Extended Proxy-tester

Laura Brandán Briones* and Agnes Madalinski†
*CONICET and Fa.M.A.F., Universidad Nacional de Córdoba, Argentina, Email: lbrandan@famaf.unc.edu.ar
†Chair of Software Engineering, Otto-von-Guericke-University Magdeburg, Germany, Email: amadalin@gmail.com

***Abstract*—We present a proxy-tester, which is an engine for ioco-passive testing. We show characteristics of soundness and completeness of our approach with respect to the ioco-testing relation. Later, we present a novel framework that combines our ioco-passive tester with the check of possible attacks from malicious users.**

I. Introduction

Today, we are facing a *smart* world where embedded devices with electronic and software functions, making them more and more complex. Many of these systems are critical and require the highest dependability standards because a system failure might cause injuries or even deaths. A real challenge is to ensure that a system operates, during its functioning, properly i.e. according to its specification. To identify any dysfunction due to an error occurrence is a demanding task. Therefore, early detection of errors is the key to support system performances, ensuring system safety, and increasing system life.

Testing is a well known practice in industry as well as in research. Testing's aim is to execute an implementations of a system to find failures, i.e. discrepancies between its real behavior and the intended behavior described by a specification. In particular, the model-based testing approach formally describes the system to be tested with a specification model that express its correct behavior. Two advantages are achieved with this practice: first formal techniques can be applied to the model and second the testing process can be automatized. Particularly, our research is based on model-based testing (MBT), i.e. the research area that comprises the usage of models to automate test activities and generate test from the model [1].

There are two complementary approaches to test implementations: a) active testing: where test cases are derived from the specification and then executed trigging its implementation checking if the implementation conforms to (w.r.t. a given relation) the specification; and b) passive testing: where a monitor passively observes the implementation without disturbing it and checks if the sequences of observed events conform to (again, w.r.t. a given relation) the specification. These two approaches are usually applied to different states of the implementation process. Normally active testing is done before the implementation is delivered, in order to realize if some changes should be done in it; meanwhile passive testing is done when the implementation is already in use, because testing infinitely is impossible or because the system is already in use.

Accordingly, passive testing methods aim to detect faults by passively observing the implementation input/output events, without interrupting its normal behavior. Commonly, the tester corresponds to a kind of sniffer, which can catch the inputs that go to the implementation and the outputs that the implementation gives to the environment, where it is running. Then, this information is used to check if the implementation behavior corresponds or not to the desired specification behavior.

In this paper we initially present a passive testing approach based on the notion of a Proxy-tester similar as the one presented in [2]. The idea is to make an intermediate (the Proxy-tester) between the implementation and the user. This Proxy-tester receives inputs from the environment and forwards them to the implementation and does the opposite with outputs, meanwhile it checks for the **ioco** conformance testing relation.

There are two important notions here. First, we recall that complete test suites are infinite, and thus, not practically executable. Our contribution gives another instance of testing to aid in the naturally incomplete testing phase, but it does not try to overcome the testing phase. Second, because of the previous point more than checking for conformance our approach checks for non-conformance, meaning that more than validating implementations we find out **ioco**-incorrect implementations.

It is known that **ioco** testing theory has the assumption that implementations are input enabled, i.e. that they can never refuse an input that the environment gives to it. If

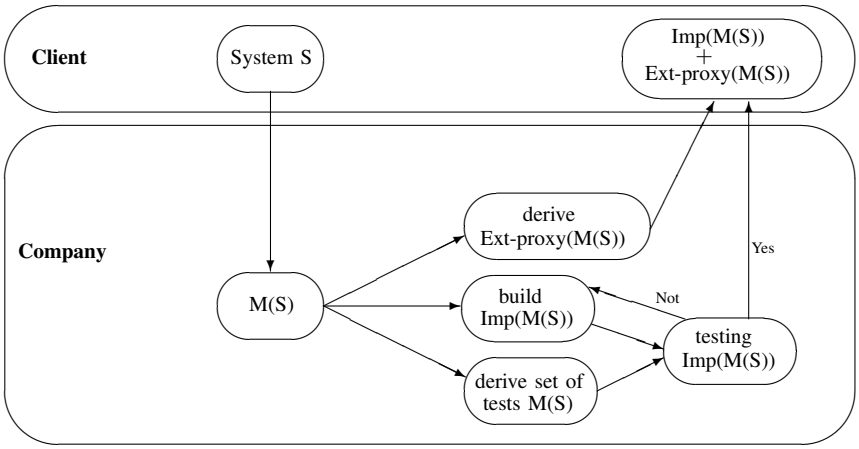

Fig. 1: Motivation

it is known that an implementation was checked with **ioco** testing then the input enabled requirement could allow malicious users to trigger the implementation with a malicious purpose. Thus, using the advantage that our Proxy-tester receives events from the environment and it passes them trough to the implementation we can also be aware of the non specified inputs to rise an alarm, etc. This is the other contribution of this paper.

Therefore, we present a novel framework that integrates passive testing identifying failures with respect to the **ioco** model-based testing theory and at the same time informs if a malicious user is trying to provoke the implementation with a non specified input of the specification. Hence, the possibly incorrect implementations that are not discovered by the testing performed before delivering, these failures, when they occur, can be detected by the Extended-Proxy engine. This gives an extensive inside of the erroneous behavior of the monitored system. Moreover, we show that the set of failures that **ioco** testing framework caught is still being caught, proving that our approach is safe with respect to that theory.

The idea behind our proposed framework is illustrated in Figure **??**. Suppose a client orders a system to a company. Then, the company builds such as system from the given specification and performs all the tests that it can. Note that if necessary the implementation is modified to satisfy the testing relation. Naturally, this phase has to be finite so even an infinite test could be derived only a finite number of them can be executed. Meanwhile, an Extended-Proxy (the passive Testing engine and the malicious-user checker) is derived. Finally, after testing is successfully performed the implemented system and the Extended-Proxy are delivered to the client. So, with the Extended-Proxy the client is able to perform test failures from an incorrect implementation (which had not been caught in the inherently incomplete testing) and to, at the same time, controls for the possible attacks of malicious-users.

The paper is organized as follows. First, in Section III a theoretical background is presented introducing input-output transition system (IOTS) and some notions related to our approach. In Section IV we make a short introduction to the **ioco** model-based testing theory. Second, in Section V our Proxy-tester is defined, and we prove some properties. Third, in Section VI we extend the Proxy-tester and finally in Section VII conclusions are drawn and future works are presented. Throughout the paper we illustrate the presented notions with examples.

II. Related work

The theory of testing has become a strong subject of research, in particular, the application of formal methods in the area of model-driven testing has led to a better understanding of the conformance notion between implementations and specifications. Automate generation methods of test suites from specifications [3], [4], [5], [1] where developed, which have led to a new generation of powerful test generation and execution tools such as SpecExplorer [6], TorX [7] and TGV [8].

A clear advantage of a testing formal approach is the provable soundness of the generated test suites, i.e. the property that each generated test suite will only reject implementations that do not conform to the given specification. In many cases also an exhaustiveness result is obtained, i.e. the property that for each non-conforming

implementation a test case can be generated that will expose all errors (cf. [3]). In practice, the above notion of exhaustiveness is usually problematic, since exhaustive test suites will contain infinitely many tests. This raises the question on test selection, i.e. the selection of well chosen, finite test suites that can be generated (and executed) within the available resources [9]. Commonly, it is clear that some tests will not be executed before product delivery (maybe the longer ones or the non common ones, etc.).

On the other hand passive testing offers a continues monitoring for systems under test while they are in operation without disturbing them. The **ioco** passive testing has been introduced in [2], where traces are extracted by means of a Proxy-tester which represents an intermediary between client applications and the implementation (i.e. without running in the same environment as the implementation). Later, a combination of **ioco** passive testing and runtime verification has been presented in [10].

Other works on passive testing are related to networks [11], protocols [12] and Web services [2]. They rely on sniffer-based tool as a central point to extract all the client requests and implementation reactions (messages, packets, etc.). A restricted access to the implementation environment is considered in [13], where a Proxy-tester has been presented. Such a Proxy-tester represents an intermediary between client applications and the implementation, which is able to receive the client traffic and to forward it to the implementation and vice versa.

In Section VI we consider the possibility that a malicious-user is trying to use the system in a manner that was not specified by the specification, so our proxy can inform us about this unexpected behavior from the user. This analysis can also be related with aspect-oriented programming where to increase modularity a separation of concerns is done, namely a cross-cutting concerns (security, profiling, etc.) [14], [15]. Where additional behavior is added to the code without modifying the code itself, similarly as our proxy. We plant to investigate further this relation to establish similarities.

III. Preliminaries

We use standard definitions for labeled transition systems. Let Σ be any set of events. Then, with Σ^* and Σ^ω we denote the set of all finite and infinite sequences over Σ. With $\sigma \sqsubseteq \rho$ we denote that σ is a prefix of ρ.

An *input-output transition system* is an IOTS with the set of observable events, Σ, subdivided into input events Σ_I and output events Σ_O. Formally:

Definition 1 (IOTS) *A* input-output transition system*, denoted IOTS, is a tuple* $L = \langle Q, q_0, \Sigma_{I,O}, T \rangle$*, where*

- Q *is a finite set of states;*
- $q_0 \in Q$ *is an initial state;*
- $T \subseteq Q \times \Sigma_\tau \times Q$ *is a finite branching transition relation;*
- Σ *is a finite set of events partitioned into: a set of* input events Σ_I *and* output events Σ_O *(i.e.* $\Sigma = \Sigma_I \cup \Sigma_O$ *&* $\Sigma_I \cap \Sigma_O = \emptyset$*); and* $\Sigma_\tau = \Sigma \cup \{\tau\}$ *where* τ *denote unobservable events.*

As normally we use '?' to denote input events ($a? \in \Sigma_I$) and '!' to denote outputs events ($a! \in \Sigma_O$).

Figure 2 depicts an example of a IOTS, which will be used throughout this paper as a running example.

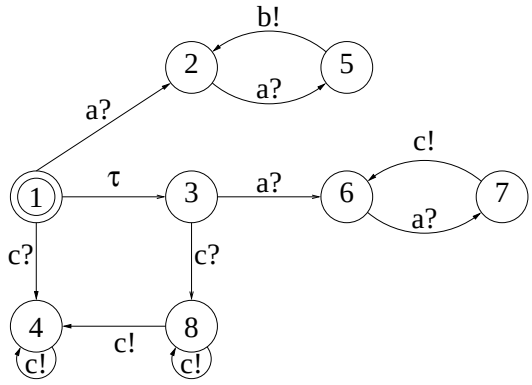

Fig. 2: input-output transition system

Definition 2 *Let* $L = \langle Q, q_0, \Sigma_{I,O}, T \rangle$ *be a IOTS, then*

- *A* path *in L is a sequence* $\rho = q_0 a_0 q_1 \ldots$ *such that for all i we have* $(q_i, a_i, q_{i+1}) \in T$*. We denote by* $\mathsf{paths}(q)$ *the set of paths starting in q. The set of* $\mathsf{cycle}(L)$ *is the set of paths starting and ending in the same state for all states in L. The trace* σ *of a path* ρ*, is the sequence* $\sigma = a_0 a_1 \ldots$ *of events in* Σ_τ *occurring in* ρ*, so* $\sigma \in \Sigma_\tau^*$*;*
- *We use* $q \xrightarrow{a} q'$ *in case that there exists an event a such that* $(q, a, q') \in T$*, we use* $q \xrightarrow{a}$ *to denote that there exists a state* q' *such that* $q \xrightarrow{a} q'$*, and we use* $q \rightarrow$ *to denote that there exists an event a and a state* q' *such that* $q \xrightarrow{a} q'$*;*
- *We write* $q \overset{\epsilon}{\Rightarrow} q'$ *in case that* $q = q'$ *or there exist states* $q_1, \cdots, q_n$ *such that* $q \xrightarrow{\tau} q_2, \cdots, q_n \xrightarrow{\tau} q'$*. Moreover, for a given event* $a \in \Sigma$ *we write* $q \overset{a}{\Rightarrow} q'$ *if there exist* q_1, q_2 *such that* $q \overset{\epsilon}{\Rightarrow} q_1, q_1 \xrightarrow{a} q_2, q_2 \overset{\epsilon}{\Rightarrow} q'$*;*
- *In the case* σ *is finite, with* $|\sigma|$ *we denote the length of the trace* σ*;*

- *The* observable trace *of a trace σ, denoted $obs(\sigma)$, is the sequence $a_0 a_1 \ldots$ of events in Σ occurring in σ;*
- *Given a trace $\sigma \in \mathsf{traces}(L)$, we write L* **after** *σ to denote all the states that we can arrive after the observable events of σ, i.e. L* **after** *$\sigma = \{q \subseteq Q \mid q_0 \overset{\sigma}{\Rightarrow} q\}$*
- *Given a set of states $Q' \subseteq Q$, we write $out(Q')$ to the set of output events that are allowed in all states of the set Q', i.e. $out(Q') = \{a \in \Sigma_O \mid \exists\, q \in Q' : q \overset{a}{\Rightarrow}\}$.*

For example, in Figure 2, $\pi_1 = q_1 a?q_2 a?q_5 b!q_2$ is a path, $\pi_2 = q_2 a?q_5 b!q_2$ is a cycle, $\text{trace}(\pi_1) = a?a?b!$ and $|\text{trace}(\pi_1)| = 3$. Moreover, if $\sigma = a?a?c!\tau a?c!$ then its observable trace is $obs(\sigma) = a?a?c!a?c!$.

We restrict our work to *convergent* and *live* IOTS, meaning that for all L ∈ IOTS that we work with, each cycle path has at least one observable event

$$\forall\, \pi \in \mathsf{cycle}(L) : \exists\ a \in \Sigma : a \in \pi \tag{1}$$

and for all states there exists a transition initiated in that state, i.e.

$$\forall\ q \in Q : q \rightarrow \tag{2}$$

IV. Testing

Testing is the process of executing a system trying to realize if a failure occurred. Testing should produce observable, unambiguous and consistent results. If a failure occurred it is supposed that there exists a fault that produced that failure. Following these ideas, we treat faults as unobservable and failures as observations that tell us that an unobservable fault occurred.

Model based testing relies on models of the system under test and its specification to automate test case derivation, test decision, etc. The idea is that the testing process tries to find a discrepancy between the specification model and the implementation.

A. **ioco** *model-based testing theory*

We recall the basic theory about test derivation from input-output transition systems similarly as it was initially defined in [3], where it is possible to find a more detailed exposition. Although, our approach followed the **ioco** testing theory, we believed that it can be applied to any testing relation with finites tree shape tests.

As in **ioco** model-based testing faults are not considered to be modeled but only to exists, we simply treat faults as unobservable events. On the other hand, as normally is done in testing we consider failures as observable discrepancy between behaviors, the one from the implementation with respect to the one from the specification.

An important contribution of the **ioco** testing theory is the quiescence concept (i.e. the absence of outputs), given that trace inclusion with the quiescence concept is more powerful than simple trace inclusion. Thus, we incorporate quiescence in specification, by adding a self loop $q \xrightarrow{\delta} q$ labeled with a special label δ to each quiescent state q, i.e. $\forall\, q \in Q$ with $\forall\, a \in \Sigma_O : q \overset{a}{\not\rightarrow}$ and considering δ as an output event.

As normally is done in model-based testing, we restrict our work to *input-enabling* IOTS implementations, meaning that for all P implementation that we work with, all inputs are accepted in all states, i.e.,

$$\forall\, q \in Q^P : \forall\, a? \in \Sigma_I^P : q \overset{a?}{\Rightarrow} \tag{3}$$

So, we formally recall the definition of the **ioco** testing relation to relate an input-enabled implementation with a specification modeled as IOTS, where for all specification traces the implementation outputs should be a subset of the allowed specification outputs.

Definition 3 *Given a specification $S \in IOTS$ and an input-enabled implementation $P \in IOTS$ then:*

$$P \textbf{ ioco } S \textit{ if and only if } \forall\, \sigma \in \mathsf{traces}(S) : out(P \textbf{ after } \sigma) \subseteq out(S \textbf{ after } \sigma)$$

As an example consider $S = \langle \{q_0, q_1, q_2\}, q_0, \{a?, b!\}, \{(q_0, a?, q_1), (q_1, b!, q_1), (q_0, \tau, q_2), (q_2, b!, q_2)\}\rangle$ a specification system and $P = \langle \{q_0, q_1\}, q_0, \{a?, b!\}, \{(q_0, a?, q_1)\}\rangle$. It is clear that P is not **ioco** correct to S. Because the trace $\sigma = \delta$ is a trace in the specification, i.e. $\sigma \in \mathsf{traces}(S)$ and the set of outputs from the implementation after σ is: $out(P \textbf{ after } \delta) = \{\delta\}$ and the set of outputs from the specification is: $out(S \textbf{ after } \delta) = \{b!\}$.

As follows we present how **ioco** tests are derived in the canonical way to be able to understand how we will construct our passive **ioco** tester. The **ioco** test cases are adaptive, that is, the next event to be performed (observe the system, stimulate the system or stop the test) may depend on the test history, i.e. the trace observed so far. If, after a trace σ, the Tester decides to stimulate the system with an input $a?$, then the new test history becomes $\sigma a?$; in case of an observation, the test accounts for all possible continuations $\sigma b!$ with $b! \in \Sigma_O$ an output event (also, the quiescent events are considered as outputs in this step). Figure 3 shows three different tests of the specifications presented in Figure 2. The failures state are represented as an octagon with a cross meaning

that the system is in a failure if it arrives to that state, we call them error-states.

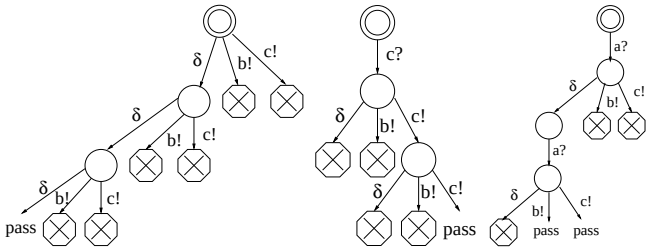

Fig. 3: Tests

The **ioco** theory requires that tests are *fail fast*, i.e. stop after the discovery of the first failure, and never fail immediately after an input. Formally, a test case consists of the set of all possible test histories obtained in this way.

Definition 4 *Given a specification S ∈ IOTS then,*

- *A* test case *(or test) t is a finite, prefix-closed subset of* Σ^* *such that*
  - *if* $\sigma a? \in t$*, then* $\sigma b \notin t$ *for any* $b \in \Sigma$ *with* $a? \neq b$*;*
  - *if* $\sigma a! \in t$*, then* $\sigma b! \in t$ *for all* $b! \in \Sigma_O$*;*
  - *if* $\sigma \notin \mathsf{traces}(L)$*, then no proper suffix of* σ *is contained in t.*
- *The* length $|t|$ *of test t is the length of the longest trace in t, i.e.* $|t| = \max_{\sigma \in t} |\sigma|$.

Because **ioco**-testing is proven [3] to be exhaustive we know that all failures can be exposed by its tests. Following the ideas from [9], it is possible to see the set of all tests derived from a given specification *S* as a big tree where we combine inputs and outputs. The merging of all tests from a given specification is called Mother Tree of Tests (MTT), where from the initial state we do all specification input event and we consider all output events (including quiescent). Figure 4 shows a graphic representation of a Mother Tree of our running example *S*, where the infinite following of the drawing is represented by dots in the bottom.

The MTT has some important characteristics:

- commonly it has infinite length;
- it has only failures as leafs; and
- it is deterministic.

We lift all concepts and notations (e.g. traces, etc.) that have been defined for *IOTS*s to MTTs. In a sense, our MTT is a representation of all tests such that each state that is not an error stats means that we could stop testing or we can follow.

V. Proxy for passive ioco testing

The idea of our *Proxy-tester* is to passively check for **ioco** conformance relation to overcome with the inheritable incomplete testing phase, generating a monitoring systems with all the MTT information from a given specification.

The Proxy-tester of a given specification is a deterministic IOTS extended with an error-state that tells us that a not specified output happened. The advantage of our method is that we extend the testing time towards infinite time, that is the finite time of the execution of some derived test during the testing phase is extended by the additional on-line testing by the Proxy-tester **ioco** testing.

Definition 5 (proxy-tester) *Given a IOTS specification* $S = \langle Q^S, q_0^S, \Sigma_{I,O}^S, T^S \rangle$*, then its Proxy-tester* $Y = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ *is constructed as follows*

- $Q^Y \subseteq \mathcal{P}(Q^S) \cup$ *error-state;*
- $q_0^Y = (q_0^S)$*;*
- $\Sigma_{I,O}^Y = \Sigma_{I,O}^S$*;*
- $T^Y = \{(q, e, q') \mid e \in \Sigma_{I,O}^S : \exists q_1, q_1' \in Q^S : \exists q_1 \in q : \forall\ q_1' \in q' : q_1 \stackrel{e}{\Rightarrow} q_1' \in Q^S\} \cup \{(q, a, \textit{error-state}) \mid \forall a \in \Sigma_O^S : \forall q \in Q^Y : \forall q_1 \in Q^S : q_1 \in q \wedge q_1 \stackrel{a}{\not\Rightarrow}\} \cup \{(q, \delta, \textit{error-state}) \mid \forall q \in Q^Y : \forall q' \in Q^S : q' \in q \wedge q' \stackrel{\delta}{\not\Rightarrow}\} \cup \{(q, \delta, q) \mid \forall q \in Q^Y : \forall q' \in Q^S : q' \in q \wedge q' \stackrel{\delta}{\rightarrow} q'\}$

Figure 5 shows the Proxy-tester for our specifications presented in Figure 2. We can see that this proxy has for all states the possibility to do inputs that are allowed by the specification and it tells us where to go for all outputs, including quiescence.

Since we propose to do passive testing in a proxy style, we do not interfere with the normal behavior of the system and we require to only log the observable information about the behavior of the system to be able to conclude if an unexpected failure occurred, i.e. if an unspecified output occurred.

Note, from the definition of our Proxy-tester it is possible to note that the only events that permit to arrive to the error-state are outputs events. Moreover, this outputs are not permitted in the specification. Those results are presented in the next property.

Property 1 *Given* $S \in IOTS$ *a specification,* $Y_S = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ *its Proxy-tester, then* $\forall\ q \in Q^Y$*:*

- *if* $(q, a, error - state) \in T^Y$ *then:*
  - $a \in \Sigma_O(S)$

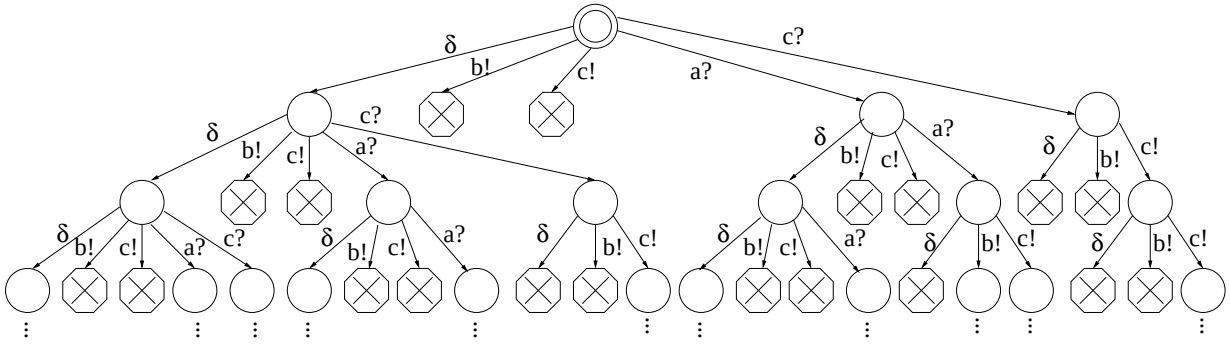

Fig. 4: The Mother Tree of Tests of S, MTT(S)

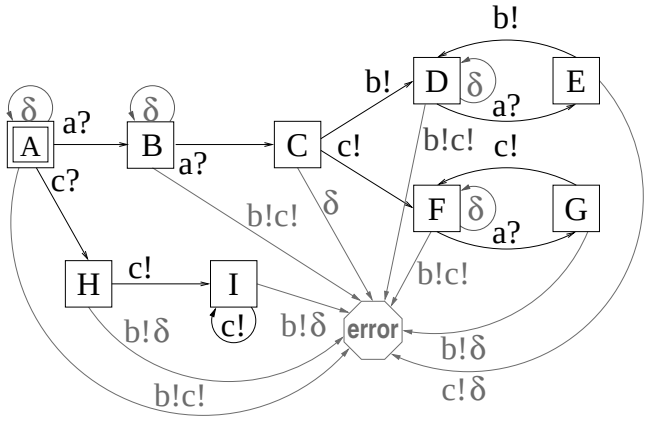

Fig. 5: Proxy-tester

  - $\forall\ q' \in Q(S) : q' \in q : \forall\ \sigma \in \mathsf{traces}(S) : a \notin out(S\ \mathbf{after}\ \sigma)$
- $\forall\ a \in \Sigma_O(S) : q \overset{a}{\rightarrow}$

Using some of this properties, as follows, we can prove Lemma 1.

Lemma 1 *Given $S \in IOTS$ a specification, its Proxy-tester $Y_S = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ and $P \in IOTS$ an input-enabled implementation, then: P* **ioco** *S if and only if*

$$\forall\ \sigma \in \mathsf{traces}(P) : out(P\ \mathbf{after}\ \sigma) \notin out(S\ \mathbf{after}\ \sigma)$$
$$then\ error - state = (Y\ \mathbf{after}\ \sigma)$$

Proof Suppose $\sigma \in \mathsf{traces}(P)$ is one of the trace that because P **ioco** S then $out(P\ \mathbf{after}\ \sigma) \notin out(S\ \mathbf{after}\ \sigma)$. So we can suppose that exists $\sigma' \in \mathsf{traces}(P)$ with $\sigma' \in \mathsf{traces}(S) \wedge \sigma' \in \mathsf{traces}(Y)$ and that exists a output $a \in \Sigma_O^S \cup \{\delta\}$ so that $\sigma' a = \sigma$. Now, because $\sigma' \in \mathsf{traces}(S)$ and $\sigma' a \notin \mathsf{traces}(S)$ we know that for all state $q \in Q(S) : q \in (S\ \mathbf{after}\ \sigma')$ then $q \overset{a}{\not\rightarrow}$. So by Definition 5, we know that $\exists\, q' \in Q(Y)$ and $q' = (S\ \mathbf{after}\ \sigma') \wedge (q', a, error - state) \in T^Y$. Then we can conclude that $error - state = (Y\ \mathbf{after}\ \sigma)$.

Following some ideas from passive testing of [13], we observe the system to be checked without any interaction with it. Our Proxy-tester differs from [13] so that we do not communicate the Proxy-tester with the implementation though inputs and outputs but we considers them as events that synchronize between them. Hence, we present as follows, our composition between the implementation and the Proxy-tester.

A. Composition

The integration of our implementation and our Proxy-tester can be modeled algebraically by putting the components in parallel while synchronizing their common events. Note that here we do not relate inputs with outputs and we do not make them outputs as normally it is done in the literature. Instead we relate the same events and they keep been what they are, inputs or outputs.

This is done in that way so that the existence of this proxy does not interrupt the normal behavior of our system. The synchronization between an implementation P and a Proxy-tester Y is denoted by $P \,||\, Y$.

Definition 6 (composition) *Given P an IOTS implementation $P = \langle Q^P, q_0^P, \Sigma_{I,O}^P, T^P \rangle$ and $Y = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ a Proxy-tester, then their composition $P \,||\, Y = \langle Q^{P||Y}, q_0^{P||Y}, \Sigma_{I,O}^{P||Y}, T^{P||Y} \rangle$ is constructed as follows*

- $Q^{P||Y} = \{q_1 \,||\, q_2 : q_1 \in Q^P \wedge q_2 \in Q^Y\}$;
- $\Sigma_{I,O}^{P||Y} = \Sigma_{I,O}^P = \Sigma_{I,O}^Y$;
- $T^{P||Y}$ *is the minimal set satisfying the following inference rules* $(a \in \Sigma_{I,O,\tau}^{P||Y})$:

$$q_1 \xrightarrow{a} q_1', a \notin \Sigma_{I,O}^Y \quad \vdash \quad q_1 \,||\, q_2 \xrightarrow{a} q_1' \,||\, q_2$$
$$q_2 \xrightarrow{a} q_2', a \notin \Sigma_{I,O}^P \quad \vdash \quad q_1 \,||\, q_2 \xrightarrow{a} q_1 \,||\, q_2'$$
$$q_1 \xrightarrow{a} q_1', q_2 \xrightarrow{a} q_2', a \neq \tau \quad \vdash \quad q_1 \,||\, q_2 \xrightarrow{a} q_1' \,||\, q_2'$$
$$q_1 \xrightarrow{a} q_1', q_2 \xrightarrow{a} q_2', a \neq \tau \quad \vdash \quad q_1 \,||\, q_2 \xrightarrow{a} q_1' \,||\, q_2'$$

Note that, because our Proxy-tester does not have internal transitions, i.e. τ, is output enabled (Property 1) and the implementation is input-enabled then ($q_2 \xrightarrow{a} q_2', a \notin \Sigma_{I,O}^P$) do not happen. Nevertheless, for completeness we defined it.

Lemma 2 *Given $S \in$ IOTS a specification, $Y_S = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ its Proxy-tester and $P \in$ IOTS an input-enabled implementation, then:*

$$\forall\ \sigma \in \mathsf{traces}(Y) : \sigma \in \mathsf{traces}(P \,||\, Y)$$

Proof We prove it by induction in the length of a trace. First for the empty trace it is trivial. So, we suppose for all traces σ of length n, i.e. $|\ \sigma\ | = n$, then if $\sigma \in \mathsf{traces}(Y)$ then $\sigma \in \mathsf{traces}(P \,||\, Y)$. Now, for all trace $\sigma' = \sigma a$ with $\sigma' \in \mathsf{traces}(Y)$ by definition of $\mathsf{traces}(Y)$ we know that $\sigma' \in \mathsf{traces}(S)$. So, I) if $a \in \Sigma_I(S)$, because P is input-enabled we can conclude that $\sigma \in \mathsf{traces}(P \,||\, Y)$, II) if $a \in \Sigma_O(S)$ then: a) if $a \in (P$ **after** $\sigma)$ then we can conclude that $\sigma \in \mathsf{traces}(P \,||\, Y)$, or b) if $a \notin (P$ **after** $\sigma)$ then using the second rule of $T^{P||Y}$ in Definition 6 we can conclude that $\sigma \in \mathsf{traces}(P \,||\, Y)$.

The soundness of our approach is proven in the following Theorem.

Theorem 7 *Given $S \in$ IOTS a specification, $Y_S = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ its Proxy-tester, then $\forall\ P \in$ IOTS an input-enabled implementation, then:*

$$P \ \ \mathbf{ioco}\ S : \exists\, \sigma \in \mathsf{traces}(S) : P \,||\, Y \xrightarrow{\sigma} P' \,||\, \mathit{error-state}$$

Proof This prove is directly using Lemma 1 and Lemma 2.

This theorem tells us that our Proxy-tester is really a sniffer that does not modify the behavior of the implementation and in the same time it can conclude, when it arrives to the error state, when an implementation does not **ioco** conforms with respect to the specification.

VI. Extended-Proxy tester

Our Proxy-tester is a deterministic model of the system and augmented with the information about where to go (to an error-state) when a non-expected output comes. Now, we propose to improve it with a new state that informs us if a malicious user of the implementation is trying to trigger it with input events that where not specified. To do so, we present the *Extended-Proxy*.

Definition 8 (Extended-Proxy) *Given a Proxy-tester $Y = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ of a specification system $S = \langle Q^S, q_0^S, \Sigma_{I,O}^S, T^S \rangle$, the Extended-Proxy $E = \langle Q, q_0, \Sigma_{I,O}, T \rangle$ is constructed as follows:*

- $Q = Q^Y \cup$ *malicious-state;*
- $q_0 = q_0^Y$*;*
- $\Sigma_I = \Sigma_I^Y$ *and* $\Sigma_O = \Sigma_O^Y$*;*
- $T = T^Y \cup \{(q, a,$ *malicious-state*$) \mid \forall\, q \in Q^Y : \forall\, a \in \Sigma_I^S : \forall\, q' \in Q^S : q' \in q \wedge q' \overset{a}{\not\rightarrow}\}$

Figure 6 shows the Extended-Proxy for our running example. The idea of considering this inputs enabledness becomes interesting when we think that the implementation can interact with a malicious user, that is trying to take advantage of the implementation input-enabled property. Finally, we can prove that all traces of our Proxy-tester behave in the same manner in our Extended-Proxy.

Lemma 3 *Given $S \in$ IOTS a specification, $Y_S = \langle Q^Y, q_0^Y, \Sigma_{I,O}^Y, T^Y \rangle$ its Proxy-tester and $E = \langle Q, q_0, \Sigma_{I,O}, T \rangle$ its Extended-Proxy, then:*

$$\forall\ \sigma \in \mathsf{traces}(Y) : \sigma \in \mathsf{traces}(E)$$

Proof Direct by Definition 8.

VII. Conclusions

The testing process tries to detect failures as discrepancies between the actual behavior of an implementation and the intended one described by the specification. In this paper we realize that specifications not only give us information about how an implementation should behave but also about how the environment should supposedly trigger that implementation. Thus, we use this information and combine them in a monitoring engine: the Extended-Proxy tester. Our engine, at the same time, makes checking of the **ioco** testing relation in a passive way and informs us if a malicious user is trying to provoke the implementation with an input that was not specified by the specification. Basically, we are simultaneously identifying **ioco** failures and are aware of malicious intended users.

The approach presented in this paper does not suppose to replace the testing phase in the common development of a software system. Instead, our proposed proxy-tester tries to extend the testing phase as much as possible.

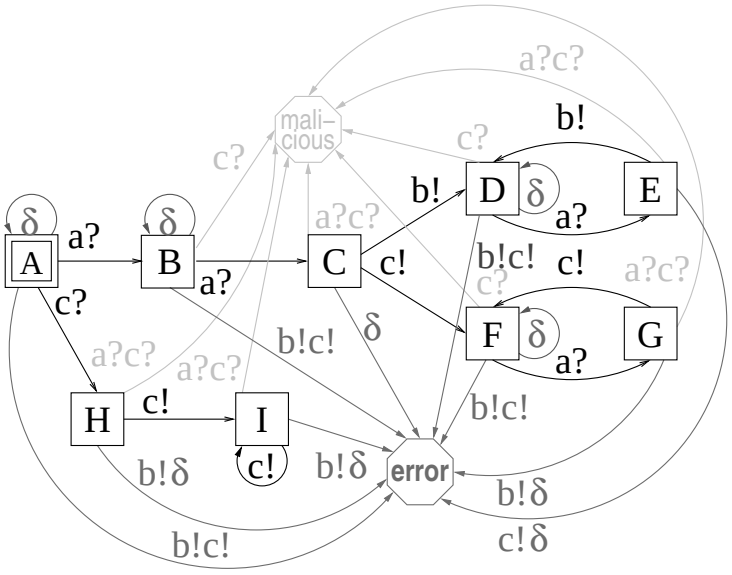

Fig. 6: Proxy-tester + malicious-tester = Extended-Proxy

Also, as it is not normally done in ioco passive testing, our extended-proxy would consider the possible malicious users behaviors, i.e. it will analyze inputs that where not specified.

As future works we have several issues that we would like to address. First, if we consider specification augmented with weight information that tell us about the severity of a failure, then we can immediately enrich our Extended-Proxy and be aware of this information. Second, we would like to analyze communication protocols to study how these ideas can be implemented there in a security environment. Third, we would like to implement the proxy-tester and try it on industrial cases to analyze how good our theories apply to real industry.

REFERENCES

[1] R. D. Nicola and M. C. B. Hennessy, "Testing equivalences for processes," *Theoretical Computer Science*, pp. 83–133, 1984.

[2] S. Salva and I. Rabhi, "Stateful web service robustness," *ICIW '10: Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, vol. 14:424–437, 2010.

[3] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," Enschede, the Netherlands, 1996.

[4] J. Tretmans and E. Brinksma, "Torx: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering*, A. Hartman and K. Dussa-Ziegler, Eds., December 2003, pp. 31–43.

[5] L. Brandán Briones and E. Brinksma, "A test generation framework for quiescent real-time systems," in *IN FATES'04*. Springer-Verlag GmbH, 2004, pp. 64–78.

[6] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, ser. Lecture Notes in Computer Science. Springer Verlag, 2008, vol. 4949, pp. 39–76, the attached file is a preliminary version.

[7] A. Belinfante, L. Frantzen, and C. Schallhart, "Tools for test case generation," *In Model-Based Testing of Reactive Systems*, p. 391–438, 2004.

[8] J. Claude and J. Thierry, "Tgv: Theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 297–315, Aug. 2005.

[9] L. Brandán Briones, E. Brinksma, and M. Stoelinga, "A semantic framework for test coverage," in *Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science 4218, 2006, pp. 399–414.

[10] S. Sébastien and T.-D. Cao, "A model-based testing approach combining passive testing and runtime verification. application to web service composition testing in clouds," in *Software Engineering Research, Management and Applications*, 2014, pp. 99–116.

[11] R. H. R. E. M. J. W. David Lee, Dongluo Chen and X. Yin, "Network protocol system monitoring: a formal approach with passive testing," *IEEE/ACM Trans. Netw.*, vol. 14:424–437, 2006.

[12] S. M. Tao Lin and J. Park, "A framework for wireless ad hoc routing protocols," *WCNC 2003, New Orleans, LA, USA, IEEE society press*, 2006.

[13] S. Salva, "Passive Testing with Proxy-testers," in *International Journal of Software Engineering and Its Applications*, 2011.

[14] M. Katara and S. Katz, "A concern architecture view for aspect-oriented software design," *Software and Systems Modelling Journal (SoSyM), vol. 6*, pp. 247–265, 2007.

[15] T. Aaltonen, J. Helin, M. Katara, P. Kellom´aki, and T. Mikkonen, "Coordinating aspects and objects," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 3, pp. 248–267, 2003, foclasa 2002, Foundations of Coordination Languages and Software Architectures (Satellite Workshop of CONCUR 2002).