Proxy-Monitor: An integration of runtime verification with passive conformance testing.

Sébastien Salva, Tien-Dung Cao

Title: Proxy-Monitor: An integration of runtime verification with passive conformance testing.

Authors

Sébastien Salva,
Corresponding author
LIMOS CNRS UMR 6158, PRES Clermont-Ferrand University, France
Email: sebastien.salva@udamail.fr

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

Proxy-Monitor: An integration of runtime verification with passive conformance testing.

ABSTRACT

This paper proposes a conformance testing method combining two well-known testing approaches, runtime verification and passive testing. Runtime verification addresses the monitoring of a system under test to check whether formal properties hold, while passive testing aims at checking the conformance of the system in the long-term. The method, proposed in this paper, checks whether an implementation conforms to its specification with reference to the *ioco* test relation. While passively checking if ioco holds, it also checks whether the implementation meets safety properties, which informally state that "nothing bad ever happens". This paper also tackles the trace extraction problem, which is common to both runtime verification and passive testing. We define the notion of Proxy-monitors for collecting traces even when the implementation environment access rights are restricted. Then, we apply and specialise this approach on Web service compositions. A Web service composition deployed in different Clouds is experimented to assess the feasibility of the method.

Keywords: Conformance testing, Passive Testing, Runtime Verification, Proxy-Tester, ioco, Monitoring, Service Composition, Clouds.

INTRODUCTION

Model-based Testing is a Software testing approach that is gaining ground in the Industry as an automatic solution to find defects in black-box implementations. Usually, deciding whether an implementation conforms to its specification comes down to checking whether a test relation holds. Such a relation defines the notion of correctness without ambiguity by expressing a comparison of observable functional behaviours. But beyond the use of formal techniques, model-based testing offers the advantage to automate some (and eventually all) steps of the testing process. Generally, the mainstream of testing is constituted by active methods: basically, test cases are constructed from the specification and are experimented on its implementation to check whether the latter meets desirable behaviours. Active testing may give rise to some inconvenient though, e.g., the repeated or abnormal disturbing the implementation.

Passive testing and runtime verification are two complementary approaches, employed to monitor implementations over a longer period of time without disturbing them. 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 (Miller & Arisha, 2000; (Alcalde, Cavalli, Chen, Khuu, & Lee, 2004; Lee, Chen, Hao, Miller, Wu, & Yin, 2006). 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 (Leucker &

Schallhart, 2009).

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 method, which combines the two previous approaches. Our main contributions can be summarized 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 are known to be monitorable and can be used to express a very large set of properties. We combine this monitoring approach with a previous work dealing with ioco passive testing (IdentificationRemoved, 2012). Ioco (Tretmans, 1996) is a well-known conformance test relation that defines the conforming implementations by means of suspension traces (sequences of actions and quiescence). Starting from an ioSTS (input output Symbolic Transition System) model, our method generates monitors for checking whether an implementation ioco-conforms to its specification and meets safety properties,
2. trace extraction: the trace recovery requires an open testing environment where tools, workflow engines or frameworks can be installed. Nonetheless, the real implementation environment access is more and more frequently restricted. For instance, Web server accesses are often strictly limited for security reasons. And these restrictions prevent from installing monitors collecting traces. Another example concerns Clouds. Clouds, and typically PaaS (Platform as a service) layers are virtualized environments where Web services and applications are deployed. These virtualizations of resource, whose locations and details are not known, combined with access restrictions, make difficult the trace extraction. We face this issue by using the notion of transparent proxy and by assuming that the implementation can be configured to pass through a proxy (usually the case for Web applications). But, instead of using a classical proxy to collect traces, we propose generating a formal model from a specification, called Proxy-monitor, which reflects a proxy functioning combined with the automatic detection of 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. We compare these two modes and give some experimental measurements.

The paper is structured as follows: we review some related work on passive testing and runtime verification in the next Section. We provide some notations to be used throughout the paper in Section Model definition and Notations. Then, we recall the principles of runtime verification and ioco passive testing. The combination of both which leads to the Proxy-monitor model is defined in Section Combining runtime verification and Proxy-tester. Afterwards, we apply the concept of Proxy-monitor on Web service compositions. Conclusions with directions for further research are drawn in Section Conclusion.

RELATED WORK

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

In most of the runtime verification approaches, violations of safety properties are detected by monitors whose functioning can be summarized by: maintain a checker state from system

observations and produce a verdict (Havelund & Rosu, 2002; Barringer, Goldberg, Havelund & Sen, 2004; Falcone, Jaber, Nguyen, Bozga, & Bensalem, 2011). Safety properties can be modelled with several formalisms, e.g., temporal logics (Arthoa, Barringerb, Goldbergc, Havelundc, Khurshidd, Lowrye, Pasareanuf, Rosug, Seng, Visserh, & Washingtonh, 2005), automata or similar formalisms (Falcone, Jaber, Nguyen, Bozga, & Bensalem, 2011; Constant, Jeron, Marchand & Rusu, 2007). (Falcone, Jaber, Nguyen, Bozga, & Bensalem, 2011) proposed an approach dedicated to runtime verification of component-based systems. Instead of considering one general model for describing the composition as in this paper, each composite component has its own ioLTS model. The specialized 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. Passive testing is often used to check whether a system under test conforms to its specification by means of a forward checking algorithm (Miller & Arisha, 2000; Lee, Chen, Hao, Miller, Wu, & Yin, 2006). 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 et al. propose a passive testing method dedicated to wire protocols (Lee, Chen, Hao, Miller, Wu, & Yin, 2006). In (Lalanne, Che & Maag, 2011; Che, Lalanne & Maag, 2012), a data-centric approach is here proposed to test the conformance of protocols by defining a message as a collection of data fields and a logic syntax and semantics based in Horn logic in order to express properties. Forward checking algorithms may be improved with backward checking (Alcalde, Cavalli, Chen, Khuu, & Lee, 2004). With this approach, the specification is covered with a given trace in a backward manner to seek the starting states in which the variables can be 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 (Bayse, Cavalli, Nunez & Zaidi, 2005; Andres, Cambronero & Nunez, 2011) where invariants represent properties that are always true. This method is very similar to runtime verification.

Few works have focused on the combination of runtime verification with conformance testing. (Arthoa, Barringerb, Goldbergc, Havelundc, Khurshidd, Lowrye, Pasareanuf, Rosug, Seng, Visserh, & Washingtonh, 2005; Leucker & Schallhart 2009) consider active testing and therefore a combination of properties with classical test cases that are later actively executed on the system. 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 (Constant, Jeron, Marchand & Rusu, 2007). Here, the combination of active testing with runtime verification helps 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 is based upon passive testing and 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 also been proposed. Monitors can be encapsulated within the system (Barringer, Gabbay & Rydeheard, 2007; Cavalli, Benameur, Mallouli & Li, 2009), can be composed of probes deployed in the system environment (D'Angelo, Sankaranarayanan, Sanchez, Robinson, Finkbeiner, Sipma, Mehrotra & Manna,

2005; Pellizzoni, Meredith, Caccamo & Rosu, 2008; Falcone, Jaber, Nguyen, Bozga, & Bensalem, 2011), or composed of probes directly injected into the code (d'Amorim & Havelund, 2005). 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.

MODEL DEFINITION AND NOTATIONS

In this paper, we focus on models called input/output Symbolic Transition Systems (ioSTS). An ioSTS is a kind of automata model, which is extended with two sets of variables, 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 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 ioco theory (Tretmans, 1996).

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. Outputs of the environment are never rejected. ioSTSs gather two set of variables, internal and interaction variables. This distinction is convenient to clearly express the state of the system (internal variables) and to model complex actions composed with communication parameters.

Below, we recall the definition of an extension, called ioSTS suspension 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, unauthorized deadlocks in the system under test. Usually, quiescence is observed on implementations with timers: after each event, a timer is reset. If it expires, then quiescence is observed. Timers are assumed initialized with a duration sufficiently long to ensure that any output action, provided by the implementation, can be observed.

Quiescence is modelled by a new symbol $!d$ and an augmented ioSTS denoted $\mathrm{D}(ioSTS)$. For an ioSTS S, $\mathrm{D}(S)$ is obtained by adding a self-loop labelled by $!d$ for each location where quiescence may be observed.

Definition 1 (ioSTS suspension) A deterministic Input Output Symbolic Transition System (ioSTS) suspension $\mathrm{D}(S)$ is a tuple $\langle L, l_0, V, V_0, I, \mathrm{L} \text{ È } \{!d\}, \text{®}_{\mathrm{D}(S)} \rangle$, 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 assignment of values of a set of variables Y Í X is denoted by valuations where a valuation is a function $v : Y$ ® D. We denote D_Y the set of valuations over the set of variables Y. The internal variables are initialized with the valuation V_0 Î D_V, which is assumed to be unique,
- L is the finite set of symbolic actions $a(p)$, with $p = (p_1, ..., p_k)$ a unique finite set of

parameter variables in $I^k (k \hat{I} ¥)$: if $a(p) \hat{I} L$, then $a(p')$, with $p' ¹ p$ does not belong to L. L is partitioned by $L = L^I È L^O$ $L^I(L^O)$ represents the set of input actions beginning with ? (the set of output actions beginning with ! respectively),

- ® is the finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \hat{I} L$ to $l_j \hat{I} L$, also denoted l_i ¾ $a(p)$,G,A ® l_j is labelled by an (input or output) action $a(p) \hat{I} L$, $G \subseteq D_p \times D_V \times D_{T(p \cup V)}$ is a guard on internal variables, parameters and $T(p \cup V)$ a set of functions that return Boolean values only (a.k.a. predicates) over $p \cup V$. Internal variables are updated with a set A of assignments of the form $(v := Av)_{v \hat{I} V}$ such that for each variable v, Av is an expression on $p È V È T(p È V)$,
- for any location $l \in L$ and for all pair of transitions $(l, l_1, a(p), G_1, A_1)$, $(l, l_2, a(p), G_2, A_2)$ labelled by the same action, $G_1 \wedge G_2$ is unsatisfiable.

These notations are expressed in the straightforward example of Figure 1 and Table 1. This specification, taken from the BPEL 2.0 specification (Jordan & Evdemon 2007), describes the functioning of a *Loan approval* service receiving as input *loan* requests composed of personal information and the amount being requested. If the amount is less than or equal to $10,000, a *Risk-assessment* service is called to return a risk level. The loan request is then approved when the risk level is estimated as low. For larger amounts or when the risk level is medium or high, the request requires the call of the *Approver* service which yields the final decision. In the remainder of the paper, we use the label "*" as a shortcut notation gathering all the valued actions that are not explicitly carried by other transitions.

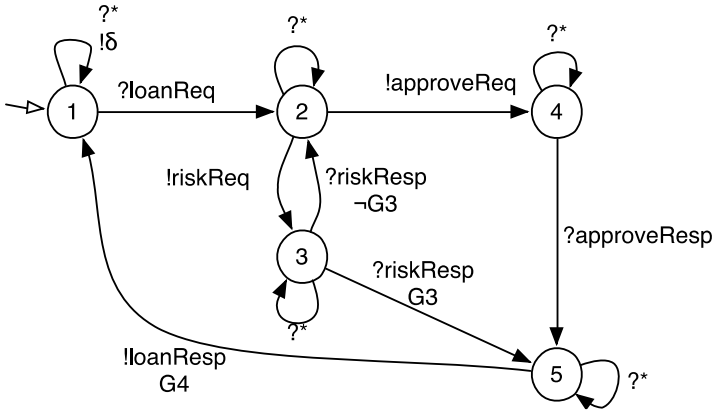


Figure 1: An ioSTS suspension

Symbol	Message	Guard	Update
?loanReq	?loanRequest(profile, amount)		a:=amount p:=profile
!riskReq	!assessmentRequest(profile, amount)	G_1=[profile=p∧amount= a∧amount £10,000]	
!approveReq	!approveRequest(profile,amount)	G_2=[profile=p∧amount= a∧(amount>10,000∨r=" unknown")]	

?riskResp G_3	?assessmentResponse(risk)	G_3=[risk="low"]	r:="approved"
?riskResp ¬ G_3	?assessmentResponse(risk)	¬ G_3	r:="unknown"
?approveResp	?approveResponse(resp)		r:=resp
!loanResp G_4	!loanResponse(result)	G_4=[result=r∧(result="approved∨result="refused")]	
?loanReq*	?loanRequest(profile, amount)	G_5=[amount £10,000]	
!riskReq*	!assessmentRequest(profileObject, amount)		
!loanResp*	!loanResponse(result)		
!loanResp ¬ G_4	!loanResponse(result)	¬ G_4	
?loanReq1	?loanRequest(profileObject, amount)	¬ G_5	
?R1	?assessmentRequest(profileObject, amount) ?approveRequest(profileObject, amount) ?δ	¬ G_1 ¬ G_2	
?R2	?assessmentRequest(profile, amount) ?approveRequest(profile, amount) ?δ		
?R3	?assessmentRequest(profile, amount) ?approveRequest(profile, amount)		

Table 1: Symbol table

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 parameter valuations.

Definition 2 (ioLTS semantics) The semantics of an ioSTS $S = \langle L, l_0, V, V_0, I, \mathrm{L}, \rightarrow \rangle$ is the ioLTS $\|S\| = \langle Q, q_0, \Sigma, \rightarrow \rangle$ where:

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

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

This rule can be read as follows: for an ioSTS transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain an ioLTS transition $(l_1,v) \xrightarrow{a(p),q} (l_2,v')$ with v a valuation over the internal variable set, if there exists a valuation q such that the guard G evaluates to true with v and q. Once the transition is fired, the internal variables are assigned with v' derived from the assignment A over $v \cup q$. 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 now be derived from the ioLTS semantics:

Definition 3 (Runs and traces) For an ioSTS $S = \langle L, l_0, V, V_0, I, \Lambda, \rightarrow \rangle$, interpreted by its ioLTS semantics $\|S\| = \langle Q, q_0, \Sigma, \rightarrow \rangle$, a run $q_0 a_0 q_1 .. q_{n-1} a_{n-1} q_n$ 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 $F \times D_V \cap Q$ with F a location set in L. It follows that a trace of a run r is defined as the projection $proj_\Sigma(r)$ on actions. $Traces_F(S) = Traces_F(\|S\|)$ 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 4 (Compatible ioSTSs) An ioSTS $S_1 = \langle L_1, l^0_1, V_1, V^0_1, I_1, \Lambda_1, \rightarrow_1 \rangle$ is compatible with $S_2 = \langle L_2, l^0_2, V_2, V^0_2, I_2, \Lambda_2, \rightarrow_2 \rangle$ iff $V_1 \cap V_2 = \emptyset$, $\Lambda^I_1 = \Lambda^I_2, \Lambda^O_1 = \Lambda^O_2$ and $I_1 = I_2$.

Definition 5 (Parallel product $\|$) The parallel product of two compatible ioSTSs $S_1 = \langle L_1, l^0_1, V_1, V^0_1, I_1, \Lambda_1, \rightarrow_1 \rangle$ and $S_2 = \langle L_2, l^0_2, V_2, V^0_2, I_2, \Lambda_2, \rightarrow_2 \rangle$, denoted $S_1 \| S_2$, is the ioSTS $P = \langle L_P, l^0_P, V_P, V^0_P, I_P, \Lambda_P, \rightarrow_P \rangle$ such that $V_P = V_1 \cup V_2$, $V^0_P = V^0_1 \cup V^0_2$, $I_P = I_1 = I_2$, $L_P = L_1 \times L_2$, $l^0_P = (l^0_1, l^0_2)$, $\Lambda_P = \Lambda_1 = \Lambda_2$, The transition set $\rightarrow_P$ is the smallest set satisfying the following rule:

$$\frac{l_1 \xrightarrow{a(p),G_1,A_1}_{S_1} l_2, l_1' \quad \xrightarrow{a(p),G_2,A_2}_{S_2} l_2'}{(l_1,l_1') \xrightarrow{a(p),G_1 \wedge G_2, A_1 \cup A_2}_{P} (l_2,l_2')}$$

Lemma 1 (Parallel product traces) $Traces_{F_1 \times F_2}(S_1 \| S_2) = Traces_{F_1}(S_1) \cap Traces_{F_2}(S_2)$ with $F_1 \subseteq L_{S_1}, F_2 \subseteq L_{S_2}$.

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

Definition 6 (Mirrored ioSTS and traces) Let S be an ioSTS. $refl(S) =_{def}$ $\left\langle L_S, l_S^0, V_S, V_S^0, I_S, \mathrm{L}_{refl(S)}, ®_S \right\rangle$, where $\mathrm{L}^I_{refl(S)} = \mathrm{L}^O_S, \mathrm{L}^O_{refl(S)} = \mathrm{L}^I_S$. We extend the $refl$ notation on trace sets. $refl : (å^*)^* ® (å^*)^*$ 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).

RUNTIME VERIFICATION AND PASSIVE IOCO TESTING

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.

Runtime verification

The primary objective of runtime verification is to check whether an implementation I meets a set of properties expressed in trace predicate formalisms such as regular expressions, temporal logics or state machines. We propose to reuse the notion of observers (Chen & Wagner, 2002; Constant, Jeron, Marchand & Rusu, 2007) for modelling safety properties. An observer is an ioSTS specialized to capture the negation of a safety property by means of final "bad" locations. Runs, which lead to these locations, represent behaviours violating the property.

Definition 7 (Observer) An Observer is a deterministic ioSTS W composed of a non-empty set of sink locations $Violate_W Ì L_W$, called violation locations. W must be both input and output-enabled, i.e. for each state $(l,v) \in (L_\Omega / Violate_\Omega) \times D_\Omega$, and for each valued action $(a(p),\theta) \in \Lambda_\Omega \times D_p$, there exists $(l,v) \xrightarrow{a(p),\theta} (l',v') \in \rightarrow_{\|\Omega\|}$.

Given an ioSTS S, $Comp(S)$ stands for the set of compatible Observers of S. We shall also say that $Traces_{Violate_\Omega}(\Omega)$ gathers all the traces in $Traces(\Omega)$ which violate Ω.

Given an implementation I, I satisfies the Observer Ω if I does not yield any trace which also violates Ω:

Definition 8 (Implementation satisfies Observer) Let S be an ioSTS and I an implementation. I satisfies the Observer $\Omega \in Comp(\Delta(S))$, denoted $I \models \Omega$, if $Traces(\Delta(I)) \cap Traces_{Violate_\Omega}(\Omega) = \varnothing$.

Figures 2 and Table 1 illustrate an Observer example that expresses a safety property for the specification of Figure 1. The underlying property means that the receipt of a loan response, without requesting the Assessment service when the loan amount is less than $10,000, must never occur.

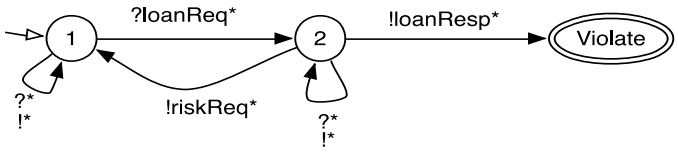


Figure 2: A safety property

Two Observers Ω_1 and Ω_2, describing two different safety properties can be interpreted by the Observer $\Omega_1 \| \Omega_2$ composed of a Violation location set $Violate_{\Omega_1} \times Violate_{\Omega_2}$. In the remainder of the paper, we shall consider only one Observer, assuming that it may represent one or more safety properties.

Ioco testing with proxy-testers

In the paper, we define conformance with the ioco test relation (Tretmans, 1996), which intuitively means that I conforms 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 9 An implementation I ioco-conforms to a specification S, denoted I ioco S if $Traces(\Delta(S)).(\delta \cup \{!d\}) \cap Traces(\Delta(I)) \subseteq Traces(\Delta(S))$.

We have shown in (IdentificationRemoved, 2012) that the ioco theory can be applied on passive testing with the concept of Proxy-tester. A Proxy-tester formally expresses the functioning of a transparent proxy, able to collect traces and to detect non-conformance without requiring to be set up in the same environment as the implementation one. We recall here the notion of Proxy-tester that shall be combined 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 the Canonical tester of the specification. A Canonical tester is an ioSTS containing 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:

Definition 10 (Canonical Tester). Let $S = \langle L_S, l^0_S, V_S, V^0_S, I_S, \Lambda_S, \rightarrow_S \rangle$ be an ioSTS and $\Delta(S)$ be its suspension. The Canonical tester of S is the ioSTS $Can(S) = \langle L_S \cup LF_{Can(S)}, l^0_S, V_S, V^0_S, I_S, \Lambda_{\Delta(refl(S))}, \rightarrow_{Can(S)} \rangle$ such that $LF_{Can(S)} = \{Fail\}$ is the Fail location set composed of the $Fail$ location. $\rightarrow_{Can(S)}$ is defined by the following rules:

$$\frac{t \in \rightarrow_{D(refl(S))}}{t \in \rightarrow_{Can(S)}}$$

$$\frac{a \in L_S^O \cup \{!d\}, l_1 \in L_S, G_a = \bigvee_{l_1 \xrightarrow{\sigma(p),G_a,A_a} l \in \rightarrow_{D(S)}} \neg G}{l_1 \xrightarrow{?a(p),G_a,A_a(x:=x')} Fail \in \rightarrow_{Can(S)}}$$

As an example, the Canonical tester of the ioSTS depicted in Figure 1 is illustrated in Figure 3. If we consider the location 2, new transitions to $Fail$ (grouped by the symbol ?R1) are added to model the receipt of unspecified actions.

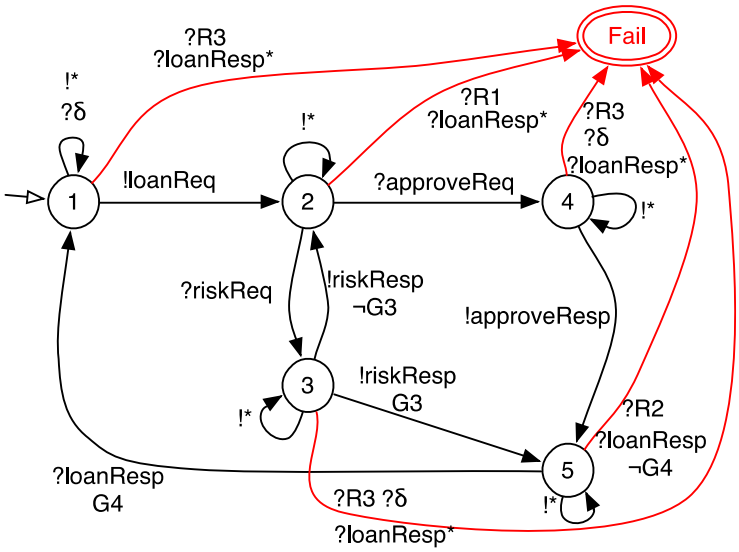
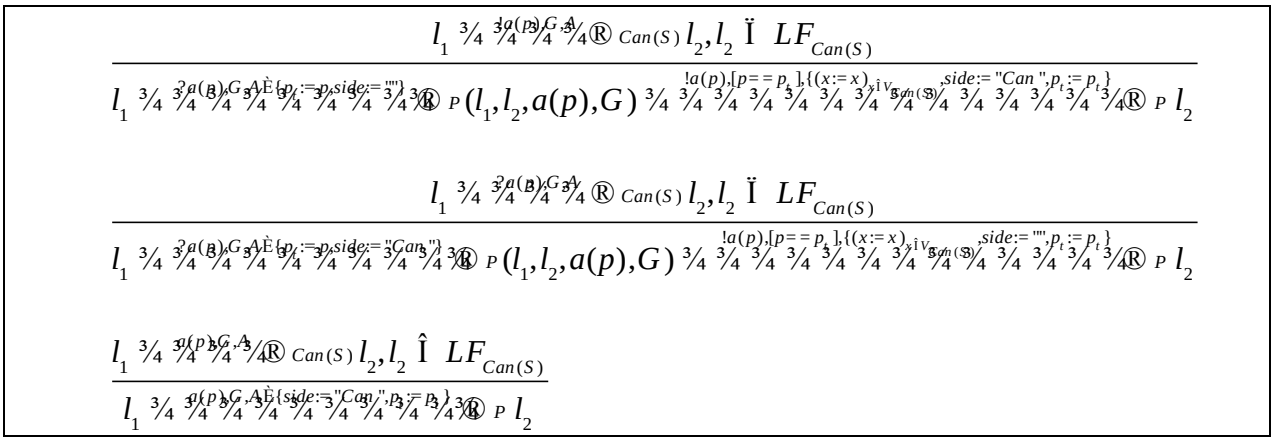


Figure 3: An ioSTS Canonical tester

The Proxy-tester of an ioSTS S corresponds to a Canonical tester where all the transitions, except those leading to $Fail$, are doubled to express the receipt of an event and its forwarding to its real recipient.

Definition 11 (Proxy-tester) The Proxy-tester of the ioSTS $S = \langle L_S, l^0_S, V_S, V^0_S, I_S, L_S, \rightarrow_S \rangle$ is the ioSTS $Pr(Can(S))$ where Pr is an ioSTS operation such that $Pr(Can(S)) =_{def} \langle L_P \cup LF_P, l^0_{Can(S)}, V_{Can(S)} \cup \{side, pt\}, V^0_{Can(S)} \cup \{side := "", pt := ""\}, I_{Can(S)}, L_P, \rightarrow_P \rangle$. $LF_P = LF_{Can(S)} = \{Fail\}$ is the Fail location set. L_P, L_P and $\rightarrow_P$ are constructed with the following rules:



Intuitively, the two first rules double the transitions whose terminal locations are not *Fail* to describe the functioning of a transparent proxy. The first rule means that, for an event (action or quiescence) initially sent to the implementation, the Proxy-tester waits for this event and then forwards it. The two transitions are separated by a unique location composed of the tuple $(l_1, l_2, a(p), G)$ to ensure that these two transitions, and only them, are successively fired. The last rule enriches the resulting ioSTS with transitions leading to $Fail$. A new internal variable, denoted $side$, is also added to keep track of the transitions provided by the Canonical tester (with the assignment $side := "Can"$). This distinction shall be useful to define partial traces of Proxy-testers and to express conformance with them.

As ioSTS specifications are input-enabled, a Proxy-tester accepts any output provided by the external environment. Since it is constructed from a Canonical tester, it also accepts any output provided by the implementation. Consequently, deadlocks can only occur in Proxy-testers when one of its $Fail$ states is reached.

Figure 4 depicts the resulting Proxy-tester obtained from the previous specification (Figure 1) and its Canonical tester (Figure 3). For readability, the dashed transitions stand for the transitions labelled by the assignment ($side := "Can"$) and the self-loop transitions, carrying "!*" in the Canonical tester, are gathered and only depicted with empty transitions and locations labelled by "!*". Figure 4 clearly illustrates that the initial behaviours of the Canonical tester are kept.

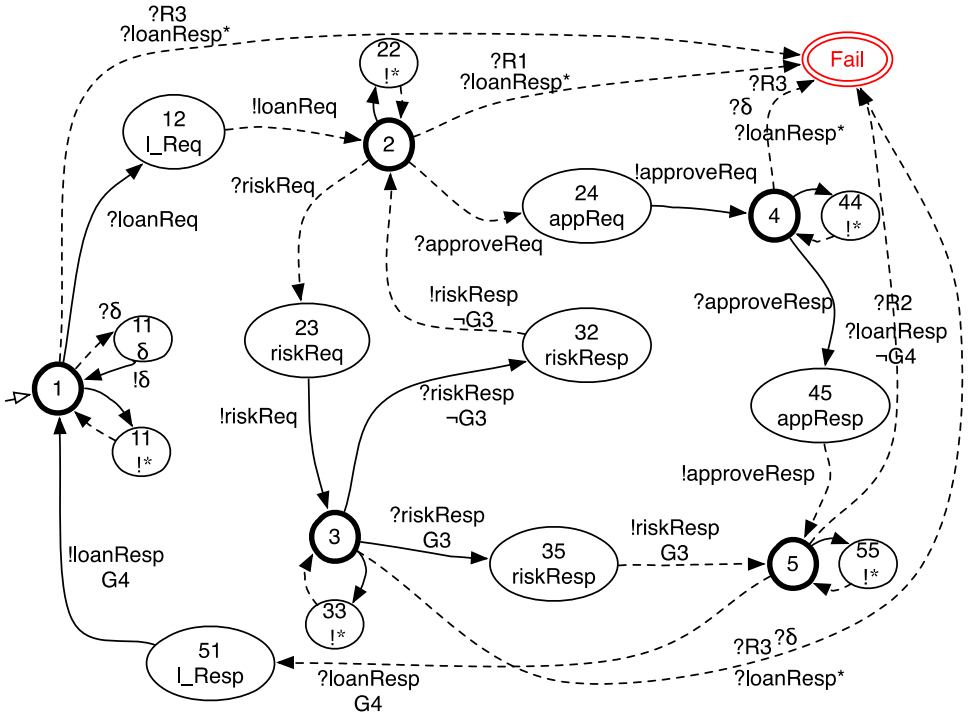


Figure 4: A Proxy-tester.

Previously, we have intentionally enriched Proxy-tester transitions with assignments on the variable side. These assignments help extract partial runs and traces in Proxy-testers:

Definition 12 (Partial runs and traces) Let P be a Proxy-tester and $\| P \| = < Q_P, q0_P, \Sigma_P, \rightarrow_P >$ be its ioLTS semantics. We define $Side : Q_P \rightarrow D_{V_P}$ the mapping returning the valuation of the side variable of a state in Q_P. $Side^E(Q_P) \subseteq Q_P$ is the set of states such that $Side(q) = E$. Let $RUN(P)$ be the set of runs of P. We denote $RUN^E(P)$ the set of partial runs derived from the projection $proj_{Q_P \text{å} {}_P Side^E(Q_P)}(RUN(P))$. It follows that $Traces^E(P)$ is the set of partial traces of (partial) runs in $RUN^E(P)$.

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

Proposition 1. Let S be an ioSTS, we have $Traces^{Can}(Pr(Can(S))) = Traces($ $Pr^{-1}(Pr(Can(S)))) = Traces(Can(S))$. In particular, $Traces^{Can}_{Fail}(Pr(Can(S))) = Traces_{Fail}($ $Can(S))$.

The definition of Pr^{-1} (with the proof of the previous Proposition is given in (IdentificationRemoved, 2013). The ioco relation can now be rephrased by:

Proposition 2. I *ioco* S Û $Traces(\mathrm{D}(I))$ Ç $refl(Traces_{Fail}^{Can}(Pr(Can(S))))$ = Æ.

So defined, *ioco* means that I conforms to its specification when the implementation traces do not belong to the set of partial Proxy-tester traces leading to $Fail$.

COMBINING RUNTIME VERIFICATION AND PROXY-TESTER

As defined, both Canonical testers and Observers describe undesired behaviours. This similarity tends to combine them to produce a model that could be used to detect both property violations and non-conformance. We call this model a Monitor. It results from the product of one Canonical tester $Can(S)$ with an Observer W and refines $Can(S)$ by separating the traces that violate safety properties among all the traces. A monitor is defined as:

Definition 13 (Monitor) Let $\mathrm{D}(S)$ be an ioSTS suspension and WÎ $Comp(\mathrm{D}(S))$ be an Observer. The Monitor of the Canonical tester $Can(S)$ and of the Observer W is the ioSTS $M = Can(S) \| refl(\mathrm{W})$.

A monitor M is still a Canonical tester in the sense that it owns a mirrored action set and can communicate with an implementation. It also has a Fail location set. But, it is specialised to recognise property violations.

As an example, the Monitor constructed from the previous Canonical tester (Figure 3) and the Observer of Figure 2 is depicted in Figure 5 and Table 1. It contains different verdict locations: *Fail* received from the Canonical tester, *Violate* received from the Observer and a combination of both *Fail/Violate*, which denotes non-conformance and the safety property violation. For example, the trace "?loanRequest("John_account";9000) !loanResponse("denied")" violates the Observer of Figure 2 because the action !assessmentRequest does not appear. This trace also reflects an incorrect behaviour because the response "denied" is incorrect. It must be either "approved" or ""rejected.

The combination of Canonical tester and Observer locations leads to new locations labelled by local verdicts. We group these locations in verdict location sets:

Definition 14 (Verdict location sets) Let $Can(S)$ be a Canonical tester and WÎ $Comp(\mathrm{D}(S))$ be a compatible Observer with $\mathrm{D}(S)$. The parallel product $M = Can(S) \| refl(\mathrm{W})$ produces new sets of verdict locations defined as follows:

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

We also denote $LF_M = FAIL$ È $FAIL / VIOLATE$, the *Fail* location set of *M*.

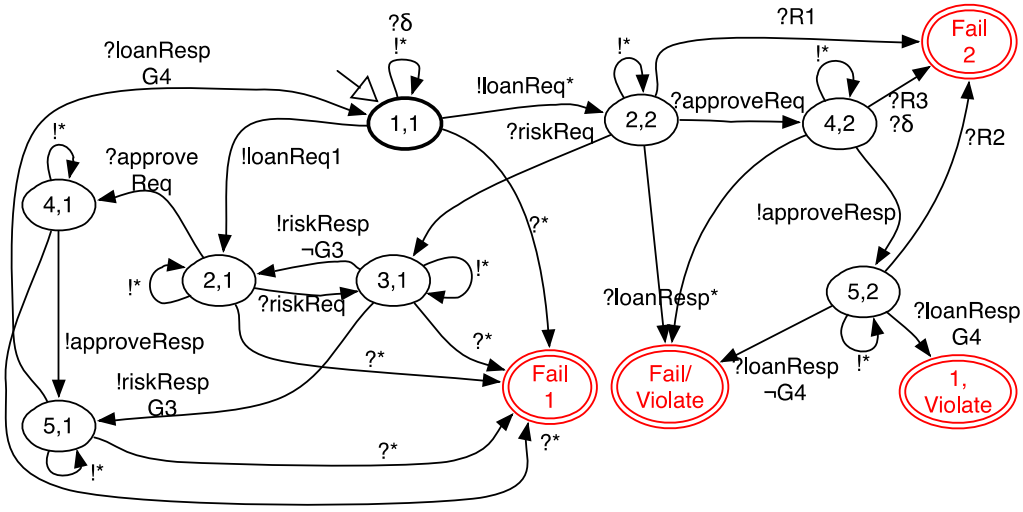


Figure 5: An ioSTS Monitor

As for the parallel product $\|$ (Lemma 1), the traces of a Monitor $M = Can(S) \| refl(\Sigma)$ can be expressed with the traces of the composed ioSTSs.

Lemma 2 (Monitor traces) Let M be a Monitor constructed from a Canonical tester $Can(S)$ and a compatible Observer $\Sigma \hat{I} Comp(\Delta(S))$. We have: $Traces_{F_1 \cup F_2}(M) = Traces_{F_1}($ $Can(S)) \cap Traces_{F_2}(refl(\Sigma))$.

In accordance with Definition 13, a Monitor can be seen as a Canonical tester, with a verdict location set *LF*, specialised for recognizing property violations. Consequently, to passively check whether an implementation meets safety properties, it sounds natural to apply the concept of Proxy-tester on Monitors. This gives a final model called Proxy-monitor:

Definition 15 (Proxy-monitor) Let M be a Monitor resulting from the parallel Product $Can(S) \| refl(\Sigma)$ with *S* an ioSTS and $\Sigma \hat{I} Comp(\Delta(S))$ an Observer compatible with the suspension of *S*. We denote *Pr(M)*, the Proxy-monitor of *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 $FAIL \cup FAIL/VIOLATE$ since it stems from a composition between an Observer and a Canonical tester. Except this difference, transitions of the Monitor are still doubled in its Proxy-monitor.

It remains to define formally the notion of passive monitoring of an implementation *I* by means of a Proxy-monitor. This 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 can be also modelled with an ioLTS suspension *Env* such that $refl(Env)$ is compatible with I and $Traces(\Delta(Env))$ is composed of sequences in $refl((\Sigma_{\Delta(I)})^*)$.

Definition 16 (Implementation monitoring) Let be $PM = \langle Q_{PM}, q^0_{PM}, \text{å}_{PM}, \text{®}_{PM} \rangle$ the ioLTS semantics of a Proxy-monitor *Pr(M)* derived from an ioSTS *S* and an Observer WÎ *Comp*(D(*S*)). QF_{PM} Í $Q_{PM} = LF_{Pr(M)} \cdot D_{V_{Pr(M)}}$ is its *Fail* state set. $I = \langle Q_I, q^0_I, \text{å}_I$ Í $\text{å}_M, \text{®}_I \rangle$ is the implementation model, assumed compatible with S and $Env = \langle Q_{Env}, q^0_{Env}, \text{å}_{Env}$ Ì $\text{å}_P, \text{®}_{Env} \rangle$ is the ioLTS modelling the external environment compatible with $refl(I)$. The monitoring of *I* by $Pr(M)$ is expressed with the product $\|_P(Env, PM, I) = \langle Q_{Env} \cdot Q_{PM} \cdot Q_I, q^0_{Env} \cdot q^0_{PM} \cdot q^0_I, \text{å}_{PM}, \text{®}_{\|_P(Env,PM,I)} \rangle$ where the transition relation $\text{®}_{\|_P(Env,PM,I)}$ is defined by the smallest set satisfying the following rules.

For readability reason, we denote an ioLTS transition $q_1 \xrightarrow{a} q_2$ if $Side(q_2) = E$ (the variable side is valued to *E* in q_2).

$$\frac{q_1 \xrightarrow{a} \text{®}_{D(Env)} q_2, q''_2 \xrightarrow{a} \text{®}_{D(I)} q''_3, q'_1 \xrightarrow{a} \text{®}_{PM} q'_2 \xrightarrow{!a} \text{®}_{PM} q'_3}{q_1 q'_1 q''_2 \xrightarrow{a} \text{®}_{\|_P(Env,PM,I)} q_2 q'_2 q''_2 \xrightarrow{!a} \text{®}_{\|_P(Env,PM,I)} q_2 q'_3 q''_3}$$

$$\frac{q_2 \xrightarrow{a} \text{®}_{D(Env)} q_3, q''_1 \xrightarrow{a} \text{®}_{D(I)} q''_2, q'_1 \xrightarrow{?a} \text{®}_{PM} q'_2 \xrightarrow{a} \text{®}_{PM} q'_3}{q_2 q'_1 q''_1 \xrightarrow{?a} \text{®}_{\|_P(Env,PM,I)} q_2 q'_2 q''_2 \xrightarrow{a} \text{®}_{\|_P(Env,PM,I)} q_3 q'_3 q''_2}$$

$$\frac{q_2 \xrightarrow{d} \text{®}_{D(Env)} q_3, q''_1 \xrightarrow{a} \text{®}_{D(I)} q''_2, q'_1 \xrightarrow{?a} \text{®}_{PM} q'_2, q'_2 \text{ Î } QF_{PM}}{q_2 q'_1 q''_1 \xrightarrow{?a} \text{®}_{\|_P(Env,PM,I)} q'_2}$$

One can deduce from the $\text{®}_{||_p(Env,PM,I)}$ definition that (Identificationremoved, 2012):

Proposition 3 We consider the notations of Definition 16. We have $Traces^{Can}(||_p(Env, PM, I)) = refl(Traces(D(I))) \cap Traces^{Can}(PM) = refl(Traces(D(I))) \cap Traces^{Can}(Pr(M))$.

The verdict list can now be drawn up from Definition 14. 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 4 (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 Ω Î *Comp*(D(*S*)). Let OT Í $Traces(||_p(Env, PM, I))$ be the observed trace set. If there exists s Î OT such that:

1. *s* belongs to $Traces_{FAIL/VIOLATE}(||_p(Env, PM, I))$, then *I* does not satisfy the safety property and *I* does not ioco-conform to *S*,

2. s belongs to $Traces_{FAIL}(||_p (Env, PM, I))$, then I does not ioco-conform to S. No violation of the safety property were detected on I,
3. s belongs to $Traces_{VIOLATE}(||_p (Env, PM, I))$, then I does not satisfy the safety property. Non-conformance between I and S were not detected.

Proof of Proposition 4 (sketch).

Proof of 1):

s belongs to $Traces_{FAIL/VIOLATE}(||_p (Env, PM, I))$. Therefore, $Traces_{FAIL/VIOLATE}($ $||_p (Env, PM, I)) \neq \varnothing$ and $Traces^{Can}_{FAIL/VIOLATE}(||_p (Env, PM, I)) \neq \varnothing$.

$Traces^{Can}_{FAIL/VIOLATE}(||_p (Env, PM, I)) = refl(Traces(\mathrm{D}(I))) \cap Traces^{Can}_{FAIL/VIOLATE}(Pr(M))$ (Proposition 3)

$Traces^{Can}_{FAIL/VIOLATE}(Pr(M)) = Traces_{FAIL/VIOLATE}(M)$ (Proposition 1).

It follows that $Traces^{Can}_{FAIL/VIOLATE}(||_p (Env, PM, I)) = refl(Traces(\mathrm{D}(I))) \cap$ $(Traces_{Fail}(Can(S)) \cap Traces_{Violate\Omega}(refl(\Omega))) \neq \varnothing$ (Lemma 2).

We deduce that $Traces^{Can}_{FAIL/VIOLATE}(||_p (Env, PM, I)) \neq \varnothing$ iff $refl(Traces(\mathrm{D}(I))) \cap$ $Traces_{Fail}(Can(S)) \neq \varnothing$ (a) and iff $refl(Traces(\mathrm{D}(I))) \cap refl(Traces_{Violate_{\Omega}}(\Omega)) \neq \varnothing (b)$.

From (a) and Propositions 1 and 2, we have $\not\equiv I$ ioco S.

From (b) and Definition 8, we have I —Ω.

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).□

APPLICATION TO WEB SERVICE COMPOSITION DEPLOYED IN CLOUDS

This section presents the practical application of our approach on Web service compositions deployed in Clouds. We particularly present the Proxy-monitor algorithms which are designed to test concurrent instances of a composite service.

Web service composition modelling

Web services are components offering some special features relative to the service-oriented architecture. Firstly, their methods, named operations, are called with the sending or the receipt of messages. To model them, we assume that an action $a(p)$ represents either the call of an operation op ($a(p) = opReq(p)$), or the receipt of an operation response ($a(p) = opResp(p)$). The set of parameters p must also be composed of these specific variables:

- the variable *from* stands for the calling partner (Web service or client), the variable *to* stands for the called partner,
- Web services may engage in concurrent interactions by means of several stateful instances called *sessions*, each one having its own state. For the delivery of incoming messages to the correct session, when several sessions are running in parallel, the usual solution is to add in messages a set of correlation values which match a part of the session state. Hence, an action

$a(p)$ must be composed of a valuation called *correlation set* to identify the session.

The use of correlation sets also involves assuming the following hypotheses so that clients and Web service instances can correctly be correlated:

Session identification: the specification is well-defined. When a message $a(p)$ is received, it always correlates with at most one session.

Message correlation: except for the first operation call which starts a new composition instance, an operation call $opReq(p)$ must contain a correlation set $coor \subseteq p$ such that a non-empty part $c \subseteq coor$ of the correlation set is composed of parameter valuations given in previous messages.

The first hypothesis results from the correlation functioning. The last one is added to correlate the successive operation calls of a given composite service instance.

The example depicted of Figure 1 represents a Web service composition. Nonetheless, to match the above message modelling, the ioSTS symbols have to be replaced with those of Table 2.

Symbol	Message	Guard	Update
?loanReq	?loanReq(from,to,profile, amount,corr)	from="Client"∧to="LoanA.Service"∧corr={profile, amount}	a:=amount p:=profile c1:=corr
!riskReq	!assessmentReq (from,to,profile, amount,corr)	from=”LoanA.Service"∧to="RiskA.Service"∧coor=c1∧profile=p∧amount=a∧amount £10,000	
!approveReq	!approveReq(from,to, profile, amount,coor)	from=”LoanA.Service"∧to="A.Service"∧coor=c1∧profile=p∧amount=a∧amount >10,000	
?riskResp G_3	?assessmentResp(from,to,risk, coor)	from=”RiskA.Service"∧to="LoanA.Service"∧coor=c1∧ risk=”low”	r:=risk
?riskResp ¬ G_3	?assessmentResp(from,to,risk, coor)	from=”RiskA.Service"∧to="LoanA.Service"∧coor=c1∧risk≠”low”	
?approveResp	?approveResp(from, to, resp, coor)	from=”A.Service"∧to="LoanA.Service"∧coor=c1	r:=resp
!loanResp G_4	!loanResp(from, to, result,coor)	from=”LoanA.Service"∧to="Client"∧coor=c1∧result=r∧(result=”approved∨result=”refused”)	

Table 2: Symbol table

Passive tester Implementation

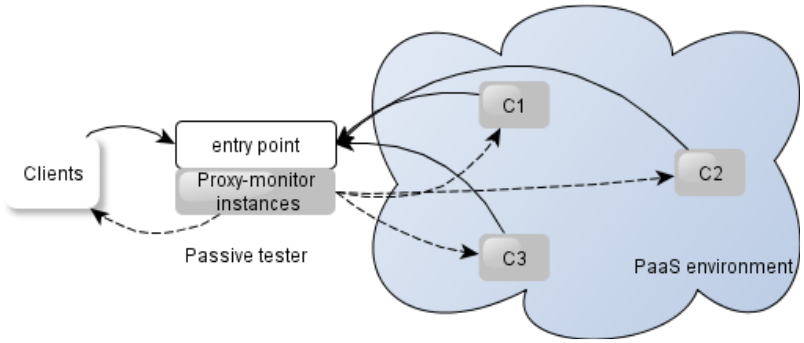


Figure 6: The passive tester architecture

The passive tester architecture, depicted in Figure 6, 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. To collect traces from several instances, several Proxy-monitor analysers are executed in parallel. Any incoming message received from the same composition instance must be delivered to the right Proxy-monitor analyser: this step is performed by a module called *entry point* which routes messages to Proxy-monitor analysers by means of correlation sets.

The entry point functioning is given in Algorithm 1. It handles a set L of pairs (p_i, PV) with p_i a Proxy-monitor analyser identifier and PV the set of parameter values received in previous messages. Whenever a message $(e(p), q)$ is received, its correlation set c is extracted to check if a Proxy-monitor analyser is running to accept it. This analyser exists if L contains a pair (p_i, PV) such that a subset $c \not\in$ c is composed of values of PV (correlation hypothesis). In this case, the correlation set has been constructed from parameter values received in previous messages. If an analyser is already running, the message is forwarded to it. Otherwise, (line 7), a new one is started. When an analyser p_i ends (line 9), the resulting trace set is stored in $Traces(Pr(M))$.

Algorithm 1: Entry point

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

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

The functioning of one Proxy-monitor analyser is described in Algorithm 2. 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., $(l0_{Pr(M)}, V0_{Pr(M)})$ and constructs runs stored in the $RUNS$ set. Whenever an event $(e(p), q)$ is received, with eventually q a valuation over p (line 3), it looks for the next transition which can be fired for each run r in $RUNS$ (line 7). This 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 valuations v and q. If a verdict location is reached (Definition 14) then the algorithm adds the resulting run r' to $RUNS$ and ends by returning the trace set T derived from $RUNS$ (lines 11-16). Otherwise, the event $(e(p), q)$ 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 $q_{next2} = (l_{next2}, v\emptyset)$. Then, the algorithm waits for the next event. The algorithm also ends when no new event is observed after a delay sufficient to detect several times quiescent states (set to ten times in the algorithm with the variable qt).

Algorithm 2: Proxy-Monitor analyser

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

1 $RUNS := \{(q_0 = (l0_{Pr(\mathcal{M})}, V0_{Pr(\mathcal{M})}))\}$;
2 $qt = 0$;
3 **while** *Event*$(e(p), \theta) \wedge$ *Fail is not detected* $\wedge qt < 10$ **do**
4 $r' = \varnothing$;
5 **if** $e(p) = !\delta$ **then**
6 $qt := qt + 1$;
7 **foreach** $r = q_0\alpha_0 ... q_i \in RUNS$ *with* $q_i = (l, v)$ **do**
8 **foreach** $t = l \xrightarrow{e(p),G,A} l_{next} \in \rightarrow_{Pr(\mathcal{M})}$ *such that* $G(\theta, v)$ *true* **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($t_2 = l_{next} \xrightarrow{!e(p),G_2,A_2}_{Pr(\mathcal{M})} l_{next2}$);
 ; // 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 $T = proj_{\Sigma_{||Pr(\mathcal{M})||}}(RUNS)$;

Algorithm 2 reflects exactly the monitoring of an implementation (Definition 16). It collect valued events and constructs traces of $||_p (Env, PM, I)$ by supposing that both I and Env are ioLTS suspensions. Definition 16 (implementation monitoring) is implemented in lines (7-21). When a verdict location lv is reached in particular (line 11 or 14), the analyser has constructed a run which belongs to $RUN_V(||_p (Env, PM, I))$ with V a verdict location set. From this run, we obtain a trace of $Traces_V(||_p (Env, PM, I))$. Having in mind Proposition 4, we can now state the correctness of the algorithm with:

Proposition 5 The algorithm has reached a location verdict in:

- FAIL/VIOLATE Þ $Traces_{FAIL/VIOLATE}(||_p (Env, PM, I))$ ¹ Æ Þ I — $(\Omega, Violate_\Omega)$ and Ø$(I \ ioco \ S)$,
- FAIL Þ $Traces_{FAIL}(||_p (Env, PM, I))$ ¹ Æ Þ Ø$(I \ ioco \ S)$,
- VIOLATE Þ $Traces_{VIOLATE}(||_p (Env, PM, I))$ ¹ Æ Þ I —$(\Omega, Violate_\Omega)$.

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. The synchronous method is particularly interesting for example when the implementation supports recovery actions whenever a fault is identified. However, this analysis can be also 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.

Pragmatically, the above proposition holds on condition that the delay involved in routing messages through Proxy-monitors is lower than the quiescence timeout. Furthermore, the message flows sent by the implementation under test and the client side have to be ordered. We assume that the messages flows are sent either through ordered queues or by means of a sequencing protocol (Oasis, 2009). When the implementation is a component-based system, the former case involves that each component is connected to the passive tester with one queue, but two actions of two different queues may be still interleaved. A formal solution for this problem is to consider an ioSTS subclass so that ioco holds despite the action interleaving, which is explicitly represented in the specification (Petrenko, Yevtushenko & Le Huo, 2003; Noroozi, Khosravi, Mousavi & Willemse, 2011). Another more pragmatic solution is to assume that the component clocks are synchronous and that timestamps are added into messages to support the message sequencing in the entry point. This solution is sound on condition that two successive actions could not be executed at the same time (the action execution is ordered). This is the solution chosen for the experimentation presented in the next Section.

Experimentation

We have implemented this approach in a tool called *CloudPaste* (Cloud PASsive Testing, IdentificationRemoved) to assess the feasibility of the approach. We experimented it with the Web service composition of Figure 1. To illustrate the flexibility of proxy-monitoring, we deployed this composition in two Clouds, Microsoft Azure (http://www.windowsazure.com) and

Google AppEngine (https://developers.google.com/appengine/): the Loan Approval service was coded in C# and deployed in Azure, the two other services were coded in Java and deployed in AppEngine.

The guard solving in Algorithm 2 is performed by the SMT (Satisfiability Modulo Theories) solver Z3 (http://z3.codeplex.com) that we have chosen since it offers good performance, takes several variable types and allows a direct use of arithmetic formulae. However, it does not support String variables. So, we extended the Z3 expression language with String-based predicates. In short, our tool takes Z3 expressions enriched with predicates. These are evaluated and replaced with Boolean values, then a Z3 script, composed of the current valuations and the guard, is dynamically written before calling Z3.

We generated a Proxy-monitor from the ioSTS of Figure 1 combined with five safety properties. The first is the one described in Figure 2. The other properties are based on basic security vulnerabilities. Quiescence was implemented with a timeout set to 10 seconds with respect to the HTTP timeout (usually set between 3 and 100 seconds). Client applications were simulated with at most 50 instances of Java applications performing requests to the Loan Approval service in a continuous loop with one of these scenarios:

1) a client asks for an amount greater than $10000, therefore the Loan Approval service calls the Approver service to retrieve a decision (4 messages),
2) a client asks for an amount less than $10000, hence the Loan Approval service calls the Assessment service which returns a high risk level, involving the call of the Approver service (6 messages).

Firstly, the experimentation showed that Proxy-monitor analysers can correlate the successive messages of one composition instance by means of the Message correlation assumption and hence can compute traces. We also injected some faults (modification of values, of messages, etc.) in the implementation code to check if these could be detected. The obtained results were promising since we detected all of them.

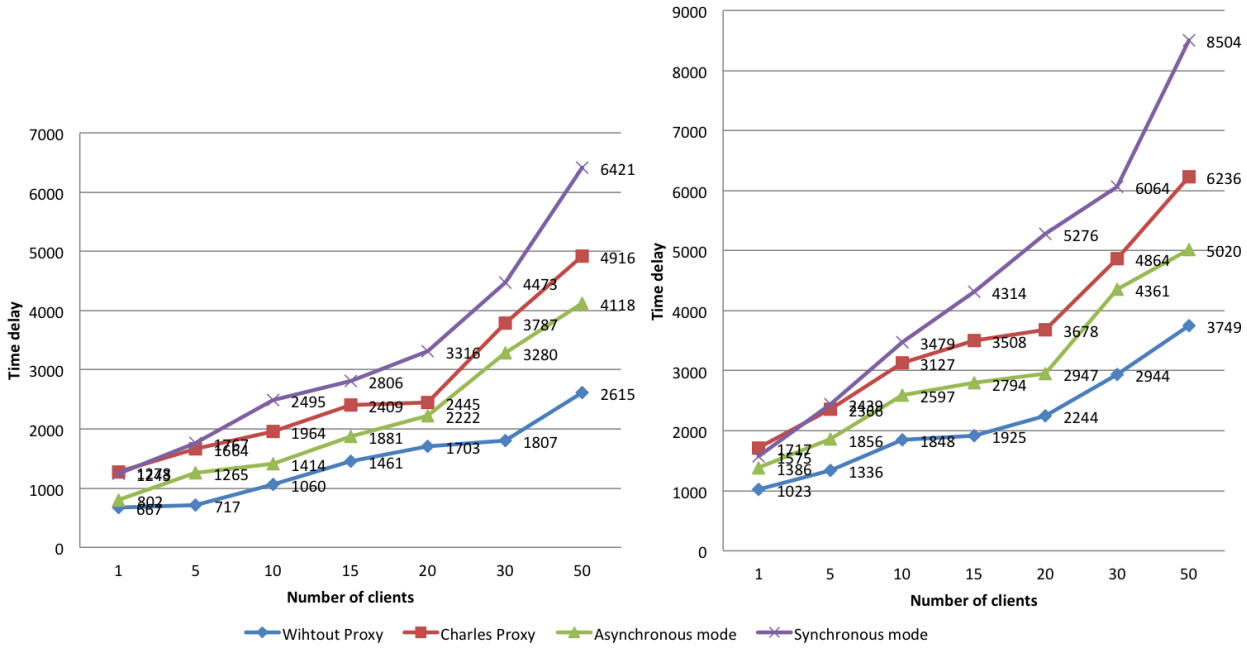


Figure 7: Processing time measurements.

Figure 7 depicts the average processing time (milliseconds) of one client performing one request when one up to 50 clients are running concurrently. Figure 7 (left) gives the processing time obtained when the clients execute the first scenario. Figure 7 (right) is dedicated to the second scenario. The curves represent the average time, without passive-tester, with the use of the debugging proxy Charles (http://www.charlesproxy.com), with CloudPaste in both asynchronous and synchronous modes.

Initially, the use of CloudPaste in asynchronous mode gives a reasonable higher processing time than not using any proxy (average difference of 1s with scenario 1 thus 250ms per message and 992ms with scenario 2 thus 166ms per message). This good performance originates from our choice of deploying CloudPaste in Azure. Indeed, the communication delays between Clouds are low, lower than the communication delays measured between our clients, executed in a computer settled in Vietnam, and the Clouds. Depending on the client and the passive tester locations, the processing times could be further reduced. With both scenarios, the average time processing is far lower than the quiescence timeout (and than the HTTP timeout as well). CloudPaste offers a better performance than Charles: for instance, when 20 clients are launched, CloudPaste processes one request with 223ms less than Charles with scenario 1 (731ms with scenario 2).

In synchronous mode, the checking overhead with CloudPaste becomes higher than using Charles though. The difference of delay equals to 871ms for 20 clients with scenario 1 (1.6s with scenario 2). This difference, which is not surprising, results from the call of Z3 to check if guards are satisfiable. With more than 50 simultaneous clients, we started to reach the limitations of CloudPaste as it is implemented at the moment. Indeed, neither the entry point nor the call of Z3 are multithreaded. This leads to a bottleneck with a high number of clients.

Nevertheless, in synchronous mode and even with 50 clients, the processing time (8504ms with scenario 2, thus 1417ms per message) is still lower than the timeout set to observe quiescence (the testing process can be done) and than the HTTP timeout (messages can be forwarded correctly). This mode is also particularly interesting since it offers the advantage to eventually implement recovery action calls, e.g., error compensation or implementation reset, whenever an error is detected. Error recovery is not possible with classical proxies or in asynchronous mode.

These results, obtained from a composition deployed in two different Clouds tend to show that our approach represents an efficient testing solution and that it can be done in real-time.

CONCLUSION

We have proposed a testing method which combines passive conformance testing with runtime verification. Our method generates Proxy-monitors from safety properties and specifications modelled with ioSTSs. Then, it simultaneously checks if an implementation ioco-conforms to its specification and meets safety properties. Proxy-monitors are 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, on condition that they could be configured to send messages through a proxy.

In this paper, we have dealt with deterministic specifications, like many testing approaches proposed in the literature. A direct solution for considering nondeterministic ioSTSs is to apply determinization techniques (Jéron, Marchand & Rusu, 2006) on them. In a future work, we could

also consider nondeterministic ioSTSs and a weaker test relation than ioco to generate nondeterministic Proxy-monitors. Another direction for future research is to focus on security testing instead of conformance. Indeed, a Proxy-monitor exhibits the functioning of a transparent intermediary between clients and the implementation under test. It could be modified to separate the actions received from external entities (clients or other components) to those produced by the implementation itself. As a consequence, a Proxy-monitor could protect the implementation from external attacks while checking if it meets security policies expressed with safety properties (Schneider, 2000). This would result in a kind of specialised application firewall associated with a security passive testing tool.

REFERENCES

Arthoa, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., & Washington, R. (2005). Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3), 209–234.

Alcalde, B., Cavalli, A. R., Chen, D., Khuu, D., & Lee, D. (2004). Network protocol system passive testing for fault management: A backward checking approach. In Frutos-Escrig, D. & Nunez, M. (Ed.), *FORTE, Vol 3235, Lecture Notes in Computer Science*, (pp 150–166). Springer.

Andres, C., Cambronero, M. & Nunez, M. (2011). Passive testing of web services. In Bravetti M. & Bultan, T., (Ed.), *Web Services and Formal Methods, Vol 6551, Lecture Notes in Computer Science*, (pp 56–70). Springer Berlin / Heidelberg.

Bayse, E., Cavalli, A. Nunez, M. & Zaidi, F. (2005). A passive testing approach based on invariants: application to the wap. *Computer Networks*, Vol 48, 247–266, Elsevier Science.

Barringer, H., Goldberg, A., Havelund, K. & Sen, K. (2004). Rule based runtime verification. In Steffen, B. & Levi, G., (Ed.), *VMCAI, Vol 2937, Lecture Notes in Computer Science*, (pp 44–57), Springer.

Barringer, H., Gabbay, D. & Rydeheard, D. (2007). From runtime verification to evolvable systems. In I*n the 7th international conference on Runtime verification, RV'07*, (pp 97–110), Springer-Verlag.

Cavalli, A., Benameur, A., Mallouli, A. & Li, K. (2009). A Passive Testing Approach for Security Checking and its Practical Usage for Web Services Monitoring. In *NOTERE 2009*.

Constant, C., Jéron, T., Marchand, H. & Rusu, V. (2007). Integrating formal verification and conformance testing for reactive systems. *IEEE Trans. Softw. Eng.*, 33(8), 558–574.

Che, X., Lalanne, F., & Maag, S. (2012). A logic-based passive testing approach for the validation of communicating protocols. *In ENASE 2012 – Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering*, Wroclaw, Poland, (pp 53–64).

Chen, H & Wagner, D. (2002). Mops: an infrastructure for examining security properties of software. In *the 9th ACM Conference on Computer and Communications Security*, (pp 235–244). ACM Press.

D'Amorim, M. & Havelund, K. (2005). Event-based runtime verification of java programs. In *the 3rd international workshop on Dynamic analysis*, (pp 1–7), ACM Press.

D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S. & Manna, Z. (2005). Lola: Runtime monitoring of synchronous systems. In *the 12th International Symposium on Temporal Representation and Reasoning*, (pp 166–174), IEEE Computer Society.

Falcone, Y., Jaber, M., Nguyen, T. H., Bozga, M. & Bensalem, S. (2011). Runtime Verification of Component-Based Systems. In *9th International Conference on Software Engineering and Formal Methods.*

Havelund, K. & Rosu, G. (2002). Synthesizing monitors for safety properties. In *the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, (pp 342–356), Springer-Verlag.

Jéron, T. Marchand, H. & Rusu, V. (2006). Symbolic determinisation of extended automata. In Navarro, G., Bertossi, L. & Kohayakawa, Y., (Ed.), *Fourth IFIP International Conference on Theoretical Computer Science, Vol 209, IFIP International Federation for Information Processing*, (pp 197–212), Springer US.

Jordan, D. & Evdemon, J. (2007). Web Services Business Process Execution Language Version 2.0, *OASIS Standard*, (pp 179-183), from http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

Lalanne, F., Che, X., and Maag, S. (2011). Data-centric property formulation for passive testing of communication protocols. *In Proceedings of the 13th IASME/WSEAS, ACC'11/MMACTEE'11*, (pp 176–181).

Lee, D., Chen, D., Hao, R., Miller, R. E., Wu, J. & Yin, X. (2006). Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Trans. Netw.*, 14:424–437.

Leucker, M. & Schallhart, C. (2009). A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5), 293–303.

Miller, R.E. & Arisha, K.A. (2000). On fault location in networks by passive testing. In *IEEE International Conference on Performance, Computing, and Communications*, (pp 281–287).

Montesi F. & Carbone, M. (2011). Programming services with correlation sets. In Kappel, G., Maamar, Z. & Motahari-Nezhad, H. M., (Ed.), *ICSOC, Vol 7084, Lecture Notes in Computer Science*, (pp 125–141), Springer.

Noroozi, N., Khosravi, R., Mousavi, M. R., & Willemse, T. A. C., (2011), Synchronizing asynchronous conformance testing, *Proceedings of the 9th international conference on Software engineering and formal methods, SEFM'11*, 334—349, Montevideo, Uruguay, Springer-Verlag.

Oasis (2009) Web Services Reliable Messaging (WS-ReliableMessaging), http://docs.oasis-open.org/ws-rx/wsrm/200702.

Pellizzoni, R., Meredith, P., Caccamo, M. & Rosu, G. (2008). Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium*, (pp 481–491), IEEE Computer Society.

Petrenko A., Yevtushenko N., & Le Huo J. (2003), Testing Transition Systems with Input and Output Testers, *TESTERS, PROC TESTCOM 2003*, SOPHIA ANTIPOLIS, 129--145, Springer-Verlag.

IdentificationRemoved (2012)

IdentificationRemoved (2013)

Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50.

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