTS. We have } \text{Traces}^{\text{Can}}(Pr(\text{Can}(S))) = \text{Traces}(Pr^{-1}(Pr(\text{Can}(S)))) = \text{Traces}(\text{Can}(S)). \]

We have to prove that \( \text{Traces}^{\text{Can}}(Pr(\text{Can}(S))) = \)
Traces(Can(\$)) since Pr^{-1}(Pr(Can(\$))) = Can(\$) (Proposition 2).

We shall prove this equality by considering the underlying semantics level of ioSTSs because traces are derived from the projection of runs on actions and runs stem from ioLTS sequences.

The Pr operator is defined at the ioSTS syntax level whereas traces are given from the semantics level of ioSTSs. Firstly, we study the relation that we have between an ioSTS semantics transition of \$|\$ and \$|\$ is transformed into the unique sequence seq = l_1.

Lemma 6 Let t = l_1 a(p),G \rightarrow l_2 be a transition of \$\rightarrow Can(\$). And let seq be the resulting transition sequence obtained from t with Definition 12. For any valued action \$a(p),\$ in \$\Sigma|\$Can(\$)| \subset \Sigma|\$Pr(Can(\$))|, we have:

1. \$a(p) \in \Lambda_{\text{Can(\$)}}^O \{l_1, v_1 \cup \{\text{side := A1, pt := B1}\} \}$

2. \$a(p) \in \Lambda_{\text{Can(\$)}}^I \{l_1, v_1 \cup \{\text{side := A1, pt := B1}\} \}$

3. \$a(p) \in \Lambda_{\text{Can(\$)}}^I \{l_1, v_1 \cup \{\text{side := A1, pt := B1}\} \}$

Proof

1. \$l_1 \xrightarrow{a(p),G} l_2 \in \rightarrow Can(\$)$ with \$l_2 \neq \text{Fail}$ is the ioLTS semantics transition of a ioSTS transition \$l_1 \xrightarrow{\#a(p),G} l_2 \in \rightarrow Can(\$) \text{ iff } v_1 \cup \theta \models G$ and \$v_2 = A(v_1 \cup \theta)$. 

\$l_1 \xrightarrow{a(p),G} l_2 \text{ is transformed into the unique sequence } seq = l_1 \xrightarrow{\#a(p),G} l_2$ (Rule 1, Definition 12). The ioLTS semantics transitions of \$seq$ are \$l_1, l'_1 \xrightarrow{\#a(p),\theta}$.
From 1), 2) and 3), we deduce directly that Lemma 6 holds.

We have $V_{Pr(\text{Can}(S))} = V_{\text{Can}(S)} \cup \{\text{side}, pt\}$.

$v'_1 \cup \theta \models G$ iff $v'_1 = v_1 \cup \{\text{side} := A1, pt := B1\}$ with any $A1 \in D_{Pr(\text{Can}(S))}$, $B1 \in D_{Pr(\text{Can}(S))}$ and $v_1$ satisfies $G$.

$v'_2$ is obtained with $A \cup \{pt := p, side := ""\}(v'_1)$, so $v'_2 = A(v_1 \cup \theta) \cup \{pt := p, side := ""\}$. We also have $v'_2 \models [p := pt]$.

$v'_3$ is obtained in $V_{Pr(\text{Can}(S))}$ with $v'_3 = \{\text{side} := "Can", pt := B1\}(v_2 \cup \{\text{side} := "", pt := \theta\})$. $v'_3 = v_2 \cup \{\text{side} := "Can", pt := B1\}$.

2. same reasoning as 1) by considering Rule 2 in Definition 12,

3. $(l_1, v_1) \xrightarrow{a(p), \theta} (l_2, v_2) \in \rightarrow_{||\text{Can}(S)||}$ with $l_2 = \text{Fail}$ is the iOLTS semantics transition of a STS transition $l_1 \xrightarrow{a(p), G, A} \text{Fail} \in \rightarrow_{\text{Can}(S)}$ iff $v_1 \cup \theta \models G$ and $v_2 = A(v_1 \cup \theta)$. $l_1 \xrightarrow{a(p), G, A} \text{Fail}$ is transformed into the unique transition $t = l_1 \xrightarrow{a(p), G, A \cup \{\text{side} := "Can"\}} \text{Fail} \in \rightarrow_{\text{Can}(S)}$ (Rule 3 Definition 12).

The iOLTS semantics transition of $t$ is $(l_1, v_1) \xrightarrow{a(p), \theta} (\text{Fail}, v'_2)$.

We have $V_{Pr(\text{Can}(S))} = V_{\text{Can}(S)} \cup \{\text{side}, pt\}$.

$v'_1 \cup \theta \models G$ iff $v'_1 = v_1 \cup \{\text{side} := A1, pt := B1\}$ with any $A1 \in D_{Pr(\text{Can}(S))}$, $B1 \in D_{Pr(\text{Can}(S))}$ and $v_1 \models G$.

$v'_2$ is obtained in $V_{Pr(\text{Can}(S))}$ iff $v'_2 = v_2 \cup \{\text{side} := "Can", pt := B1\} = A \cup \{\text{side} := "Can"\}(v_1 \cup \{\text{side} := A1, pt := B1\})$ and $v_2 = A(v_1)$.

From 1), 2) and 3), we deduce directly that Lemma 6 holds.

Next, we prove the following lemma which gives a relation on valued actions between $||\text{Can}(S)||$ and $||\text{Pr}(\text{Can}(S))||$.

**Lemma 7** For any $\sigma = \alpha_1 \ldots \alpha_n \in \Sigma^n_{||\text{Can}(S)||} \subseteq \Sigma^n_{||\text{Pr}(\text{Can}(S))||}$, we denote that $||\text{Pr}(\text{Can}(S))||$ has the $\sigma_{\text{Can}}$ behaviour when $||\text{Pr}(\text{Can}(S))||$ has the sequence seq composed by:

1. $(l_{i-1}, v_{i-1} \cup \{\text{side} := A_{i-1}, pt := B_{i-1}\}) \xrightarrow{\alpha_i} ((l_{i-1}, l_i, a_i(p), G_i), v_i \cup \{\text{side} := "", pt := B_i\})$ with $0 < i \leq n$, $l_i \neq \text{Fail}$ and $\alpha_i \in \Sigma^n_{||\text{Can}(S)||}$.

2. $(l_{i-1}, v_{i-1} \cup \{\text{side} := A_{i-1}, pt := B_{i-1}\}) \xrightarrow{\alpha_i} ((l_{i-1}, l_i, a_i(p), G_i), v_i \cup \{\text{side} := "Can", pt := B_i\})$ with $0 < i \leq n$, $l_i \neq \text{Fail}$ and $\alpha_i \in \Sigma^n_{||\text{Can}(S)||}$.

3. $(l_{i-1}, v_{i-1} \cup \{\text{side} := A_{i-1}, pt := B_{i-1}\}) \xrightarrow{\alpha_i} (l_i, v_i \cup \{\text{side} := "Can", pt := B_i\})$ with $0 < i \leq n$, $l_i = \text{Fail}$ and $\alpha_i \in \Sigma^n_{||\text{Can}(S)||}$.
iff \(||Can(S)||\) has the sequence \((l_0, v_0) \xrightarrow{\alpha_1} (l_1, v_1) \cdots (l_{n-1}, v_{n-1}) \xrightarrow{\alpha_n} (l_n, v_n)\)

**Proof** (1) It results from Definition 12 that \((l_0, V_0 Can(S))\) is the initial state of \(||Can(S)||\) and that \((l_0, V_0 Can(S)) \cup \{side := "", pt := ""\}\) is the initial state of \(||Pr(\text{Can}(S))||\).

Hypothesis (a): for, \(\sigma' = \alpha_1 \cdots \alpha_{n-1} \in \Sigma^{-1}_{||Can(S)||} \subseteq \Sigma^{-1}_{||Pr(\text{Can}(S))||}\); we suppose that \(||Pr(\text{Can}(S))||\) has the \(\sigma'_\text{Can}\) behaviour iff \(||Can(S)||\) has the sequence \((l_0, v_0) \xrightarrow{\alpha_1} (l_1, v_1) \cdots (l_{n-2}, v_{n-2}) \xrightarrow{\alpha_{n-1}} (l_{n-1}, v_{n-1})\).

Consider \(\sigma = \sigma' \alpha_n \in \Sigma^n_{||Can(S)||}\). For \(\alpha_n = (a_n(p_n), \theta_n)\), we have either (Lemma 6):

1. \(a(p) \in \Lambda^O_{Can(S)}, (l_{n-1}, v_{n-1}) \cup \{side := A_1, pt := B_1\} \xrightarrow{\theta_0} ((l_{n-1}, l_n, a_n(p_n), G_n)v_n \cup \{side := "", pt := \emptyset\}) \xrightarrow{\theta_n} (l_n, v_n \cup \{side := "Can", pt := \emptyset\}) \in (\rightarrow||Pr(\text{Can}(S))||)^2\), with \(A_1 \in D_{Pr(\text{Can}(S))}, B_1 \in D_{Pr(\text{Can}(S))}\) iff \((l_{n-1}, v_{n-1}) \xrightarrow{\alpha_n} (l_n, v_n) \in \rightarrow||Can(S)||\) and \(l_n \neq \text{Fail}\),

2. \(a(p) \in \Lambda^L_{Can(S)}, (l_{n-1}, v_{n-1}) \cup \{side := A_1, pt := B_1\} \xrightarrow{\theta_0} ((l_{n-1}, l_n, a_n(p_n), G_n)v_n \cup \{side := "Can", pt := \emptyset\}) \xrightarrow{\theta_n} (l_n, v_n \cup \{side := "", pt := \emptyset\}) \in (\rightarrow||Pr(\text{Can}(S))||)^2\), with \(A_1 \in D_{Pr(\text{Can}(S))}, B_1 \in D_{Pr(\text{Can}(S))}\) iff \((l_{n-1}, v_{n-1}) \xrightarrow{\alpha_n} (l_n, v_n) \in \rightarrow||Can(S)||\) and \(l_n \neq \text{Fail}\),

3. \(a(p) \in \Lambda^C_{Can(S)}, (l_{n-1}, v_{n-1}) \cup \{side := A_1, pt := B_1\} \xrightarrow{\theta_0} (\text{Fail}, v_n \cup \{side := "Can", pt := B_1\}) \in \rightarrow||Pr(\text{Can}(S))||\), with \(A_1 \in D_{Pr(\text{Can}(S))}, B_1 \in D_{Pr(\text{Can}(S))}\) iff \((l_{n-1}, v_{n-1}) \xrightarrow{\alpha_n} (l_n, v_n) \in \rightarrow||Can(S)||\) with \(l_n = \text{Fail}\).

Therefore, \(||Pr(\text{Can}(S))||\) has the \(\sigma_\text{Can}\) behaviour iff \(||Can(S)||\) has the sequence \((l_0, v_0) \xrightarrow{\alpha_1} (l_1, v_1) \cdots (l_{n-1}, v_{n-1}) \xrightarrow{\alpha_n} (l_n, v_n)\).

By applying iteratively Hypothesis (a) with (1), we obtain that \(||Pr(\text{Can}(S))||\) has the \(\sigma_\text{Can}\) behaviour iff \(||Can(S)||\) has the sequence \((l_0, v_0) \xrightarrow{\alpha_1} (l_1, v_1) \cdots (l_{n-1}, v_{n-1}) \xrightarrow{\alpha_n} (l_n, v_n)\).

From Lemma 7, we have for \(\sigma = \alpha_1 \cdots \alpha_n \in \Sigma^n_{||Can(S)||}, ||Pr(\text{Can}(S))||\) has the \(\sigma_\text{Can}\) behaviour iff \(||Can(S)||\) has the sequence \(\text{seq} = (l_0, v_0) \xrightarrow{\alpha_1} (l_1, v_1) \cdots (l_{n-1}, v_{n-1}) \xrightarrow{\alpha_n} (l_n, v_n)\) and \(\text{Traces}(\text{seq}) = \sigma\).

\(\text{Traces}_{\text{Can}}(\sigma_{\text{Can}})\) is the projection \(\text{proj}_{\text{Pr}(\sum_{\text{Side}} \text{Can}(Q))(\text{seq}') = \alpha'_1 \cdots \alpha'_n\text{ (Definition 13)}\) such that \(\text{seq}'\) is composed of (Lemma 7):

\[-\text{seq}_i = (l_{i-1}, v_{i-1}) \cup \{side := A_{i-1}, pt := B_{i-1}\} \xrightarrow{\theta_0} ((l_{i-1}, l_i, a_i(p), G_i), v_i \cup \{side := "", pt := B_i\}) \xrightarrow{\theta_n} (l_i, v_i \cup \{side := "Can", pt := B_i\})\text{ with }0 < i \leq n, l_i \neq \text{Fail}\text{ and }\alpha_i \in \Sigma^O_{||Can(S)||}, \text{ or}\]
\[ \text{Proposition } 9 \text{ Consider an external environment } Env, \text{ an implementation } I \text{ monitored with a Proxy-monitor } Pr(M), \text{ itself derived from an ioSTS } S \text{ and an Observer } \emptyset \in \text{Comp}(\Delta(S)). \text{ Let } \Sigma \subseteq \text{Traces}(\parallel P(Env, PM, I)) \text{ be the observed trace set. If it exists } \sigma \in \Sigma \text{ such that:} \\
\begin{enumerate}
\item \sigma \text{ belongs to } Traces_{FAIL/VIOSTATE}(\parallel P(Env, PM, I)), \text{ then } I \text{ does not satisfy the safety property and } I \text{ is not ioco-conforming to } S, \\
\item \sigma \text{ belongs to } Traces_{FAIL}(\parallel P(Env, PM, I)), \text{ then } I \text{ is not ioco-conforming to } S. \text{ No violation of the safety property were detected on } I, \\
\item \sigma \text{ belongs to } Traces_{VIOSTATE}(\parallel P(Env, PM, I)), \text{ then } I \text{ does not satisfy the safety property. Non-conformance between } I \text{ and } S \text{ were not detected.}
\end{enumerate}
\]

**Proof**

Proof of 1):

\[ \sigma \text{ belongs to } Traces_{FAIL/VIOSTATE}(\parallel P(Env, PM, I)). \text{ Therefore,} \]

\[ Traces_{FAIL/VIOSTATE}(\parallel P(Env, PM, I)) \neq \emptyset \text{ and } Traces_{FAIL/VIOSTATE}(\parallel P(Env, PM, I)) \neq \emptyset. \]

\[ Traces_{FAIL/VIOSTATE}(\parallel P(Env, PM, I)) = refl(Traces(\Delta(I))) \cap Traces_{FAIL/VIOSTATE}(Pr(M)) \]

(Proposition 5)

\[ Traces_{FAIL/VIOSTATE}(Pr(M)) = Traces_{FAIL/VIOSTATE}(M) \] (Proposition 3) and by considering that \( M \) is also a Canonical tester which is specialised to also recognise property violations. It is a Canonical tester in the sense that it has a mirrored action set and can communicate with the specification (and the implementation). It also has a Fail location set. As a consequence, (Proposition 3) also holds with \( M \).

It follows that

\[ Traces_{FAIL/VIOSTATE}(\parallel P(Env, PM, I)) = refl(Traces(\Delta(I))) \cap (Traces_{FAIL}(Can(S)) \cap Traces_{VIOSTATE}(refl(\emptyset))) \neq \emptyset \] (Lemma 2).
We deduce that $\text{Traces}^{\text{Can/Violate}}(\parallel_\rho(\text{Env}, \text{PM}, I)) \neq \emptyset$ iff $\text{refl}(\text{Traces}(\Delta(I))) \cap \text{Traces}^{\text{Fail}}(\text{Can}(S)) \neq \emptyset(a)$ and iff $\text{refl}(\text{Traces}(\Delta(I))) \cap \text{refl}(\text{Traces}^{\text{Violate}_\mu}(O)) \neq \emptyset(b)$. From (a) and Proposition 1, we have $\neg I \text{ioco} S$.

From (b) and Definition 9, we have $I \not\models O$.

Consequently, $I$ does not satisfy the safety property and $I$ is not ioco-conforming to $S$.

Proofs of 2) and 3) can be deduced by considering the same reasoning as 1).

References


