Automatically Checking Conformance on Asynchronous Reactive Systems

Camila Sonoda Gomes
Computing Department
UEL - State University of Londrina
Londrina, Brazil
Email: camilasonoda@uel.br

Adilson Luiz Bonifacio
Computing Department
UEL - State University of Londrina
Londrina, Brazil
Email: bonifacio@uel.br

Abstract—**Software testing is an important issue in the software development process to ensure the quality of products. Formal methods have been promising on testing reactive systems, where accuracy is mandatory and any fault can cause severe damage. Systems of this nature are characterized by receiving messages from the environment and producing outputs in response. One of the biggest challenges in model-based testing is the conformance checking of asynchronous reactive systems. The aim is to verify if an implementation complies with its respective specification. In this work, we study conformance checking for reactive systems specified by Input Output Labeled Transition Systems (IOLTS). We develop a practical tool to check the conformance relation between reactive models using the classical ioco relation and a more general theory based on regular languages. In addition, we present some testing scenarios in practical applications and compare them to other tools from the literature using both notions of conformance.**

Keywords–model-based testing; conformance testing; automatic verification; reactive systems.

I. INTRODUCTION

Automatic testing tools have been proposed to support the development process of reactive systems that are characterized by continuous interaction with the environment. In this setting, systems receive external stimuli and produce outputs, asynchronously, in response. In addition, systems of this nature are usually critical and require more accuracy in their development process, especially in the testing activity, where appropriate formalisms must be used as the basis [1]–[3]. IOLTSs [2]–[5] are traditional formalisms usually applied to model and test reactive systems.

In model-based testing, an IOLTS specification can model desirable and undesirable behaviors of an Implementation Under Test (IUT). The aim is to find faults in an IUT according to a certain fault model [1] [6] in order to show if requirements are satisfied regarding its respective system specification. The well-established **ioco** conformance relation [3] requires that outputs produced by an IUT should also be produced by its respective specification. A more recent and general conformance relation [1] specifies desirable and undesirable behaviors using regular languages to define the testing fault model.

In this work, we address the development of an automatic tool for conformance verification of asynchronous reactive systems modeled by IOLTSs. We introduce both notions of conformance in our practical tool to provide a wider application range compare to other tools. JTorx [7], for instance, is a tool from the literature that also implements a conformance testing verification process, but only based on the classical **ioco** relation. Our tool comprises both the classical **ioco** relation and also the more general conformance based on regular languages. We also run some practical scenarios to evaluate aspects related to the effectiveness and usability of both conformance theories and these tools. We show scenarios where the language-based conformance is able to find faults which cannot be detected by the classical **ioco** conformance.

We organize this paper as follows. Section II describes the conformance verification methods using regular languages and the **ioco** relation. The practical tool which implements the more general method of conformance checking is presented in Section III. Some applications and a comparative study are given in Section IV. Section V describes the comparative analysis of tools. Section VI offers some concluding remarks.

II. CONFORMANCE VERIFICATION

The conformance checking task can determine if an IUT complies with its specification when both are modeled by appropriate formalisms. The classical **ioco** conformance relation [3] [5] establishes the compliance between IUTs and specifications when they are specified by IOLTS [2]–[5]. An IOLTS is a variation of the Labeled Transition Systems (LTS) [8]–[11] with the partitioning of input and output labels.

Definition 1: An IOLTS $\mathcal{S}$ is given by (S, s_0, L_I, L_U, T) where: S is the set of states; $s_0 \in S$ is the initial state; L_I is a set of input labels; L_U is a set of output labels; $L = L_I \cup L_U$ and $L_I \cap L_U = \emptyset$; $T \subseteq S \times (L \cup \{\tau\}) \times S$ is a finite set of transitions, where the internal action $\tau \notin L$; and (S, s_0, L, T) is the underlying LTS associated with $\mathcal{S}$.

A transition $(s, l, s') \in T$ indicates that from the state $s \in S$ with the label $l \in (L \cup \{\tau\})$ the state $s' \in S$ is reached in an LTS/IOLTS model. When we have a transition $(s, \tau, s') \in T$ with an internal action, it means that an external observer can not see the movement from state s to state s' in the model.

We may also have the notion of quiescent states. If a state s of an IOLTS has no output $x \in L_U$ and internal action τ defined on it, we say that s is quiescent [3]. When a state s is quiescent we then add a transition (s, δ, s), where $\delta \notin L_\tau$. Note that we denote $L \cup \{\tau\}$ by L_τ in order to ease the notation.

In a real scenario of black-box testing where an IUT sends messages to a tester and receives back responses, quiescence indicates that an IUT could no longer respond to the tester, or it has timed out, or even it is simply slow.

We also need to define the semantics of LTS/IOLTS models. But first, we introduce the notion of paths.

 ISBN: 978-1-61208-752-8

Definition 2: ([1]) Let $\mathcal{S} = (S, s_0, L, T)$ be a LTS and $p, q \in S$. Let $\sigma = l_1, \cdots, l_n$ be a word in $L^\star_\tau$. We say that σ is a *path* from p to q in $\mathcal{S}$ if there are states $r_i \in S$, and labels $l_i \in L_\tau$, $1 \leq i \leq n$, such that $(r_{i-1}, l_i, r_i) \in T$, with $r_0 = p$ and $r_n = q$. We say that α is an *observable path* from p to q in $\mathcal{S}$ if we remove the internal actions τ from σ.

A path can also be denoted by $s \xrightarrow{\sigma} s'$, where the behavior $\sigma \in L^\star_\tau$ starts in the state $s \in S$ and reaches the state $s' \in S$. An observable path σ, from s to s', is denoted by $s \overset{\sigma}{\Rightarrow} s'$. We can also write $s \xrightarrow{\sigma}$ or $s \overset{\sigma}{\Rightarrow}$ when the reached state is not important.

Paths that start from state s are called paths of s, and the semantics of an LTS model is given by the paths that start from their initial state. Now we give the semantics of LTS models.

Definition 3: ([1]). Let $\mathcal{S} = (S, s_0, L, T)$ be a LTS and $s \in S$: (1) The set of paths of s is given by $tr(s) = \{\sigma | s \xrightarrow{\sigma}\}$ and the set of observable paths of s is $otr(s) = \{\sigma | s \overset{\sigma}{\Rightarrow}\}$. (2) The semantics of $\mathcal{S}$ is $tr(s_0)$ or $tr(\mathcal{S})$ and the observable semantics of $\mathcal{S}$ is $otr(s_0)$ or $otr(\mathcal{S})$.

The semantics of an IOLTS is defined by the semantics of the underlying LTS.

Conformance checking can be established between IOLTS models over the **ioco** relation. When we apply input stimuli to both a specification and an IUT, if the IUT produces outputs that are also defined in the specification, we say that the IUT conforms to the specification. Otherwise, we say that they do not conform [3].

Definition 4: ([3]). Let $\mathcal{S} = (S, s_0, L_I, L_U, T)$ be a specification and $\mathcal{I} = (Q, q_0, L_I, L_U, R)$ be an IUT. We say that $\mathcal{I}$ *ioco* $\mathcal{S}$ if, and only if, $out(q_0\ after\ \sigma) \subseteq out(s_0\ after\ \sigma)$ for all $\sigma \in otr(\mathcal{S})$, where $s\ after\ \sigma = \{q | s \overset{\sigma}{\Rightarrow} q\}$ for all $s \in S$ and all $\sigma \in otr(\mathcal{S})$, and the function $out(V) = \bigcup_{s \in V} \{l \in L_U | s \overset{l}{\Rightarrow}\}$.

The more general conformance relation is established over regular languages. This approach provides a wider fault coverage for both LTS and IOLTS models. Basically, desirable and undesirable behaviors are specified by regular languages, D and F, respectively. Given an implementation $\mathcal{I}$, a specification $\mathcal{S}$, and regular languages D and F, $\mathcal{I}$ complies with $\mathcal{S}$ according (D, F), i.e, $\mathcal{I}\ conf_{D,F}\ \mathcal{S}$ if, and only if, no undesirable behavior of F is observed in $\mathcal{I}$ and is specified in $\mathcal{S}$, and all desirable behaviors of D are observed in $\mathcal{I}$ and also are specified in $\mathcal{S}$.

Definition 5: ([1]) Let a set of symbols L, and the languages $\mathcal{D}, \mathcal{F} \subseteq L^\star$ over L. Let $\mathcal{S}$ and $\mathcal{I}$, LTS models, with L as their set of labels, $\mathcal{I} conf_{D,F}\ \mathcal{S}$ if, and only if: (1) $\sigma \in otr(\mathcal{I}) \cap F$, then $\sigma \notin otr(\mathcal{S})$; and (2) $\sigma \in otr(\mathcal{I}) \cap D$, then $\sigma \in otr(\mathcal{S})$.

We remark that an ordinary LTS can be checked using the language-based approach using only the notion of desirable and undesirable behaviors. In this case, we do not need to partition the alphabet into input and output labels, as required by IOLTS models and crucial for **ioco** relation. The next proposition states the language-based conformance checking.

Proposition 1: ([1]). Let the specification $\mathcal{S}$ and the IUT $\mathcal{I}$ be LTS models over L, and the languages $D, F \subseteq L^\star$ over L. We say that $\mathcal{I}\ conf_{D,F}\ \mathcal{S}$ if, and only if, $otr(\mathcal{I}) \cap [(D \cap \overline{otr}(\mathcal{S})) \cap (F \cap otr(\mathcal{S}))] = \emptyset$, where $\overline{otr}(\mathcal{S})$ is the complement of $otr(\mathcal{S})$ given by $\overline{otr}(\mathcal{S}) = L^\star - otr(\mathcal{S})$.

On the other hand, the next lemma shows that the more general notion of conformance relation given in Definition 5 restrains the classical **ioco** conformance relation.

Lemma 1: ([1]). Let a specification $\mathcal{S} = (S, s_0, L_I, L_U, T)$ and an IUT $\mathcal{I} = (Q, q_0, L_I, L_U, R)$ be IOLTS models, we have that $\mathcal{I}\ ioco\ \mathcal{S}$ if, and only if, $\mathcal{I}\ conf_{D,F}\ \mathcal{S}$ when $D = otr(\mathcal{S})L_U$ e $F = \emptyset$.

Clearly, the **ioco** relation can be given by the more general conformance relation using regular languages.

The conformance checking can be obtained using the automata theory [12] as proposed by Bonifacio and Moura [1]. We transform LTS/IOLTS models into Finite State Automatons (FSAs) and apply union, intersection, and complement operations over regular languages. An FSA is formally given by $\mathcal{A} = (S, s_0, L, T, F)$, where $\mathcal{S} = (S, s_0, L, T)$ is the underlying LTS associated with $\mathcal{A}$. Note that the set of final states in $\mathcal{A}$ is defined by all states of $\mathcal{S}$, i.e., $F = S$. Since the semantics of an FSA is given by the language it accepts, a language $R \subseteq L^\star$ is *regular* if there is an FSA $\mathcal{M}$ such that $L(\mathcal{M}) = R$, where L is an alphabet [12]. Hence, we can effectively construct the automatons $\mathcal{A}_D$ and $\mathcal{A}_F$ such that D and F are regular languages and $D = L(\mathcal{A}_D)$ and $F = L(\mathcal{A}_F)$.

The notions of the test case and test suite according to formal languages are given as follows.

Definition 6: ([1]). Let a set of symbols L, the test suite T over L is a language, where $T \subseteq L^\star$, so that each $\sigma \in T$ is a test case.

If the test suite is a regular language, then there is an FSA $\mathcal{A}$ that accepts it, such that the final states of $\mathcal{A}$ are fault states. The set of undesirable behaviors, defined by these fault states, is called by fault model of $\mathcal{S}$ [1].

A complete test suite can be obtained from an IOLTS specification $\mathcal{S}$ and a pair of languages (D, F) using the Proposition 1. The test suite $T = [(D \cap \overline{otr}(\mathcal{S})) \cup (F \cap otr(\mathcal{S}))]$ is able to identify the absence of desirable behaviors specified by D and the presence of undesirable behaviors specified by F in the specification $\mathcal{S}$. We declare that an implementation $\mathcal{I}$ complies with a specification $\mathcal{S}$ if there is no test case of the test suite T that is also a behavior of $\mathcal{I}$ [1].

We also provide the determinization of models which is useful in this method. Therefore, from a deterministic IOLTS $\mathcal{S}$ we can obtain the automaton $\mathcal{A}_1$ induced by $\mathcal{S}$ that is also deterministic. We write $L(\mathcal{A}_1) = otr(\mathcal{S})$. Hence, we can effectively obtain an FSA $\mathcal{A}_2$ such that $L(\mathcal{A}_2) = L(\mathcal{A}_F) \cap L(\mathcal{A}_1) = F \cap otr(S)$. Also, consider the FSA $\mathcal{B}_1$ obtained from $\mathcal{A}_1$ by reversing its set of final states, that is, a state s is a final state in $\mathcal{B}_1$ if, and only if, s is not a final state in $\mathcal{A}_1$. Clearly, $L(\mathcal{B}_1) = \overline{L(\mathcal{A}_1)} = \overline{otr}(\mathcal{S})$. We can now effectively get an FSA $\mathcal{B}_2$ such that $L(\mathcal{B}_2) = L(\mathcal{A}_D) \cap L(\mathcal{B}_1) = D \cap \overline{otr}(\mathcal{S})$. Since $\mathcal{A}_2$ and $\mathcal{B}_2$ are FSAs, we can construct an FSA $\mathcal{C}$ such that $L(\mathcal{C}) = L(\mathcal{A}_2) \cup L(\mathcal{B}_2)$, where $L(\mathcal{C}) = T$. We can conclude that when D and F are regular languages and $\mathcal{S}$ is a deterministic specification, then a complete FSA $\mathcal{T}$ can be constructed such that $L(\mathcal{T}) = T$.

Next proposition states an algorithm with a polynomial time complexity for the language-based verification.

Proposition 2: ([1]) Let $\mathcal{S}$ and $\mathcal{I}$ be the deterministic specification and implementation IOLTSs over L with n_S and n_I states, respectively. Let also $|L| = n_L$. Let $\mathcal{A}_D$ and $\mathcal{A}_F$ be deterministic FSAs over L with n_D and n_F states, respectively, and such that $L(\mathcal{A}_D) = D$ and $L(\mathcal{A}_F) = F$. Then, we can effectively construct a complete FSA $\mathcal{T}$ with $(n_S + 1)^2 n_D n_F$ states, and such that $L(\mathcal{T})$ is a complete test suite for $\mathcal{S}$ and (D, F). Moreover, there is an algorithm, with polynomial time complexity $\Theta(n_S^2 n_I n_D n_F n_L)$ that effectively checks whether $\mathcal{I} conf_{D,F} \mathcal{S}$ holds.

Next, we obtain a similar result for **ioco** using Lemma 1.

Theorem 1: ([1]) Let $\mathcal{S}$ and $\mathcal{I}$ be deterministic specification and implementation IOLTSs over L with n_S and n_I states, respectively. Let $L = L_I \cup L_U$, and $|L| = n_L$. Then, we can effectively construct an algorithm with polynomial time complexity $\Theta(n_S n_I n_L)$ that checks whether $\mathcal{I}$ **ioco** $\mathcal{S}$ holds.

III. A Testing Tool for Reactive Systems

In this section, we present the automatic checking conformance tool *Everest (conformancE Verification on tEsting ReactivE SysTems)* [13]. Our tool supports the more general notion of conformance based on regular languages and also the classical **ioco** relation when testing reactive systems modeled by LTS/IOLTS. Everest has been developed in Java [14] using the *Swing* library [15], providing a yielding and friendly usability experience through a graphical interface.

Some features provided by the Everest tool are: (i) check conformance based on regular languages and **ioco** relation; (ii) describe desirable and undesirable behaviors using regular expressions; (iii) specify formal models in Aldebaran format [16]; (iv) generate test suites when non-conformance verdicts are obtained; (v) provide state paths, i.e, the sequence of states induced by a test case over the IUT and specification; and (vi) allow the graphical representation of the models.

The tool's architecture is organized into four modules as depicted in Figure 1. The modules are given by rectangles and the data flow between them is denoted by the arrows. The input data and the output results are represented by ellipses.

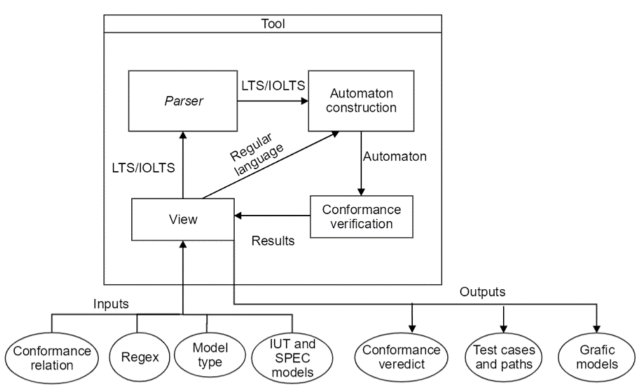


Figure 1. Tool's Architecture

The *View* module implements an intuitive graphical interface with three different views: configuration; **ioco** conformance; and language-based conformance.

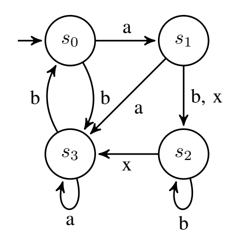
The *Parser* module reads the input data with the descriptions of IUT and specification, and turn them into data structures to internally represent their respective models. The *Automaton Construction* module transforms the LTS/IOLTS models into their respective finite automatons which, in turn, are used to construct the fault model together with the automatons obtained by means of regular languages.

The *Conformance Verification* module provides all necessary operations over regular languages such as union, intersection, and complement [12]. This module also constructs the finite automaton that represents the complete test suite and comprises both conformance verification techniques. The conformance checking processes and their essential algorithms are described in [17].

Everest defines a standard representation of LTS/IOLTS models over the Aldebaran [18] format as a set of transitions. Figure 2a presents an example of Aldebaran format and Figure 2b shows its respective IOLTS model. The header $des(s0, 9, 4)$ indicates the initial state, the number of transitions and the number of states. The set of transitions follows the header line by line, where each transition (s, a, q) is defined such that s is the source state, a is the label associated to the transition and q is the target state. Input and output labels can be indicated by the special markers "?" and "!", respectively. But we remark that the special markers just ease the graphical visualization. Everest constructs, internally, a list of input and output labels even if these labels have the special markers or they are settled by the sets of input and output, L_I and L_U, respectively (Figure 2b).

```
des (s0,9,4)
(s0,?a,s1)
(s0,?b,s3)
(s1,?b,s2)
(s1,!x,s2)
(s1,?a,s3)
(s2,?b,s2)
(s2,!x,s3)
(s3,?b,s0)
(s3,?a,s3)
```

(a) $\mathcal{S}$ in Aldebaran format



(b) IOLTS specification $\mathcal{S}$

Figure 2. An example of Aldebaran file format

In Figure 3, we can observe the configuration view, where specification and the implementation models are selected, in the Aldebaran format. When the model type is an LTS, the parameters $Label$, $Input\ labels$, and $Output\ labels$ are omitted. If IOLTS models are given then we need to inform how the input/output labels are distinguished (field $Label$), informing below the $Input$ and $Output$ labels or the special markers are assumed in the Aldebaran files, as in Figure 3.

Figure 4 presents the interface for **ioco** conformance verification. Note that in both conformance verification views all information from the configuration view remains visible to ease the reference. Also, the buttons $view\ model$ and $view\ IUT$ allow the graphical visualization of the implementation and specification models. The verdict is displayed by clicking the button $Verify$. In case of non-conformance, the tool presents a set of paths induced by the test suite that detects the faults. The tool also informs incorrectly filled fields in the configuration view in the text box $Warnings$.

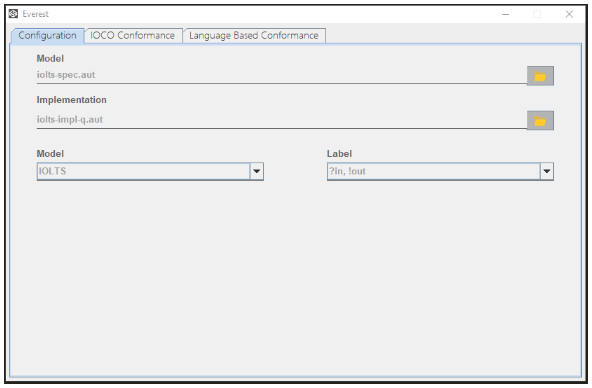


Figure 3. Configuration view

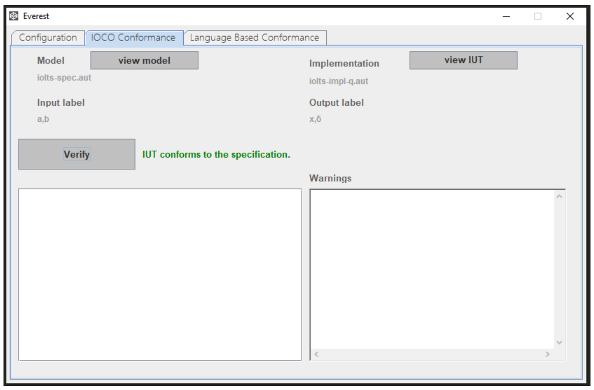


Figure 4. **ioco** verification view

In Figure 5, we present the language-based conformance verification view. *Desirable* and *Undesirable* behaviors must

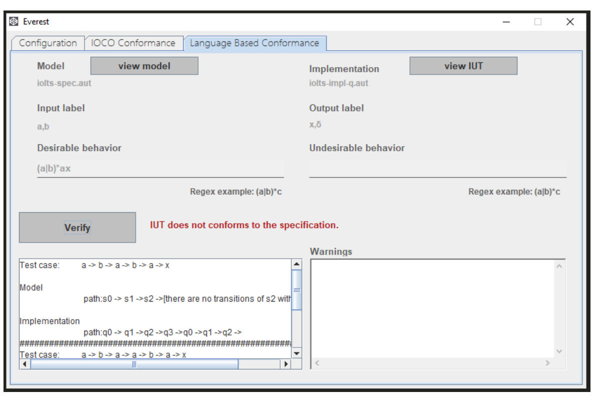
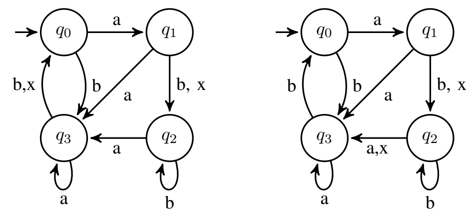


Figure 5. Language-based verification view

be specified by regular expressions. When no regular expression is provided the Kleene closure [12] is assumed over the alphabet to identify faults when models are not isomorphic. After the compliance check, the verdict is displayed similarly to the **ioco** verification conformance.

IV. Practical Application

This section describes some practical testing scenarios applied to the Everest and JTorx tools. Let $\mathcal{S}$ be the IOLTS specification of Figure 2b and let $\mathcal{R}$ and $\mathcal{Q}$ be implementations candidates as depicted in Figures 6a and 6b. Also, let $L_I = \{a, b\}$ and $L_U = \{x\}$ be the input and output alphabets, respectively. All models here are deterministic [19] [20] but we remark that our tool also deals with non-deterministic models.



(a) Implementation $\mathcal{R}$ (b) Implementation $\mathcal{Q}$

Figure 6. IOLTS Models

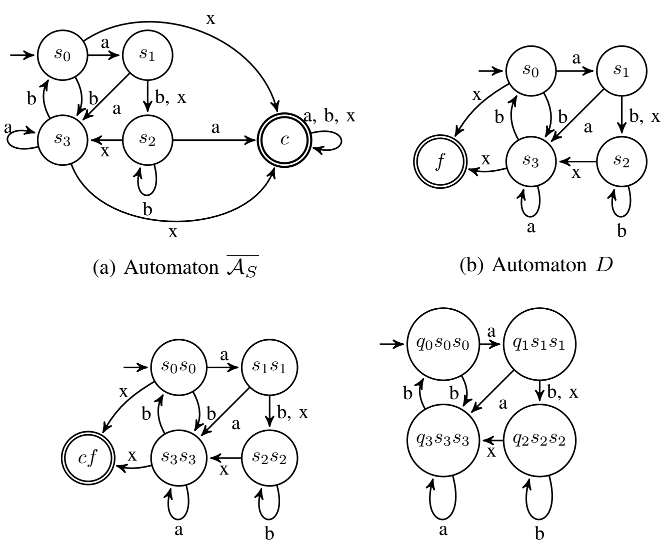
In the first scenario, we check if IUT $\mathcal{R}$ conforms to specification $\mathcal{S}$. Everest tool has returned a non-conformance verdict using **ioco** relation and generated the test suite $\{b, aa, ba, aaa, ab, ax, abb, axb\}$. The subset of test cases $\{b, aa, ba, aaa\}$ induces state paths from s_0 to s_3 in $\mathcal{S}$ and from q_0 to q_3 in $\mathcal{R}$, where the output x is produced by $\mathcal{R}$ but $\mathcal{S}$ does not. Note that s_3 in $\mathcal{S}$ is a quiescent state whence no output is defined on it. The subset $\{ab, ax, abb, axb\}$ induces state paths to state s_2 in $\mathcal{S}$ and q_2 in $\mathcal{R}$. In this case, the output δ is produced by IUT $\mathcal{R}$ whereas $\mathcal{S}$ produces x. That is, a fault is detected according to **ioco** relation. Note that both tools modify the formal models by adding *self-loops* labeled with δ [21] on quiescent states.

The same scenario has been also applied to JTorx tool, resulting in the same verdict, as expected, but it generates the test suite $\{b, ax, ab\}$. Notice that the test suite generated by JTorx is a subset of the test suite generated by Everest. That is, Everest shows all test cases and associated state paths related to each fault according to a transition cover criteria over the specification, differently from JTorx which does not apply transition coverage to test suite derivation. Everest also allows state coverage as criteria to obtain the test suite using only one path per fault when checking conformance over an IUT. But, in this case, we reduce not only the number of test cases, but also the information that might be useful to aid the tester in the fault mitigation process.

In the second scenario (Figure 5), when checking the IUT $\mathcal{Q}$ against the specification $\mathcal{S}$, the language-based conformance verification was able to detect a fault that was not detected by the **ioco** conformance relation (Figure 4). We have obtained the fault model using the regular expressions $D = (a|b)^*ax$ and $F = \emptyset$. Language D clearly expresses behaviors that finish with a stimulus a followed by an output x produced in response. Since the only complete test suite is given by $[(D \cap \overline{otr}(\mathcal{S})) \cup (F \cap otr(\mathcal{S}))]$ and $F \cap otr(\mathcal{S}) = \emptyset$, so we check the condition $D \cap \overline{otr}(\mathcal{S}) \neq \emptyset$, i.e., a fault is detected when behaviors of D are not present in $\mathcal{S}$. Everest then results in a verdict of non-conformance and produces the test suite $\{ababax, abaabax\}$ reaching a fault that is not detected by JTorx using the **ioco** relation.

The specification $\mathcal{S}$ (Figure 2b) and the candidate implementation $\mathcal{Q}$ (Figure 6b) are IOLTS models which, after being converted into underlying automatons respectively, $\mathcal{A}_S$ and $\mathcal{A}_Q$, have all their states defined as final states. Figure 7a displays the complement automaton of the specification.

The **ioco** conformance verification first obtains the underlying automata, $\mathcal{A}_S$ and $\mathcal{A}_Q$, from the IOLTS models. The automaton D (Figure 7b) is constructed to obtain the fault model (Figure 7c) by the intersection of the language $otr(S)L_u$, which is captured by D, and complement language of the specification (Figure 7a). The automaton that represents the test suite (Figure 7d) is obtained by the intersection between the fault model and the $\mathcal{A}_Q$. Since the resulting automaton has no final state, the verdict between models is that $\mathcal{I}$ **ioco** conforms to $\mathcal{S}$.



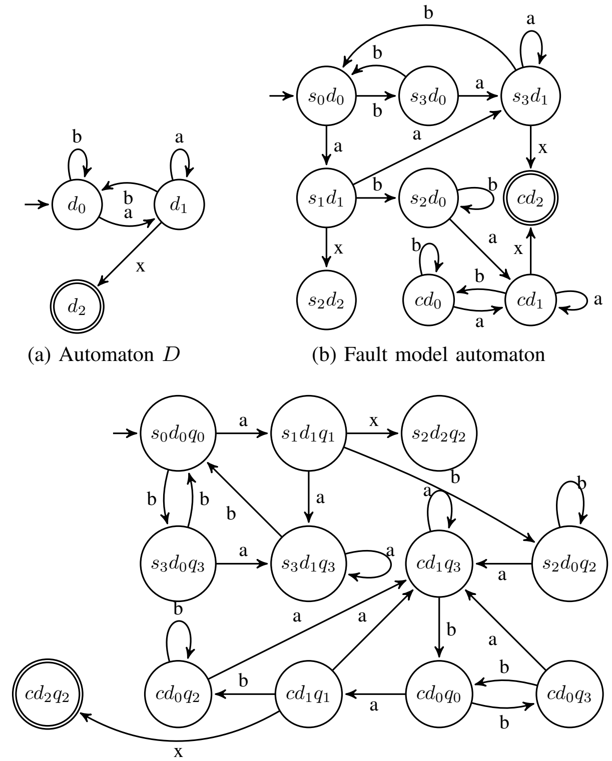
(a) Automaton $\overline{\mathcal{A}_S}$ (b) Automaton D

(c) Fault model automaton (d) Test suite automaton

Figure 7. Automatons: **ioco** conformance verification

In language-based conformance verification, the underlying automatons, $\mathcal{A}_S$ and $\mathcal{A}_Q$, are also obtained from the IOLTS models depicted in Figures 2b and 6b, respectively. From the regular expression $(a|b)^*ax$ we obtain the automaton (Figure 8a) that accepts the respective language. Since the fault model is given by $[D \cap \overline{otr}(\mathcal{S})] \cup [(F \cap otr(\mathcal{S}))]$ and no undesirable behavior F is defined, then $F \cap otr(\mathcal{S}) = \emptyset$, and the fault behaviors are reduced to $D \cap \overline{otr}(\mathcal{S})$. The automaton that represents the fault model is illustrated in Figure 8b. The automaton that represents the test suite is illustrated in Figure 8c. Note that this automaton contains a final state, indicating that the words accepted by the automaton are part of the test suite that reveals the faults and, consequently, the non-conformity between the models. The test suite generated by the Everest tool is $\{ababax, abaabax\}$.

Also, we have performed a practical study over a simple version, but a real scenario, of a vending machine. The IOLTS specification $\mathcal{N}$ of the vending machine is depicted in Figure 9a. Now consider an IUT $\mathcal{P}$ of this vending machine as given in Figure 9b. The input alphabet is given by $L_I = \{1, 3, 5\}$ which means input stimuli are received from the environment. In this case, labels $1, 3, 5$ represent coins provided by users according to the desired drinks. On the other

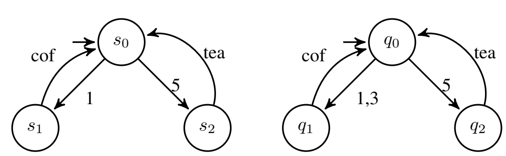


(a) Automaton D (b) Fault model automaton

(c) Test suite automaton

Figure 8. Automatons: language-based conformance verification

hand, the output alphabet is defined by $L_U = \{cof, tea\}$, that is, the vending machine gives back to the user the requested drink, a coffee or a tea.



(a) Specification $\mathcal{N}$ (b) IUT $\mathcal{P}$

Figure 9. Vending machine

When checking whether $\mathcal{N}$ conforms to $\mathcal{P}$ using the language-based method, Everest was able to detect faults that were not detected by JTorx. The desirable behaviors are expressed by $D = 3cof$ and the undesirable behaviors are specified by $F = 1(cof|tea)$. The former expression says that after a user gives the coin 3 the vending machine is supposed to give back a coffee. Similarly, the undesirable expression establishes that after a user gives the coin 1 the vending machine should return neither a coffee nor a tea.

Everest then results in a non-conformance verdict between $\mathcal{N}$ and $\mathcal{P}$, producing the test suite $\{1cof, 3cof\}$. The test case $1cof$ is generated because $\mathcal{N}$ and $\mathcal{P}$ specify that after a coin 1 is provided by the user the vending machine must return a cof which, in turn, is undesirable according to F. The test case $3cof$ reaches a fault because the desirable behavior requires the vending machine gives back a coffee if a coin 3 is inserted into the vending machine. However, this property is not specified on $\mathcal{N}$ and the IUT $\mathcal{P}$ allows such situation.

V. Comparative Analysis of Tools

In this section, we perform a comparative analysis between Everest and JTorx. Both tools implement the conformance verification between a specification and an IUT based on **ioco** theory [3], but only Everest implements the more general conformance relation based on regular languages. Table I summarizes the comparative analysis.

TABLE I. COMPARATIVE ANALYSIS

	JTorx	Everest
Conformance verification		
ioco	X	X
Language-based	-	X
Restrictions over the models		
Support underspecified models	X	X
Require *input-enabledness*	X*	-
Support quiescence	X	X

* JTorx adds self-loops with input labels

The classical **ioco** relation imposes some restrictions and properties over the models, for instance, underspecified models [22] are not allowed on IUT models. In contrast, the language-based conformance relation can deal with specification and implementation models that are not *input-enabled*. Everest implements the language-based conformance relation and also the classical **ioco** relation, which is reduced from the former. We remark that both relations developed in Everest do not require any of these restrictions over the formal models (See Lemma 1). But JTorx requires, for instance, the input enabledness over the IUT models. To overcome the problem of underspecified models, JTorx adds *self-loops* with input labels that are not defined in the states. However, such changes can result in unreliable verdicts since transitions are added to the model, modifying its original behavior. Everest treats underspecified models with no change and keeping the reliability over the original behavior of the models.

Another important issue over underspecified models is quiescence. In this case, Jtorx and Everest add self-loops labeled by δ on states that no output is specified for both specification and implementation models.

VI. Conclusion

Testing of reactive systems is an important and complex activity in the development process for systems of this nature. The complexity of such systems and, consequently, the complexity of the testing task requires high costs and resources in software development. Therefore, automation and accuracy on the testing activity have become essential in this process. Several studies have addressed the testing of reactive systems [23] [24] using model-based testing. More precisely, many works have focused on the conformance checking [1]–[3] between IUTs and specifications to guarantee more reliability.

In this work, we have developed an automatic tool for checking conformance on asynchronous reactive systems. We have implemented not only the classical **ioco** theory but also the more general language-based relation for checking conformance between IOLTS models. We observe by the practical applications that Everest could find faults using the language-based conformance verification process which was not detected by JTorx using the classical **ioco** relation. Everest then gives us an advantage with a wider range of testing scenarios and a full fault detection coverage according to a defined fault model.

There are several tools from the literature that implement conformance checking based on **ioco** relation and its variations [18] [20] [22] [25]–[28]. But, we are not aware of any other tool that implements a different conformance notion, such as the language-based relation. So, the main contribution of this work is the design of Everest tool and its more flexible conformance checking, in addition to its algorithms, the intuitive graphical interface, the practical applications and comparative studies.

A new module of Everest tool is already being developed to provide the test suite generation in a black-box setting. We also intend to perform more experiments using real-world problems with Everest and similar tools from the literature. In this way, we may give a more precise analysis regarding conformance checking for asynchronous reactive models, usability, and performance of these tools.

References

[1] A. L. Bonifácio and A. V. Moura, "Complete test suites for input/output systems," CoRR, vol. abs/1902.10278, 2019, accessed on: 2019-06. [Online]. Available: http://arxiv.org/abs/1902.10278

[2] A. da Silva Simão and A. Petrenko, "Generating complete and finite test suite for ioco: Is it possible?" in Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014., 2014, pp. 56–70, accessed on: 2019-07. [Online]. Available: https://doi.org/10.4204/EPTCS.141.5

[3] J. Tretmans, Model Based Testing with Labelled Transition Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38, accessed on: 2019-07. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_1

[4] B. K. Aichernig and M. Tappler, "Symbolic input-output conformance checking for model-based mutation testing," Electronic Notes in Theoretical Computer Science, vol. 320, 2016, pp. 3 – 19, accessed on: 2019-06. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066116000037

[5] J. Tretmans, "Testing concurrent systems: A formal approach," in CONCUR'99 Concurrency Theory, J. C. M. Baeten and S. Mauw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 46–65.

[6] B. K. Aichernig, M. Weiglhofer, and F. Wotawa, "Improving fault-based conformance testing," Electronic Notes in Theoretical Computer Science, vol. 220, no. 1, 2008, pp. 63 – 77, proceedings of the Fourth Workshop on Model Based Testing (MBT 2008). Accessed on: 2019-08. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157106610800443X

[7] "JTorX a tool for model-based testing," https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/, accessed on: 2018-06.

[8] G. Tretmans, "A formal approach to conformance testing," Ph.D. dissertation, University of Twente, 1992.

[9] P. Daca, T. A. Henzinger, W. Krenn, and D. Nickovic, "Compositional specifications for ioco testing," in 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, March 2014, pp. 373–382.

[10] F. Zeng, Z. Chen, Q. Cao, and L. Mao, "Research on method of object-oriented test cases generation based on uml and lts," in 2009 First International Conference on Information Science and Engineering, Dec 2009, pp. 5055–5058.

[11] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, "Test case generation by means of uml sequence diagrams and labeled transition systems," in 2007 IEEE International Conference on Systems, Man and Cybernetics, Oct 2007, pp. 1292–1297.

ISBN: 978-1-61208-752-8

[12] M. Sipser, Introduction to the Theory of Computation, 2nd ed. Course Technology, 2006.

[13] C. Sonoda, “Everest website,” https://everest-tool.github.io/everest-site, accessed on: 2019-07.

[14] Oracle, “Java se development kit 8,” http://www.oracle.com/technetwork/pt/java/javase/, accessed on: 2019-07.

[15] ——, “Package javax swing,” https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html, accessed on: 2019-06.

[16] “AUT manual page,” https://cadp.inria.fr/man/aut.html, accessed on: 2019-08.

[17] C. S. Gomes and A. L. Bonifácio, “Automatically checking conformance on asynchronous reactive systems,” CoRR, vol. abs/1905.08914, 2019, accessed on: 2019-08. [Online]. Available: http://arxiv.org/abs/1905.08914

[18] J. Calamé, “Specification-based test generation with tgv,” Software Engineering Notes, 2005.

[19] J. E. Hopcroft, R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation (3rd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[20] B. L. Mark Utting, practical model-based testing a tools approach, 1st ed. Elsevier, 2007.

[21] G. Tretmans, Test Generation with Inputs, Outputs and Repetitive Quiescence, ser. CTIT technical report series. Netherlands: Centre for Telematics and Information Technology (CTIT), 1996, no. TR-CTIT-96-26, cTIT Tecnnical Report Series 96-26.

[22] A. Belinfante, “Jtorx: Exploring model-based testing,” Netherlands, 9 2014, iPA Dissertation series no. 2014-09.

[23] B. K. Aichernig, E. Jöbstl, and S. Tiran, “Model-based mutation testing via symbolic refinement checking,” Science of Computer Programming, vol. 97, 2015, pp. 383 – 404, special Issue: Selected Papers from the 12th International Conference on Quality Software (QSIC 2012). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642314002329

[24] S. Anand et al., “An orchestrated survey of methodologies for automated software test case generation,” Journal of Systems and Software, vol. 86, no. 8, 2013, pp. 1978 – 2001, accessed on: 2019-08. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121213000563

[25] W. Mostowski, E. Poll, J. Schmaltz, J. Tretmans, and R. Wichers Schreur, “Model-Based Testing of Electronic Passports,” in Formal Methods for Industrial Critical Systems, M. Alpuente, B. Cook, and C. Joubert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 207–209.

[26] A. Belinfante, L. Frantzen, and C. Schallhart, 14 Tools for Test Case Generation. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 391–438, accessed on: 2019-06. [Online]. Available: https://doi.org/10.1007/11498490_18

[27] P. Bhateja, “A tgv-like approach for asynchronous testing,” in Proceedings of the 7th India Software Engineering Conference, ser. ISEC ’14. New York, NY, USA: ACM, 2014, pp. 13:1–13:6, accessed on: 2018-05. [Online]. Available: http://doi.acm.org/10.1145/2590748.2590761

[28] C. Jard and T. Jéron, “Tgv: theory, principles and algorithms,” International Journal on Software Tools for Technology Transfer, vol. 7, no. 4, Aug 2005, pp. 297–315, accessed on: 2019-08. [Online]. Available: https://doi.org/10.1007/s10009-004-0153-x

 ISBN: 978-1-61208-752-8