*Journal of Software Engineering Research and Development, 2021, 9:6,* doi: 10.5753/jserd.2021.1094

# EVEREST: An Automatic Model-Based Testing Tool for Asynchronous Reactive Systems*

**Adilson Luiz Bonifacio** **Universidade Estadual de Londrina** ***bonifacio@uel.br***
**Camila Sonoda Gomes** **Universidade Estadual de Londrina** ***camilasonoda@uel.br***

**Abstract** Reactive systems are characterized by their interaction with the environment, where the exchange of the input and output stimuli, usually, occurs asynchronously. Systems of this nature, in general, require a rigorous testing activity in the development process. Therefore model-based testing has been successfully applied over asynchronous reactive systems using Input Output Labeled Transition System (IOLTS) as the basis. In this work, we present a reactive testing tool to check conformance, generate test suites, and run test cases using IOLTS models. Our tool can check whether the behavior of an implementation under test (IUT) complies with the behavior of its respective specification. We have implemented two conformance relations in our tool: the classical ioco relation; and the conformance based on regular languages. The tool also provides a test suite generation in a black-box testing setting for finding faults over IUTs according to a specific domain. In addition, we describe some case studies to probe the tool's functionalities and also give a comparative analysis. Finally, we offer practical experiments to evaluate the performance of our tool using several scenarios.

***Keywords:*** *Model-based testing, conformance checking, test generation, reactive systems, automatic tool*

## 1 Introduction

Several real-world systems are ruled by reactive behaviors that interact constantly with the environment by receiving input stimuli and producing outputs in response. Systems of this nature, in general, are also critical thus requiring precise and automatic support in the development process. Model-based testing methods and their respective tools have been largely applied in the testing activity when developing systems. The Input Output Labeled Transition System (IOLTS) (Tretmans, 2008) has been commonly employed as the formalism on modeling and testing asynchronous reactive systems. An IOLTS can then specify desired behaviors of an implementation candidate and the testing task can be applied to find faults on it.

One important issue of model-based testing is conformance checking where we can verify whether a given Implementation Under Test (IUT) complies with its corresponding specification according to a certain fault model. Here we treat the classical notion of Input Output Conformance Testing (ioco) (Tretmans, 2008) and the more recent testing conformance relation based on regular languages (Bonifacio and Moura, 2019) to define fault models. The test generation is also an important task of model-based testing, especially when generating test cases for reactive systems in a black-box setting. In this work, we present an automatic tool, named Everest[1] (Gomes and Bonifacio, 2019), that can check conformance between a given IUT and its respective specification. Our tool can also generate test suites based on specifications modeled by IOLTS and enable black-box testing over IUTs.

We show that Everest has a wider range of applications when compared to other testing tools since it implements not only the classical ioco relation but also the more recent language-based conformance checking. We also describe real-world scenarios to relate both approaches, where the language-based conformance method has been able to find faults which cannot be detected by using ioco relation. Further, experiments are performed to evaluate our tool when generating and running test suites in a black-box scenario, and also to compare and evaluate Everest against a well-known tool (Belinfante, 2010a) from literature w.r.t. the conformance checking task.

The remainder of this paper is organized as follows. We comment on related works in Section 2. Section 3 describes the conformance checking approaches and the test suite generation method. In Section 4 we discuss important aspects comparing Everest to another tool from the literature and also present a real-world case study. Practical experiments of conformance checking and test suite generation are given in Section 5. Section 6 offers some concluding remarks and future directions.

## 2 Related Works

Reactive systems have been properly specified by IOLTS models to describe their syntax and semantics. Hence model-based testing techniques and practical tools have been applied in testing activities to support the system development using IOLTSs. Therefore several works have studied aspects related to IOLTS-based testing such as test generation, conformance relations, and their checking methods. Here we survey on some works that are more closely related to our testing tool and its features.

*Supported by CAPES.
[1] *conformancE Verification on tEsting ReactivE SysTems*

The ioco relation has been proposed by Tretmans (2008) for IOLTS models, where IUTs are treated as black-boxes, *i.e.*, the tester, which is seen as an artificial environment that drives the test activity, has no access to the internal structure of IUTs. However, some restrictions must be guaranteed over the specification, IUT and tester models, such as input-completeness and output-determinism. Further, the algorithms therein are more theoretical and may lead to infinite test suites, making it more difficult to devise solutions for practical applications.

An ioco-based testing theory has been also proposed by de Vries (2001) to obtain $e$-complete test suites. This approach focuses on specific test purposes that share particular properties related to certain testing goals. Only observable behaviors based on specific criteria are considered when testing black-box IUTs, which turns out that the test purposes somewhat limit the fault coverage spectrum, *e.g.*, producing inconclusive verdicts. The test generation method also produces large, even infinite, test suites thus requiring a test selection criteria to avoid this problem.

Simão and Petrenko (2014) have described an approach to generate finite ioco-complete test suites for a class of IOLTS models. However, their approach imposes a number of restrictions on the models. Test purposes must be single-input and output-complete, and specifications and IUTs must be input-complete, progressive, and initially-connected. So the class of IOLTS models that can be tested are very restricted according to their fault model.

Roehm et al. (2016) have introduced a conformance test based on safety properties. Despite being a weaker relation than trace-inclusion conformance, it allows for tuning a trade-off between accuracy and computational complexity when checking conformance. So their approach searches for counter-examples instead of verifying the whole system. However, this approach and previous ones have a more theoretical leaning and we are not aware of practical tools and their algorithms.

A more recent work has been proposed by Bonifacio and Moura (2019) where few restrictions are considered and finite sets of test purposes can be generated in practical situations. In some rare cases their algorithm may lead to exponential sized testers, but the approach allows for a wider class of IOLTS models, and a low degree polynomial time algorithm is devised for efficiently testing ioco-conformance in practical applications.

In this work, we have implemented the more general and recent approach (Bonifacio and Moura, 2019) with the language-based and ioco conformance relations, as well as the test suite generation for white-box and black-box scenarios. In the literature, we have found JTorx (Belinfante, 2014, 2010b), a more closely related testing tool that implements the ioco relation and the uioco variation for underspecified models. TGV (Mark Utting, 2007; Calamé, 2005; Jard and Jéron, 2005) is also a testing tool designed for checking ioco-conformance, similarly to Testor (Marsso et al., 2018), an on-the-fly test case generation tool. However, the test generation methods of TGV and Testor are only sound, *i.e.*, they are not exhaustive, and so we cannot get complete test suites. Further, the soundness property over the generated test suites is only guaranteed from both tools over specific test purposes.

Although several other tools have been proposed for model-based testing many of them somewhat move away from the scope of our work. For instance, some tools and approaches implement a variation of ioco-theory, *e.g.* rioco and sioco relations, such as STG (Symbolic Test Generator) (Clarke et al., 2002), TorXAkis (Mostowski et al., 2009) and UPPAAL-TRON (Larsen et al., 2005) that deal with symbolic and timed models.

## 3 A Model-Based Testing Method

Asynchronous reactive systems are commonly specified by IOLTS models, a variation of Labeled Transition Systems (LTSs) (Tretmans, 1993) with the partitioning of input and output labels.

**Definition 1** *An Input/Output Labeled Transition System (IOLTS) is a tuple $\mathcal{S} = (S, s_0, L_I, L_U, T)$ where:*

- *$S$ is a finite 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 an underlying LTS associated with $\mathcal{S}$.*

We indicate a transition by $(s, l, r) \in T$ where $s \in S$ is the source state and $r \in S$ is the target state labeled by $l \in (L \cup \{\tau\})$. A transition $(s, \tau, r) \in T$ indicates an internal action, which means that an external observer cannot see the movement from $s$ to $r$ in the model.

An IOLTS may also have quiescent states. A state $s$ is quiescent if no output $x \in L_U$ or an internal action $\tau$ is defined on it (Tretmans, 2008). When a state $s$ is quiescent a transition $(s, \delta, s)$ is added to $T$, where $\delta \notin L_\tau$. Note that $L \cup \{\tau\}$ is denoted by $L_\tau$ to ease the notation. We also note that in a real black-box testing scenario, where an IUT sends messages to the tester and receives back responses, quiescence will indicate that the IUT can no longer respond to the tester, it has timed out, or that it is slow (Bonifacio and Moura, 2019).

In what follows we define the semantics over IOLTS/LTS models, but first we introduce the notion of paths.

**Definition 2** *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 $L_\tau^\star$. We say that $\sigma$ 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 $\alpha$ is an* observable path *from $p$ to $q$ in $\mathcal{S}$ if we remove all internal actions $\tau$ from $\sigma$.*

A path can also be denoted by $s \xrightarrow{\sigma} s'$, where the behavior $\sigma \in L_\tau^\star$ starts in the state $s \in S$ and reaches the state $s' \in S$. An observable path $\sigma$, 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 target state is not important. All paths starting at a state $s$ we call *paths of* $s$.

Now we give the semantics over IOLTS/LTS models.

**Definition 3** *Let $\mathcal{S} = (S, s_0, L, T)$ be a LTS and $s \in S$:*

1. *the set of all paths from s is denoted by $tr(s) = \{\sigma | s \xrightarrow{\sigma}\}$ and the set of all observable paths from s is denoted by $otr(s) = \{\sigma | s \overset{\sigma}{\Rightarrow}\}$.*
2. *the semantics of $\mathcal{S}$ is given by $tr(s_0)$ or $tr(\mathcal{S})$ and the observable semantics of $\mathcal{S}$ is denoted by $otr(s_0)$ or $otr(\mathcal{S})$.*

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

## 3.1 Checking Conformance on Reactive Systems

Given an IOLTS specification, a conformance checking task can determine whether an IUT complies with the corresponding specification according to a specific fault model. The classical ioco (Tretmans, 2008) relation establishes a notion of conformance where input stimuli are applied to both IUT and specification models to observe whether outputs produced by the IUT are also defined in the specification (Bonifacio and Moura, 2019).

**Definition 4** *Let $\mathcal{S} = (S, s_0, L_I, L_U, T)$ be a specification and let $\mathcal{I} = (Q, q_0, L_I, L_U, R)$ 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 every $s \in S$. Otherwise, we get that $r\mathcal{I}\, ioco\, \mathcal{S}$ does not hold.*

A more recent conformance relation (Bonifacio and Moura, 2019) has also been proposed using regular languages. Given an IUT $\mathcal{I}$, a specification $\mathcal{S}$, and regular languages $D$ and $F$, we say that $\mathcal{I}$ 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 it is specified in $\mathcal{S}$, and all desirable behaviors of $D$ are observed in $\mathcal{I}$ and they are also specified in $\mathcal{S}$.

**Definition 5** *Given an alphabet $L = L_I \cup L_U$ and languages $\mathcal{D}, \mathcal{F} \subseteq L^*$ over L. Let $\mathcal{S}$ and $\mathcal{I}$ be IOLTS models over L we have that $\mathcal{I}\, conf_{D,F}\, \mathcal{S}$ if, and only if,*

*(i) $\sigma \in otr(\mathcal{I}) \cap F$, then $\sigma \notin otr(\mathcal{S})$; and*
*(ii) $\sigma \in otr(\mathcal{I}) \cap D$, so $\sigma \in otr(\mathcal{S})$.*

This new notion with a wider fault coverage is established by Proposition 1, where desirable and undesirable behaviors can be specified by regular languages.

**Proposition 1** *(Bonifacio and Moura, 2019). Let $\mathcal{S}$ and $\mathcal{I}$ be IOLTS models over an alphabet $L = L_I \cup L_U$, and the regular languages $D, F \subseteq L^*$ 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 $\overline{otr}(\mathcal{S}) = L^* - otr(\mathcal{S})$.*

Both notions of conformance can be related by the following lemma, where the language-based conformance relation given in Definition 5 restrains the classical ioco relation given by Definition 4.

**Lemma 1** *(Bonifacio and Moura, 2019). Let $\mathcal{I} = (Q, q_0, L_I, L_U, R)$ be an IUT and let $\mathcal{S} = (S, s_0, L_I, L_U, T)$ be a specification, we say that $\mathcal{I}\, ioco\, \mathcal{S}$ if, and only if, $\mathcal{I}\, conf_{D,F}\, \mathcal{S}$ when $D = otr(\mathcal{S})L_U$ and $F = \emptyset$.*

Bonifacio and Moura (2019) have proposed the language-based conformance checking using the theory of automata (Sipser, 2006). LTS/IOLTS models are transformed into Finite State Automata (FSA), where the semantics of FSA is given by the language it accepts. So $R \subseteq L^\star$ is *regular* if there exists an FSA $\mathcal{M}$ such that $L(\mathcal{M}) = R$, where $L$ is an alphabet. Therefore we can effectively construct the automatons $\mathcal{A}_D$ and $\mathcal{A}_F$ where $D$ and $F$ are regular languages such that $D = L(\mathcal{A}_D)$ and $F = L(\mathcal{A}_F)$.

Now we define *test case* and *test suite* according to regular languages.

**Definition 6** *Let L be a set of symbols, a test suite T over L is a language $T \subseteq L^\star$, where each $\sigma \in T$ is a test case.*

We can see that there will always be an FSA $\mathcal{A}$ that accepts a test suite since it is a regular language, where the final states are fault states. Thus the set of undesirable behaviors, so-called *fault model* of $\mathcal{S}$ (Bonifacio and Moura, 2019), is defined by the fault states.

Therefore we can obtain a complete test suite for an IOLTS specification $\mathcal{S}$ and a pair of languages $(D, F)$ using Proposition 1. That is, we can detect the absence of desirable behaviors specified by $D$ and the presence of undesirable behaviors specified by $F$ in the specification $\mathcal{S}$ using the test suite $T = [(D \cap \overline{otr}(\mathcal{S})) \cup (F \cap otr(\mathcal{S}))]$. An IUT $\mathcal{I}$ is then declared in compliance to a specification $\mathcal{S}$ if there is no test case of the test suite $T$ that is also a behavior of $\mathcal{I}$ (Bonifacio and Moura, 2019).

The testing process first obtains an automaton $\mathcal{A}_1$ induced by the IOLTS specification $\mathcal{S}$. Since $L(\mathcal{A}_1) = otr(\mathcal{S})$ we can effectively construct 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 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 we state an algorithm with a polynomial time complexity using the language-based conformance relation.

**Proposition 2** *(Bonifacio and Moura, 2019) 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.*

Now using Lemma 1 we establish a relationship between the ioco and language-based relations in Theorem 1.

**Theorem 1** *(Bonifacio and Moura, 2019) Let $\mathcal{S}$ and $\mathcal{I}$ be deterministic 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 effec-*

*tively construct an algorithm with polynomial time complexity $\Theta(n_S n_I n_L)$ that checks whether $\mathcal{I}$ ioco $\mathcal{S}$ holds.*

## 3.2 Complete Test Suite Generation

In this work, we also provide the test suite generation in a black-box testing setting using the notion of test purposes (Tretmans, 2008). A Test Purpose (TP) is formally defined by an IOLTS with two special states $\{pass, fail\}$ and, in practice, it represents an external tester that interacts with an IUT. Thus a fault model is composed of TPs that are derived from a given specification.

To ease the notation from now on we will denote by $\mathcal{IO}(L_I, L_U)$ the class of all IOLTSs over $L = L_I \cup L_U$.

**Definition 7** *Let $L_I$ and $L_U$ be the input and output alphabets, respectively, with $L = L_I \cup L_U$. A Test Purpose (TP) over $L$ is defined by an IOLTS $\mathcal{T} \in \mathcal{IO}(L_U, L_I)$ such that for all $\sigma \in L^*$ does not hold $fail \overset{\sigma}{\Rightarrow} pass$ and $pass \overset{\sigma}{\Rightarrow} fail$. The fault model over $L$ is the finite set of TPs over $L$.*

The test case generation proposed by Tretmans (2008), based on ioco relation, imposes some restrictions over the formal models. All TPs must be acyclic, with a finite run, and input-enabled, since the tester cannot predict the output produced by a black-box IUT. Therefore, all output actions that are produced by the IUT must be enabled in the respective TP. Moreover, they must be output-deterministic, *i.e.* each state can send only one output symbol to the IUT in order to avoid arbitrary and non-deterministic choices. In the $pass$ and $fail$ states only self-loop transitions are allowed since verdicts are obtained in these states.

**Definition 8** *Let $\mathcal{S} \in \mathcal{IO}(L_I, L_U)$. We say that $\mathcal{S}$ is output-deterministic if $|out(s)| = 1$ and $\mathcal{S}$ is input-enabled if $inp(s) = L_I$ for all $s \in S$, where $out(s)$ and $inp(s)$ give outputs and inputs, respectively, defined at state $s$.*

Hence all restrictions imposed by Tretmans (2008) are satisfied when a TP is input-enabled, output-deterministic, and acyclic except for $pass$ and $fail$ states. However, we see that a bound over the number of states to be considered in the IUTs must be imposed to keep the TP acyclic. So the test suite completeness property is guaranteed if given an IUT $\mathcal{I}$ and a specification $\mathcal{S}$, $\mathcal{I}$ ioco $\mathcal{S}$ for all IUT that conforms to $\mathcal{S}$. Otherwise we say that $\mathcal{I}$ ioco $\mathcal{S}$ does not hold. Therefore we define a class of implementations to guarantee the ioco completeness property on generating test suites establishing an upper bound on the number of states over the IUTs.

Now we are in a position to construct a complete test suite using the notion of TPs. But first we generate a multigraph structure as proposed by (Bonifacio and Moura, 2019). So given an IUT $\mathcal{I}$ and a specification $\mathcal{S}$, we remark that $m$ is the bound over the number of states to be considered on the IUTs, and $n$ is the number of states in $\mathcal{S}$. Then the multigraph must have $mn + 1$ levels, and at each level if a transition of $\mathcal{S}$ gives rise to a cycle then we must create a transition onto states on next level of the multigraph. A $fail$ state is also added and new transitions from every state of the multigraph are defined to the $fail$ labeled by all $l \in L_U$ when $l$ is not defined.

Having an acyclic multigraph at hand we can extract TPs using a simple breadth-first search algorithm from the initial state to $fail$. We can guarantee the input-enabledness property by adding the *pass* state to the TP and, for every output of $L_U$ and all states, we add transitions to the $pass$ state where the output is not defined. Self-loops labeled by each $l \in L_U$ are also added to the $pass$ and $fail$ states. The output-deterministic property is also guaranteed by adding a transition from every state to $pass$ where there is no input of $L_I$ defined. Note that we always refer to an input symbol of $L_U$ or an output symbol of $L_I$ from the perspective of the IUT, as commonly denoted in the literature (Tretmans, 2008; Bonifacio and Moura, 2019).

The test run is then defined by the synchronous product between a TP $\mathcal{T}$ and an IUT $\mathcal{I}$, denoted by $\mathcal{I} \times \mathcal{T}$. The TP interacts with the IUT producing outputs that are sent to $\mathcal{I}$ as inputs. Likewise, the IUT receives actions from the TP and produces outputs that are sent to $\mathcal{T}$ as inputs. So the output alphabet of $\mathcal{T}$ corresponds to $L_I$, the input alphabet of the IUT, and the input alphabet of $\mathcal{T}$ corresponds to $L_U$, the output alphabet of the IUT.

**Definition 9** *Let $\mathcal{I} = (S_\mathcal{I}, q_0, L_I, L_U, T_\mathcal{I}) \in \mathcal{IO}(L_I, L_U)$ be an implementation and $\mathcal{T} = (S_\mathcal{T}, q_0, L_U, L_I, T_\mathcal{T})) \in \mathcal{IO}(L_U, L_I)$ be a TP. We say that $\mathcal{I}$ passes $\mathcal{T}$ if for any $\sigma \in (L_I, L_U)^*$ and any state $q \in S_\mathcal{I}$, we do not have $(t_0, q_0) \overset{\sigma}{\Rightarrow} (fail, q)$ in $\mathcal{T} \times \mathcal{I}$. A path can be denoted by $q_0 \overset{\sigma}{\Rightarrow} q$ where the behavior $\sigma$ starts in the state $q_0$ and reaches the state $q$. Let $\mathcal{M}$ be the fault model, we say that $\mathcal{I}$ pass $\mathcal{M}$, if $\mathcal{I}$ passes all TPs in $\mathcal{M}$. Then given an IOLTS $\mathcal{S}$ and a set $\mathcal{IMP} \subseteq \mathcal{IO}(L_U, L_I)[m]$, we say that $\mathcal{M}$ is $m$-ioco-complete to $\mathcal{S}$ concerning $\mathcal{IMP}$ if for all IUT $\mathcal{I} \in \mathcal{IMP}$ we have $\mathcal{I}$ ioco $\mathcal{S}$ if, and only if, $\mathcal{I}$ passes $\mathcal{M}$.*

The verdicts are obtained when TPs reach the special states. The *fail* verdict gives rise to a fault behavior whereas the *pass* verdict denotes a desirable behavior. Further details can be found in (Bonifacio and Moura, 2019; Gomes and Bonifacio, 2019).

Finally, the next proposition determines a fault model that is composed of TPs obtained from a multigraph which, in turn, is constructed based on the corresponding specification.

**Proposition 3** *Let the deterministic IOLTS $\mathcal{S} \in \mathcal{IO}(L_I, L_U)$ and $m \geq 1$. Then there is a fault model $\mathcal{M}$ that is $m$-ioco-complete for $\mathcal{S}$ relatively to $\mathcal{IO}(L_I, L_U)[m]$, IOLTSs at most $m$ states, whose TPs are deterministic, output-deterministic, input-enabled, and acyclic except for self-loops on pass and fail states.*

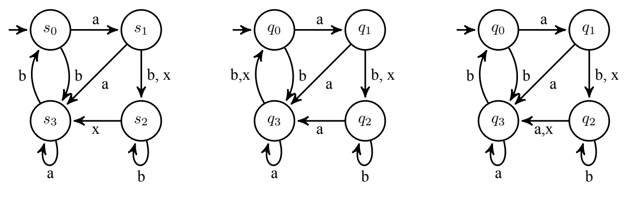
# 4 A Testing Tool for Reactive Systems

`Everest` (Gomes and Bonifacio, 2019) has been developed to check conformance, generate test suites, and run tests over reactive systems specified by LTS/IOLTS models. We have organized the tool's architecture in four modules: configuration; ioco conformance; language-based conformance; and test generation & run. The configuration module allows us to settle the testing scenario, and the checking conformance modules can yield verdicts of testing. When an IUT does not conform to the specification our tool yields the verdict along with the paths induced by the test cases that could detect the corresponding faults. The test generation & run module enables the multigraph and test purpose generation, and also allows for running test suites over the IUTs.

In this section, we look over the checking conformance and test suite generation processes. First we present some general examples to compare the conformance checking processes of Everest and JTorx. Next we show how our test suite generation method using the language-based conformance stands out from the classical approach. Finally we describe a real-world case study of an Automatic Teller Machine (ATM) to explore some real scenarios, and then give a comparative analysis between the practical tools.

## 4.1 Conformance checking process

We apply some examples to explore characteristics from both Everest and JTorx tools when checking conformance. Let $\mathcal{S}$ be a specification depicted in Figure 1a and let $\mathcal{R}$ and $\mathcal{Q}$ be IUTs depicted in Figures 1b and 1c, respectively, with $L_I = \{a, b\}$ and $L_U = \{x\}$.



**(a)** Specification $\mathcal{S}$ **(b)** IUT $\mathcal{R}$ **(c)** IUT $\mathcal{Q}$

**Figure 1.** IOLTS Models

In the first checking run we have verified whether the IUT $\mathcal{R}$ conforms to the specification $\mathcal{S}$. Our tool yielded a verdict of non-conformance and generated the test suite $T_1 = \{b, aa, ba, aaa, ab, ax, abb, axb\}$. All test cases were induced by different paths that reach a fault and were extracted using a transition cover strategy over the specification. JTorx also yielded the same verdict for this first run, as expected, but it has generated a test suite $T_2 = \{b, ax, ab\}$. We can see that $T_2 \subseteq T_1$, *i.e.*, JTorx has generated only one test case per fault in contrast to Everest that has produced several test cases using a transition coverage. Hence we notice that Everest has provided a wide range of coverage which can be more useful in a fault mitigation process.

In a second scenario, we checked the IUT $\mathcal{Q}$ against the specification $\mathcal{S}$. At this time no fault was detected by both tools using the classical ioco relation. However, Everest could find a fault using the language-based conformance relation, where the set of desirable behaviors were specified by the regular language $D = (a|b)^*ax$ and no undesirable behavior was defined, so $F = \emptyset$. The set $D$ denotes behaviors that are induced by paths finishing with an input action $a$ followed by an output $x$ produced in response. A verdict of non-conformance was obtained by our tool revealing a fault detected by the test suite $T = \{ababax, abaabax\}$. We remark that JTorx, using the classical ioco relation, was not able to detect this fault. So we can note that Everest is more

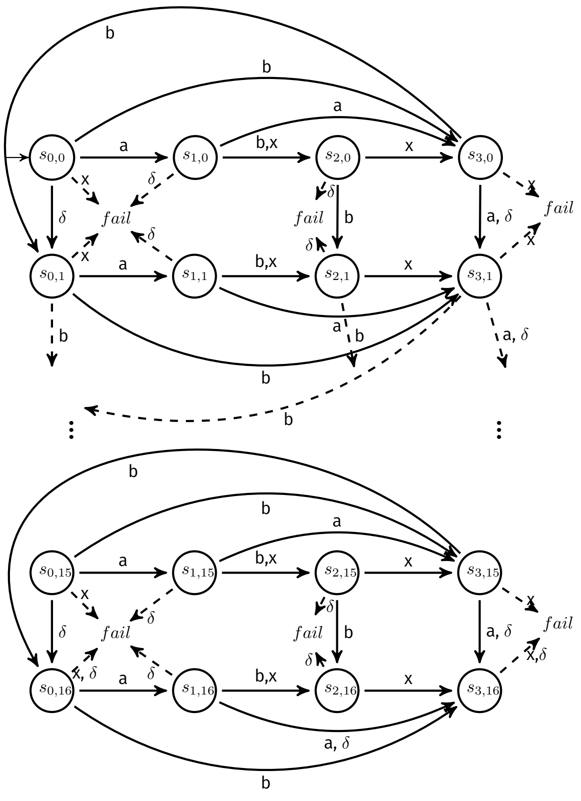


**Figure 2.** A direct acyclic multi-graph D for specification $\mathcal{S}$

general in this sense and can be applied to a wider range of scenarios when compared to JTorx.

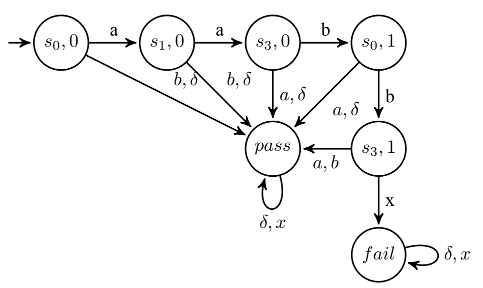
## 4.2 Everest test suite generation

We have seen that a conformance checking is run over an IUT against a given specification to yield test verdicts. If the verdict is positive, *i.e.*, faults are detected, then an associated test suite is generated with test cases that can reveal such faults. In addition, Everest can also generate complete test suites relative to a given specification.

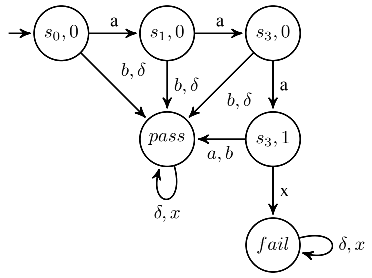
To illustrate the test suite generation process of Everest again we assume $\mathcal{S}$ as the specification depicted in Figure 1a. In the first step a direct acyclic multigraph must be constructed according to the specification, as described in Section 3. Figure 2 partially depicts the multigraph with four states at each level once the specification $\mathcal{S}$ has four states ($n = 4$). Every transition in the multigraph must go either to the next level or from left to right in the figure at the same level to secure the acyclic property. In this case we have considered IUTs with at most four states, *i.e.* the same number of states as found in the specification ($m = n = 4$). Therefore the multigraph has $mn + 1 = 17$ levels. Figure 2 shows the first two levels and also the two last levels of the multigraph. Note that we replicate the fail state in order not to clutter the figure.

With the multigraph at hand, we can apply a breadth-first search algorithm to extract paths from the initial node $s_{0,0}$ up to the fail state. We can take, for instance, the sequence $\alpha_1 = aabbx$. We see that $\alpha_1$ induces the path $s_{0,0} \to s_{1,0} \to$

$s_{3,0} \to s_{0,1} \to s_{3,1} \to fail$ in the multigraph. From Proposition 3 we can then obtain a deterministic, acyclic, input-enabled, and output-deterministic test purpose $\mathcal{T}_1$ over $\alpha_1$ as depicted in Figure 3a.



**(a)** TP $\mathcal{T}_1$ induced by $aabbx$



**(b)** TP $\mathcal{T}_2$ induced by $aaax$

**Figure 3.** TPs from multigraph of Figure 2

Note that the input-enabledness property is also guaranteed by adding a pass state and transitions from states where no output is defined to the pass state. The construction is complete by adding self-loops to the pass and fail states labeled by all output actions. Regarding the output-determinism property, for every state that no input action is defined, we also create a new transition from this state to the pass state labeled by any input action.

For the sake of exemplification we take $\alpha_2 = aaax$ as a distinct sequence. In the same way we obtain the induced path over the multigraph and construct the corresponding deterministic, acyclic, input-enabled and output-deterministic test purpose $\mathcal{T}_2$ as depicted in Figure 3b. Everest has indeed automatically constructed other fifteen test purposes based on paths induced by the set $\{\alpha_1, \alpha_2, x, a\delta, bx, \delta x, aax, bbx, ax\delta, ab\delta, \delta bx, b\delta x, aabx, bbbx, aa\delta x\}$ of sequences. From the TPs of Figure 3, Everest could generate the test suite $T = \{\alpha_1, \alpha_2, b, \delta, b\delta, bx, ab, a\delta, aa\delta, aaa, aab, aab\delta, aaba, aaaa, aaab, aaax\delta, aaaxx, aabba, aabbb, aabbx\delta, aabbxx\}$.

We then apply the test suite $T$ to the IUT $\mathcal{R}$ and a fault could be detected. By a simple inspection we see that all test cases that lead $\mathcal{R}$ from state $q_0$ to the same state $q_0$ can detect this fault. Notice that the output $x$ is produced at state $q_3$ of $\mathcal{R}$ whereas $x$ is not defined at state $s_3$ of $\mathcal{S}$. So Everest exhibits a verdict of non-conformance which means that $\mathcal{R}$ does not pass the test suite, declaring that $\mathcal{R}\, ioco\, \mathcal{S}$ does not hold.

## 4.3 A real-world case study

Now we present a real-world system to be put under test using the automatic tools. We specify functionalities of an Automatic Teller Machine (ATM) (Mark Utting, 2007; Naik and Tripathy, 2018) using an IOLTS model with the input stimuli $L_I = \{ic, pin, acc, tra, sta, wd, amo\}$, and the output responses $L_U = \{cpi, bpi, mon, rec, ins, sho\}$. The intended meaning of the input actions are: $ic$, denotes the action when the user inserts his/her card into the ATM; $pin$, indicates the pin code has been provided by the user; $tra$, requires the transfer amount; $acc$, indicates that a target account has been provided; $sta$, requires an account statement; $wd$, indicates that the user has requested a withdrawal; and $amo$, denotes the balance account. Also we give the meaning of the output alphabet: $cpi$, says the pin code is correct; $bpi$, says the provided pin is wrong; $mon$, indicates the money has been released; $rec$, indicates the receipt has been provided to the user; $ins$, denotes an insufficient balance on the account; and $sho$, indicates the statement has been shown to the user.

We model the withdrawal operation by the IOLTS $\mathcal{A}$ of Figure 4. Note that if the requested amount (amo) is greater than the available amount (ins) then the withdrawal cannot be performed and the process reaches state $s_3$ where a new withdrawal operation can be requested again. Some additional

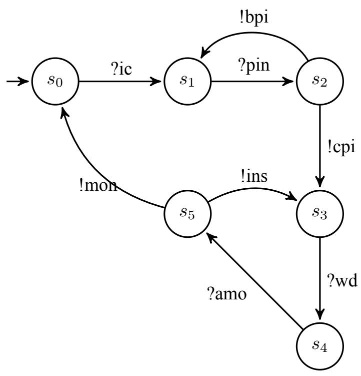


**Figure 4.** ATM specification $\mathcal{A}$

functionalities are also specified by the IOLTS $\mathcal{B}$ of Figure 5. In this case we consider not only the withdrawal (wd) operation but also the transfer (tra) and statement (sta) operations.

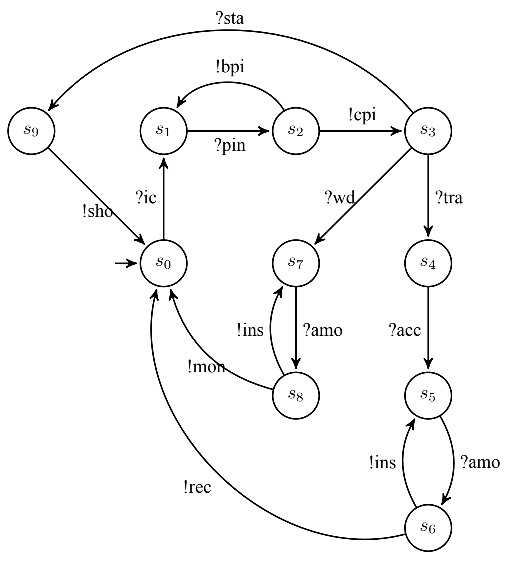


**Figure 5.** ATM specification $\mathcal{B}$

Assume the IOLTS $\mathcal{Z}$ depicted in Figure 6 as an IUT that implements the withdrawal (wd) and transfer (tra) operations. We observe that if the requested amount (amo) in a withdrawal is greater than the available amount then the IUT reaches the state $s_7$ where the user can choose a new amount again.

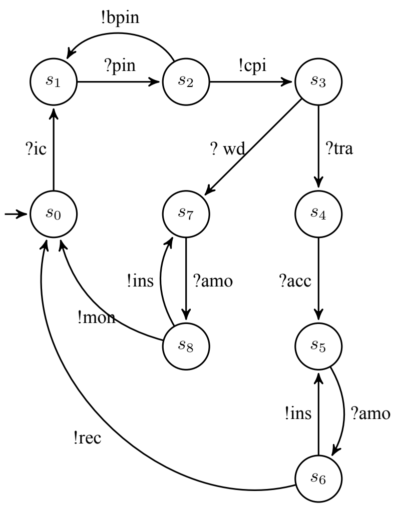


**Figure 6.** IUT $\mathcal{Z}$

Now as a first testing scenario we check whether the IUT $\mathcal{Z}$ conforms to the specification $\mathcal{A}$. We, then, run JTorx and Everest over these models to obtain conformance verdicts using the ioco relation. Both tools have returned the same verdict where $\mathcal{Z}$ complies with $\mathcal{A}$. In a second round, we run Everest using the language-based conformance relation and, at this time, a fault could be detected in the IUT $\mathcal{Z}$. The set of desirable behaviors was given by $D = \{\text{ic pin cpi wd amo ins amo}\}$, *i.e.*, a sequence of actions where the account balance is not enough according to the requested withdrawal, and the user must provide a new value. Everest has generated the test case $\{ic \to pin \to cpi \to wd \to amo \to ins \to amo\}$ because the behavior specified in $D$ is not an observable behavior in the specification model but the IUT $\mathcal{Z}$ implements it.

In a second scenario we want to verify the reliability of verdicts obtained by JTorx using ioco relation. We know that original underspecified models (See Section 4.4) requires some labor in such a way that *self-loop* transitions must be added to get all states completely specified so handing over an *input-enabled* model. Notice that the IUT $\mathcal{Z}$ is underspecified, so JTorx must change the model to guarantee the *input-enabledness* property on the IUT. After changing the model we check whether $\mathcal{Z}$ ioco-conforms to the specification $\mathcal{B}$ using JTorx. It is easy to see that the original behavior of the IUT has been modified and, in this case, a fault is then detected by the test case $\{ic \to pin \to cpin \to sta\}$.

We have also applied this second scenario to Everest using the ioco relation. In the opposite direction to the JTorx, no fault was detected by Everest once the fault behavior $\{ic \to pin \to cpin \to sta\}$ is not specified in the IUT $\mathcal{Z}$. We see that the detection of this fault by JTorx is, in fact, a false positive, due to an extra behavior that has been added after changing the IUT $\mathcal{Z}$ to become it *input-enabled.*

We also remark that Everest can detect this same fault when checking ioco conformance over the same modified model. Note that $ic$ is the only single action defined at state $s_0$ of the IUT $\mathcal{Z}$. When JTorx turns $\mathcal{Z}$ into an *input-enabled* model all input actions become enabled at all states, which is contradictory to the real functionality. For instance, we see that the action $amo$, *i.e.*, the amount value to be withdrawn, becomes enabled at state $s_0$. However, if a transfer operation ($tra$) is chosen instead of a withdrawal ($wd$), the amount value to be withdrawn should not be enabled at this moment. Hence we see that any change performed over the former model modifies the original behavior of the IUT, leading to an inaccurate conformance checking verdict relative to the real functionality of the ATM.

In the last scenario we take the IOLTS $\mathcal{Y}$ depicted in Figure 7 as a new IUT. The IUT $\mathcal{Y}$ differs from the specification

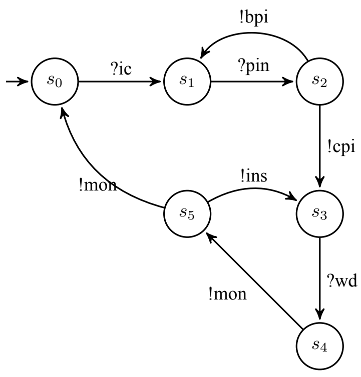


**Figure 7.** IUT $\mathcal{Y}$

$\mathcal{A}$ depicted in Figure 4 only by the transitions $(s_4, ?amo, s_5)$ and $(s_4, !mon, s_5)$, respectively. So the IUT allows a withdrawal operation with no checking over the balance ($amo$) before releasing the money ($mon$). By contrast, in the specification model, the balance is checked before releasing the money when the account balance is positive. The fault model was bounded at six states for the class of IUTs and Everest has generated eighty TPs based on the corresponding specification $\mathcal{A}$. The generated test suite has then been submitted to the IUT $\mathcal{Y}$, and a fault verdict could be obtained by the path $ic \to pin \to cpin \to wd$ using our tool. For the sake of completeness we have also applied this last scenario to JTorx, and a fault has also been detected by the test case $?ic, \delta, ?pin, !bpi, ?pin, !cpi, ?wd, !mon.$

## 4.4 A Comparative Analysis

Here we list some main aspects and compare Everest and JTorx. We have seen that both tools provide a mechanism to generate test suites, run test cases and check ioco conformance. Everest also provides the more general conformance checking based on regular languages. Further, our tool allows a complete test generation not only for the ioco relation but also for this more general conformance relation with a wider range of possibilities to specify desirable and undesirable behaviors.

JTorx's test generation employs an exhaustive strategy leading to the state space explosion problem making the process infeasible in practice. In the opposite direction, Everest is more flexible and allows for a complete test suite generation by setting the maximum number of states on the IUTs to be taken into account in the fault model.

JTorx also implements a random approach that chooses transitions to induce paths over the specification when generating test suites. Everest, however, only applies a random approach over the language-based conformance relation when desirable and/or undesirable behaviors are not provided by the tester. In this case, the test run is reduced to the problem of checking isomorphism between the IUT and the specification model.

We also note that both tools implement an online testing approach when IUTs are provided together with the specification. But only Everest provides an offline test generation process using the notion of multigraph and test purposes.

Regarding the conformance checking process, JTorx defines an online strategy where test cases are generated and right after they are already applied to the IUT. Everest follows an offline process where the whole test suite is generated and then all test cases are applied to the IUT. However we remark that Everest also has an online alternative process to check conformance where each test case obtained from the fault model is applied to IUT right after it is generated. Table 1 summarizes these aspects.

**Table 1.** Methods and Features

| | JTorx | **Everest** |
|---|---|---|
| **Conformance checking** | | |
| ioco theory | $\surd$ | $\surd$ |
| Language-based | X | $\surd$ |
| **Generation** | | |
| Test suite generation | $\surd$ | $\surd$ |
| **Test strategy** | | |
| online/offline | $\surd$ | $\surd$ |
| Test purpose | $\surd$ | $\surd$ |
| Random approach | $\surd$ | $\surd$ |

We also probe some properties over the specification and IUT models, test verdicts, and strategies of testing. See Table 2. Some restrictions are naturally imposed over the models when checking ioco conformance. Underspecified models, for instance, are not allowed on the IUT side and their internal structure must be changed to guarantee the input-enabledness.

**Table 2.** Properties and Tools

| | JTorx | **Everest** |
|---|---|---|
| **Properties** | | |
| Underspecified models | $\surd$[a] | $\surd$ |
| Non-input-enabledness | X | $\surd$ |
| Quiescence | $\surd$ | $\surd$ |
| **Veredicts** | | |
| Test run | $\surd$ | $\surd$ |
| Conformance | $\surd$ | $\surd$ |
| **Test mode** | | |
| White/black boxes testing | $\surd$ | $\surd$ |

[a]But the internal structure of models must be changed.

The language-based conformance relation does not require any restriction, that is, the more general method can deal with underspecified IUTs and specification models. Therefore Everest can handle underspecified models with no change over the models when checking conformance and also generating test suites using the language-based relation. JTorx, on the other hand, must completely explore the model's structure to add new transitions to guarantee the input-enabledness property.

We see that both tools can deal with quiescence models, where self-loops with $\delta$ actions are added at the quiescent states, and also give verdicts of conformance and run test cases in a similar way.

## 5 Practical Evaluation

In this section, we present the results of practical experiments that we have run to evaluate the tools' performance. First, we provide experiments to compare the checking conformance methods of Everest and JTorx in Subsection 5.1. Given an IUT and a specification, both tools can check whether the IUT is in conformance to the specification under ioco relation. Secondly, Subsection 5.2 assays the additional feature of Everest on generating and running test suites. In this case, given a specification model, we can generate test suites for a certain class of IUTs and then de facto apply them to the IUTs.

The experiments are classified into different groups according to the parameters under evaluation. Therefore, each group of experiments represents a different scenario, where either the specifications and the IUT models are changed to capture different situations of conformance checking, or test suite generation or test runs, *e.g.* the models must have a certain number of states and transitions. All experiments were performed using randomly generated models both for specifications and IUTs, while satisfying all required properties, if any, to avoid bias in the results. In some groups, we have taking into account submachines of specification models as the basis to generate IUTs with a certain percentage of modification.

We have organized all experiments by Research Questions (RQs) to get the desired analyses using different groups of scenarios. Our experiments have been performed on Intel Core i5 1.8 GHz CPU, with 8 GB of RAM on Windows 10.

### 5.1 Conformance checking of Everest and JTorx Tools

Here we report on some experiments to compare Everest and JTorx when checking conformance between an IUT and a given specification using their respective implementations of the ioco relation. So a single conformance checking run is defined by a pair of models, an IUT and a specification, where the result can be positive or negative. A verdict is said to be positive (ioco-conformance), when the IUT complies with the specification, or negative (non-ioco-conformance), when the IUT does not comply with the specification, according to the ioco relation.

We evaluate several parameters related to the specifications and IUTs, such as the number of states and the number of input/output actions on the models. In addition, we also consider experiments that derive verdicts of conformance and non-conformance on separated scenarios. But we remark that only input-enabled and deterministic models have been generated in our experiments to comply with the restrictions imposed by JTorx.

Each group of experiments on checking conformance is defined between one specification and ten IUT models. Therefore, each run is settled down by a pair of models, one IUT against the corresponding specification, and a group of experiments with ten specifications and ten IUTs for each specification that outcomes in hundred runs. s

Experiments with verdicts of ioco-conformance were run over IUT models obtained as submachines of their respective specifications, while IUT models with verdicts of non-ioco-conformance were randomly constructed by changing transitions from their corresponding specifications with a certain percentage of modification. Regarding IUT models with more states than the corresponding specifications, new states and new transitions have been randomly added to the models.

To illustrate, let $\mathcal{S}$ be a specification model with 20 states and 120 transitions. In order to get positive verdicts we randomly take IUT models as submachines of $\mathcal{S}$, choosing subsets of states from $\mathcal{S}$ and their respective transitions. When we want to guarantee negative verdicts for IUTs with 4% of modification from its respective specification we have to randomly choose 5 transitions to be modified, *i.e.*, over these 5 transitions we have to change the source state, or the target state, or even the action symbol. We have decided to generate IUTs using the specification models as the basis instead of completely randomized IUT models because in practical situations developers can make mistakes but, in general, they minimally implement the specified model, that is, real-world IUTs usually are not very assorted from the corresponding specification.

Hence in the first group of experiments the size of the alphabets are interchanged while we increase the number of states on the models; and in the second group we varied the number of states (and transitions, consequently) of the specification and IUT models in a stress testing. All processing times that are found in the graphics have been figured out from the mean value of the processing time of all experiments in the group.

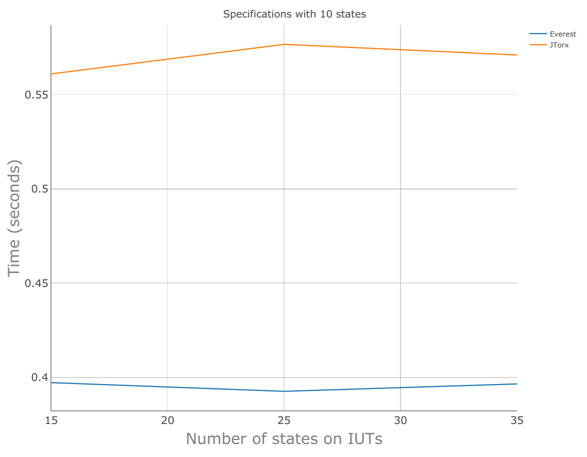
### 5.1.1 Reversing the size of input/output alphabets

In this first scenario we investigate the impact over conformance checking runs when the size of input and output alphabets are inversely proportional. We state the **RQ** as follows: *"How does the size of the input alphabet (output alphabet) impact the processing time on checking conformance?"*.

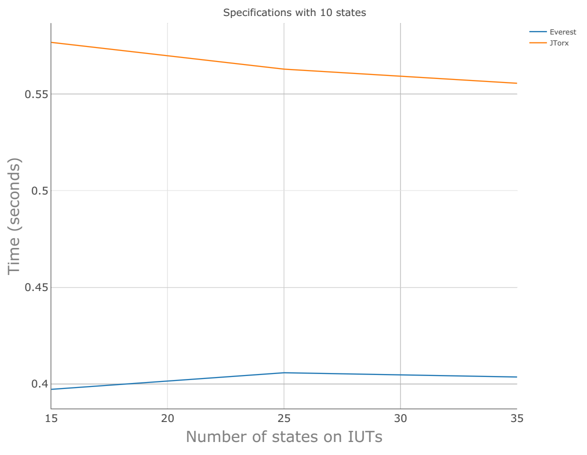
To answer this question we have run experiments where the size of input and output alphabets have been reversed on the models. First, we take a group of IOLTS models with 2 symbols in the input alphabet and 10 symbols in the output alphabet. In a second group of models we reserve the size of alphabets, so taking input alphabets with 10 symbols and output alphabets with only 2 symbols. We vary the number of states by 15, 25 and 35 on IUTs and get the specification with a fixed number of 10 states, both for verdicts of conformance and non-conformance.

From the results, we note only a small variation on the processing time when running experiments with verdicts of conformance either when the input alphabet is larger than the output alphabet or when we reverse them in size. See Figure 8. Our tool is only $2.56\%$ faster when running models with 2 inputs and 10 outputs (See Figure 8(a)) than when checking models whose size of their alphabets are reversed with 10 inputs and 2 outputs (See Figure 8(b)). Similarly, JTorx is only $3.51\%$ faster when reversing the size of the alphabets.

However, we can notice that Everest is faster than JTorx in both scenarios, where the size of the input alphabet is larger than the output alphabet, and vice-versa.



**(a)** 2 inputs, 10 outputs

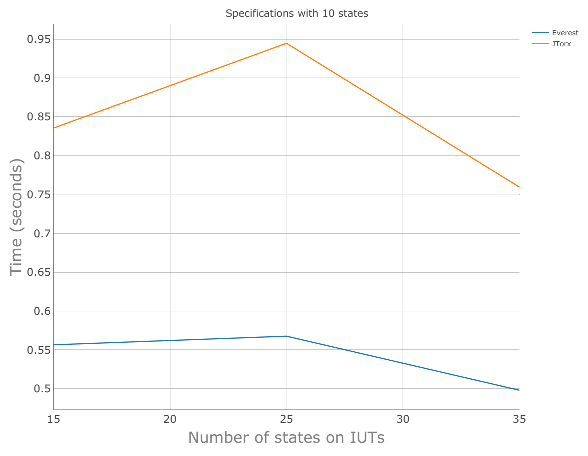


**(b)** 10 inputs, 2 outputs

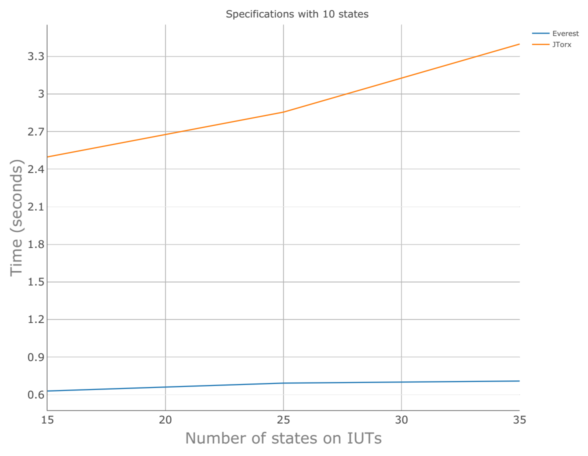
**Figure 8.** Reversing I/O alphabets with **ioco** verdicts

In contrast, regarding verdicts of non-conformance, we see an expressive impact on the verification time when running experiments with the same scenarios where the size of the input and output alphabets are reversed in size. In this case, both tools have taken less processing time for models with 2 inputs and 10 outputs. Everest is $12.73\%$ to $42.86\%$ faster, according to the number of states on IUTs, for models with 2 inputs and 10 outputs (See Figure 9(a)) than when running models with 10 inputs and 2 outputs (See Figure 9(b)). JTorx is around $200\%$ to $352\%$ faster, depending on the IUT size, for the same scenarios. We can observe that the impact over the processing time is very expressive in JTorx for verdicts of non-conformance when we reverse the size of the alphabets.

We remark that in practical applications we usually need more input actions than output actions to specify real-world systems. That is, input alphabets with a large number of ac-



**(a)** 2 inputs, 10 outputs



**(b)** 10 inputs, 2 outputs

**Figure 9.** Reversing I/O alphabets and non-**ioco** verdicts

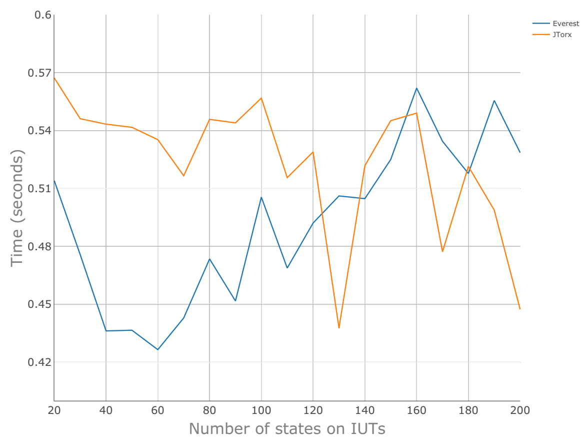
tions can weigh down the performance of JTorx tool. Further, notice that Everest always outperforms JTorx for all scenarios as depicted in all figures.

### 5.1.2 Varying the number of states

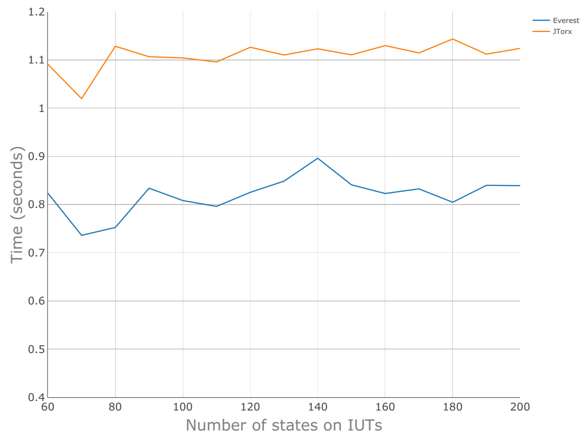
We also performed some experiments varying the number of states (and transitions) to evaluate the tools' scalability. In this case the **RQ** is: *"How does the number of states in specifications and IUTs impact the processing time on checking conformance?"*.

We answer this question running three groups of experiments: (i) specifications with 10 states and IUTs ranging from 20 to 200 states; (ii) specifications with 50 states and IUTs ranging from 60 to 200 states; and (iii) specifications with 100 states and IUTs varying from 110 to 200 states. We remark that all groups of IUT models were increased by 10 states in each group.

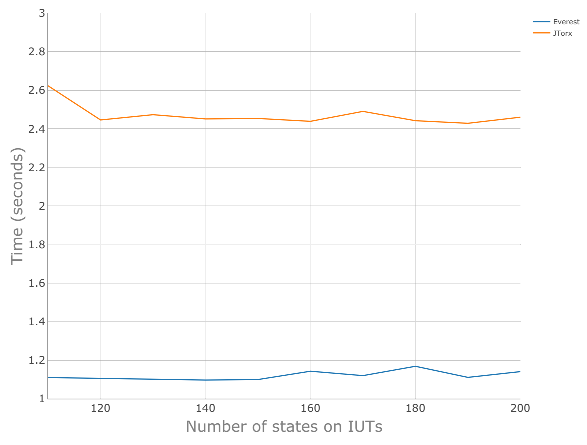
In the experiments with verdicts of conformance, specifications with 10 states and IUTs with up to 120 states, Everest attains a better performance compared to JTorx. JTorx is just slightly better when the IUT models have more than 120 states. See Figure 10a. When running experiments with verdicts of conformance, specifications with 50 and 100 states, and groups of IUTs with up to 200 states, Everest has always outperformed JTorx. See Figures 10b and 10c.



**(a)** Specification with 10 states



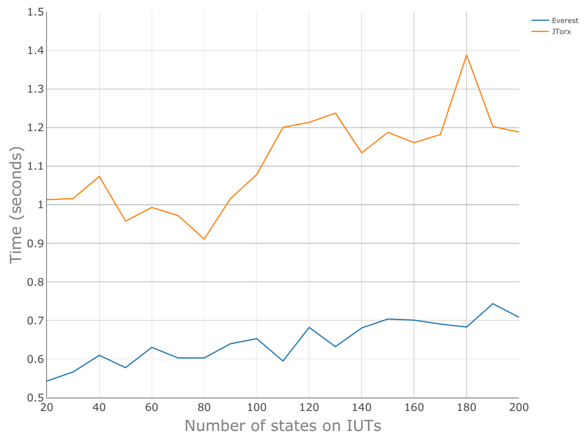
**(b)** Specification with 50 states



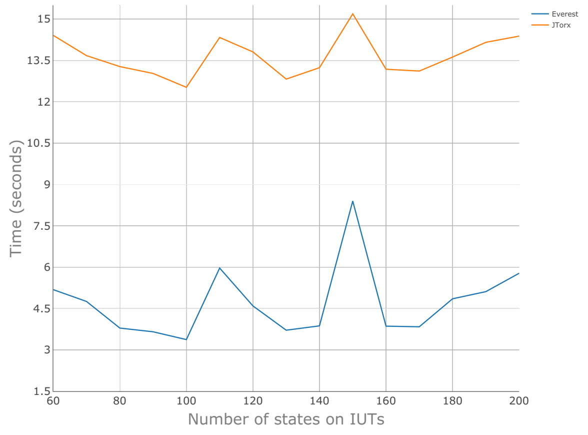
**(c)** Specification with 100 states

**Figure 10.** Varying the number of states and **ioco** verdicts

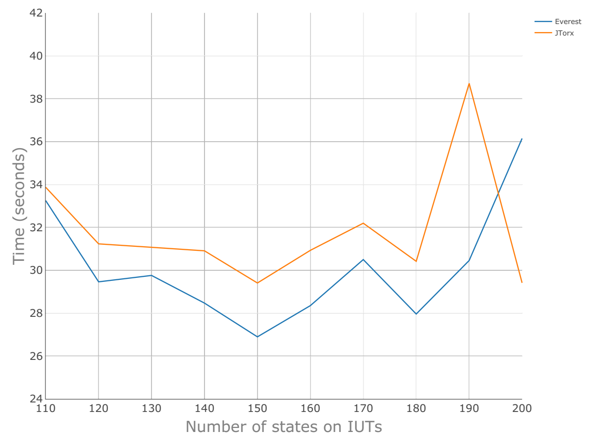
Now we turn into experiments with verdicts of non-conformance, specifications with 10 and 50 states, and IUTs with up to 120 states. We see from Figures 11a and 11b that Everest has always outperformed JTorx for any group of IUTs. JTorx gets a better performance only for IUTs with more than 200 states and specifications with 100 states. See Figure 11c.



**(a)** Specification with 10 states



**(b)** Specification with 50 states



**(c)** Specification with 100 states

**Figure 11.** Varying the number of states and non-**ioco** verdicts

## 5.2 Everest Test Suite Generation

Now we evaluate our tool for test suite generation by running experiments using a more recent approach (Bonifacio and Moura, 2019), where multigraphs must be first constructed to then generate test purposes. We vary the number of states in the specification models and also the bound to be considered over the number of states on the IUTs. We look out to construct distinguishing IUTs from their respective specifications with a certain percentage of modification over the transitions in order to assess different scenarios.

We remark that the experiments on generating test suites were performed using solely Everest due to two main reasons: (i) JTorx implements an *online* strategy where an IUT is always required to run the test generation mechanism; and (ii) JTorx's test generation process finishes at the very first detected fault, so it cannot generate complete test suites.

In the first group of experiments we vary the number of states on specification models together with the number of states to be considered on IUTs; in the second group we generate test purposes over the multigraphs obtained in the first group; and in the third group we run test suites which were extracted from the test purposes of the second group, over IUTs that were generated by modifying the corresponding specification models by a certain percentage.

### 5.2.1 Multigraph generation step

We define the following **RQ** for the multigraph generation step as follows: *"What is the impact on the processing time when generating multigraphs?"*.

To answer this question we vary the number of states on specification models and also the bound $m$ associated with the maximum number of states to be considered on IUT models. Moreover, we consider 5 to 35 states specifications and construct the corresponding multigraphs to get fault models for IUTs with 5 to 55 states. Alphabets were fixed at 5 inputs and 5 outputs and we increase the number of states by 10 for each group of IUTs. Transitions were randomly generated to ensure unbiased results.

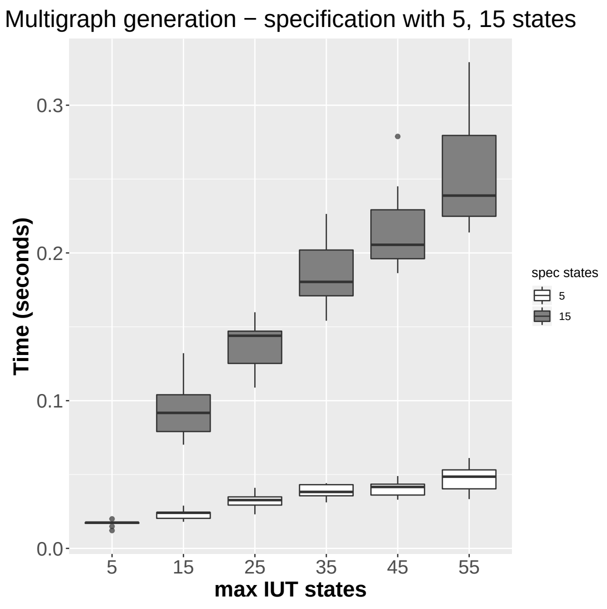
Next we briefly describe all scenarios that are taken into account in the multigraph generation step: (i) specifications with 5 states and $m$ from 5 to 55 states; (ii) specifications with 15 states and $m$ from 15 to 55 states; (iii) specifications with 25 states and $m$ from 25 to 55 states; and (iv) specifications with 35 states and $m$ from 35 to 55 states.

Figure 12 shows that the processing time for generating multigraphs grows, in general, as the number of states also grows on specification and IUT models. We first notice that the median values are lying on the medium of the boxes, which means that as the size of the models grows we also observe a well-behaved growth of the processing time.

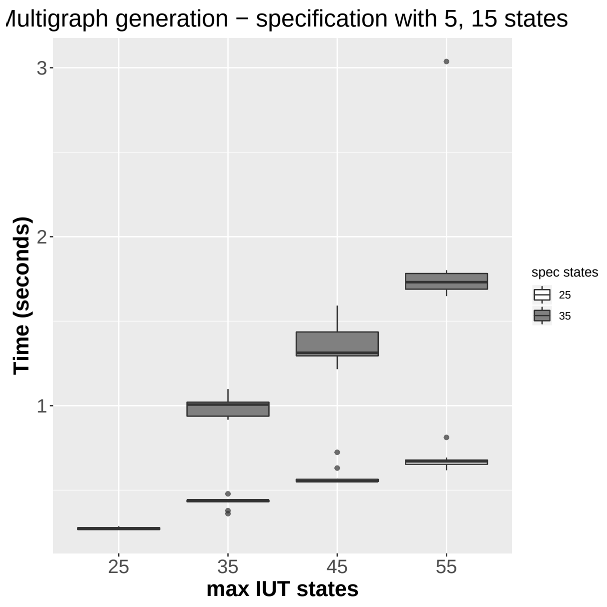
We particularly see in Figure 12a that the multigraph construction for specifications with 5 states and $m = 35$ takes $0.038$ seconds, whereas the construction for $m = 55$ takes $0.047$ seconds. So the processing time rose by $23.68\%$. Similarly, we observe that the construction process for specifications with 15 states takes $0.186$ seconds, with $m = 35$, and takes $0.253$ seconds, with $m = 55$. In this case, the processing time rose by $36.02\%$.

Taking specifications with 25 states and $m = 35$ the time consumption of the multigraph construction is $0.428$ seconds, and for $m = 55$ it takes $0.676$ seconds, as we can see in Figure 12b. The processing time rose by $57.94\%$. In the last group, we take specifications with 35 states and $m = 35$, resulting in a time consumption of $0.994$ seconds, while it takes $1.867$ seconds with $m = 55$. Here the processing time rose by $87.82\%$.

Notice that the multigraph generation with $m = 35$ is 46 times faster for specification models with 5 states than specifications with 35 states. Likewise the construction with



**(a)** Specifications with 5 and 15 states



**(b)** Specifications with 25 and 35 states

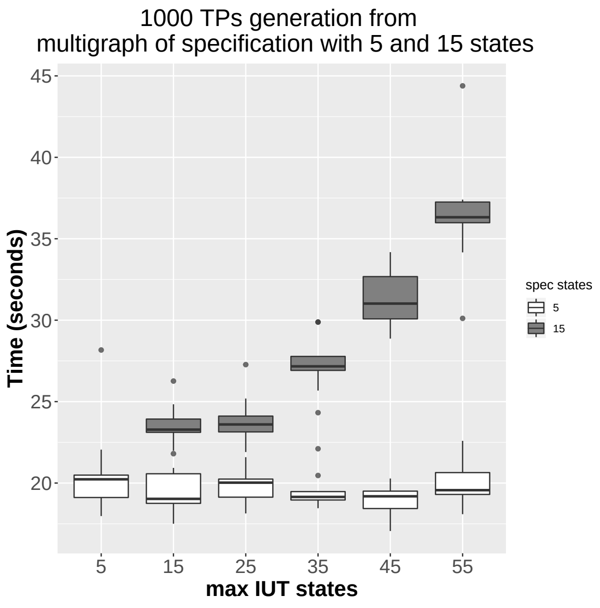
**Figure 12.** Multigraph generation

$m = 55$ is about 26 times faster for specification models with 5 states than specifications with 55 states. Therefore we can conclude that the performance of the multigraph generation decreases as the number of states on specifications and $m$ increase. But the most important issue is that the processing time is not meaningfully affected, *i.e.*, the processing time does not substantially increase as the number of states rises.

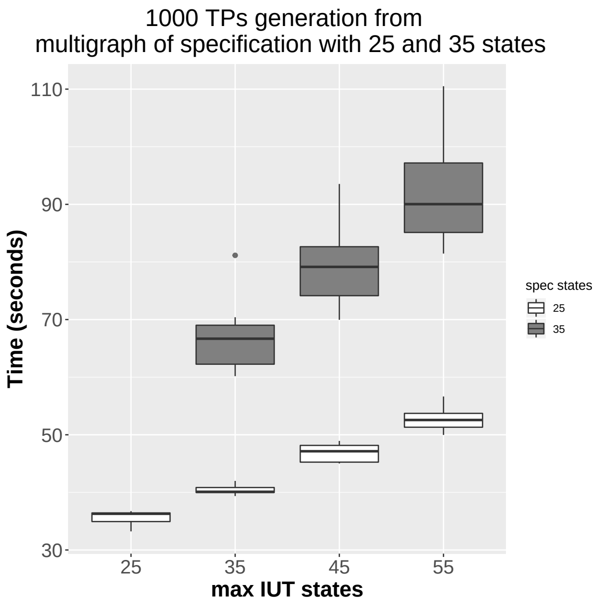
### 5.2.2 Test purpose generation process

Now we turn into the TP generation step based on multigraphs that have been generated in the previous experiments. The associated **RQ**, in this case, is: *"How the TP generation is impacted w.r.t. the processing time when we take multigraphs that have been generated by varying the number of states from the corresponding specifications and also varying the number of states on IUT models?"*.

Here we get multigraphs associated to specifications with 5 to 35 states, and vary $m$ from 5 to 55. We fixed the number of TPs to be generated at 1000. Figure 13 shows that the test generation process takes much more time compared to the multigraph generation step.



**(a)** Specifications with 5 and 15 states



**(b)** Specifications with 25 and 35 states

**Figure 13.** TP generation

From Figure 13a we see that the processing time is more uniform for specifications with 5 states no matter we vary $m$. When the number of states grows Figure 13b shows that the processing time of the TP generation grows fast as $m$ increases. The processing time for specifications with 35 states and $m = 35$ takes $66.82$ seconds whereas using $m = 55$ it takes $91.67$ seconds. So we see that the rate rose by $37.19\%$. Considering specifications with 15 states and, respectively,

$m = 35$ and $m = 55$, the rate rose by $33.75\%$, whereas for specifications with 25 states and, $m = 35$ and $m = 55$, respectively, the rate rose by $30.48\%$.

### 5.2.3 Running test suites

In the last group of experiments we evaluate the processing time on running test suites. Here the **RQ** is given as follows: *"What is the impact on the processing time when running test suites over IUTs with 1%, 2% and 4% of modification w.r.t. the specifications which were used to generate the corresponding multigraphs?"*.

To answer this question we have taken test suites from TPs that were generated for specifications with 15 and 25 states. We fixed $m = n$, that is, the number of states to be considered on IUT models is the same to the number of states in the specification models.

Figure 14 shows the processing time according to the modification rate over the IUTs. The time consumption of the test

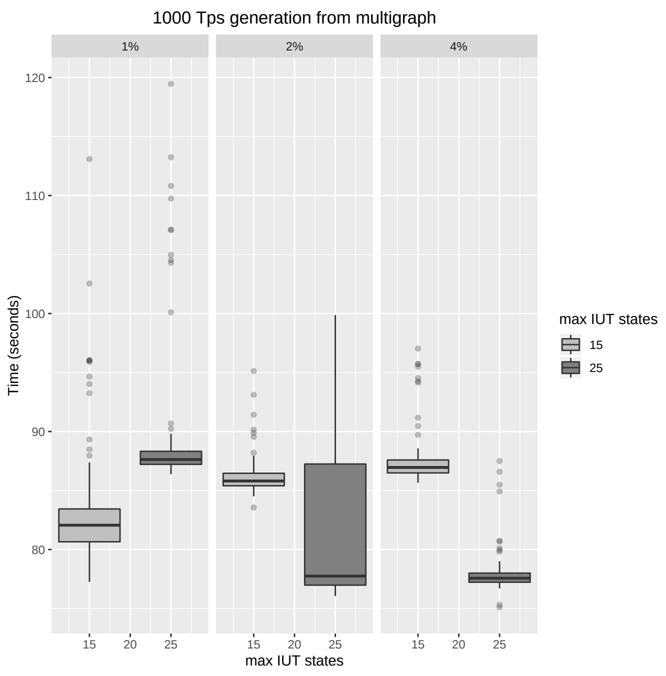


**Figure 14.** Test run

run over IUTs with $1\%$ of modification takes $83.03$ seconds with $m = 15$ and $89.78$ seconds with $m = 25$. Regarding IUTs with $2\%$ of modification, the process takes $86.19$ seconds with $m = 15$ and $80.83$ seconds with $m = 25$. Finally, for IUTs with $4\%$ of modification, the test run takes $87.54$ seconds with $m = 15$ and $77.83$ seconds with $m = 25$.

We see that the processing time of test runs over IUTs with $m = 15$ and $1\%$ of modification is $5.15\%$ faster than the test run over IUTs with $4\%$ of modification. If we consider $m = 25$ then the test run over IUTs with $4\%$ of modification is $13.31\%$ faster than over IUTs with $1\%$ of modification.

## 5.3 Threats to Validity

We list some aspects that may arise as a threat to the validity of the experiments. First we have to report a substantial intricacy to obtain the JTorx tool. Several libraries were missing and we did not have full access to the source code. We have had access only to a binary code whereupon we could make some small amendments to adapt and run it from the command line. Had we accordingly compiled and configured both tools they could be appropriately set up under the same conditions, and so the time consumption could be more easily and precisely obtained on running the experiments.

The computational resource where the experiments were run may also be a threat. We have run all experiments in a general-purpose machine whose results might be biased in some way. But we remark that both tools have run all experiments under the same conditions.

Another threat is related to the random generation of the models. Although we have randomly generated all models in order to avoid biases in the process, we had to guarantee some properties on specific classes of experiments. For instance, in some groups of experiments, we had to construct IUTs that were in conformance to their corresponding specification while in other groups we had to guarantee a certain rate of modification over the IUTs to get verdicts of non-conformance. So the results might have somehow be biased by all these extra checking tasks.

We also list as a threat, those properties that must be guaranteed over the models following restrictions imposed by JTorx. We see that the size of alphabets, the number of states, and transitions of the specification and IUT models are modified from the original models to secure such properties. So we cannot make any claim about the similarity between these modified models and the original ones w.r.t. their behaviors.

## 6 Conclusion

Conformance checking and test suite generation are important activities to improve the reliability of developing reactive systems. In this work we have presented an automatic testing tool for checking conformance and generating test suites for IOLTS models.

We have implemented the classical ioco relation and the more general approach based on regular languages. The latter, and consequently Everest tool, imposes few, if any, restrictions over the models and allows a wider range of fault models described by regular languages when checking conformance. Several works have dealt with ioco theory and its variations. However, we are not aware of any other tool that implements a different notion of conformance, such as the language-based conformance. Further our tool has implemented a complete black-box test suite generation using the notion of test purposes for certain classes of fault models.

We described some case studies to probe both tools and their functionalities in practice. We then could observe from a comparative analysis that Everest provides a wider range of testing scenarios since it was able to detect faults, using the language-based approach, that were not detected by JTorx, using the ioco theory. The effectiveness of our test suite generation method is also evaluated in black-box scenarios.

We also offered practical experiments of conformance checking to compare the performance of Everest against the JTorx. We can see that Everest outperforms JTorx in most scenarios unless for those where the structure of IUT models are quite different from the corresponding specifications. Hence we remark that although Everest implements a more general conformance relation the time consumption has not been impacted on checking runs. Also we observed from the results that Everest has a more stable behavior w.r.t. the processing time even for IUT models with quite a different number of states. We also performed experiments of test suite generation and test run using Everest tool. Our tool was able to handle specifications and implementation candidates with a reasonable number of states as seen in the experiments.

The main contribution of this work is our practical tool that can check conformance based on different relations and can generate test suites in a black-box setting. Moreover, we have presented some case studies, a comparative analysis, and also practical experiments to evaluate and compare our tool.

An extension on the current version of Everest is underway with a new module to allow conformance checking, test suite generation and test run in a batch mode, *i.e.*, it will be able to automatically test several IUT models at once. As future directions, we intend to improve our strategies and algorithms to generate test suites and run test cases more efficiently.

## References

Belinfante, A. (2010a). Jtorx: A tool for on-line model-driven test derivation and execution. In Esparza, J. and Majumdar, R., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, Lecture Notes in Computer Science, pages 266–270. Springer.

Belinfante, A. (2010b). Jtorx: A tool for on-line model-driven test derivation and execution. In Esparza, J. and Majumdar, R., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270, Berlin, Heidelberg. Springer Berlin Heidelberg.

Belinfante, A. (2014). *JTorX: Exploring Model-Based Testing*. Centre for Telematics and Information Technology (CTIT), Netherlands. IPA Dissertation series no. 2014-09.

Bonifacio, A. L. and Moura, A. V. (2019). Complete test suites for input/output systems. *CoRR*, abs/1902.10278. Accessed on: 2019-06.

Calamé, J. (2005). Specification-based test generation with tgv. *Software Engineering Notes*.

Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2002). STG: A Symbolic Test Generation Tool. In Katoen, J.-P. and Stevens, P., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–475, Berlin, Heidelberg. Springer Berlin Heidelberg.

de Vries, R. (2001). Towards formal test purposes. In Tretmans, G. and Brinksma, H., editors, *Formal Approaches to Testing of Software 2001 (FATES'01) - Volume NS-01-4*, BRICS Notes Series, pages 61–76, Aarhus, Denkmark.

Gomes, C. S. and Bonifacio, A. L. (2019). Automatically checking conformance on asynchronous reactive systems. In *The Fourteenth International Conference on Software Engineering Advances*, pages 17–23.

Jard, C. and Jéron, T. (2005). Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315. Accessed on: 2019-08.

Larsen, K. G., Mikucionis, M., Nielsen, B., and Skou, A. (2005). Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 299–306. ACM.

Mark Utting, B. L. (2007). *practical model-based testing a tools approach*. Elsevier, 1nd edition.

Marsso, L., Mateescu, R., and Serwe, W. (2018). Testor: A modular tool for on-the-fly conformance test case generation. In Beyer, D. and Huisman, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 211–228, Cham. Springer International Publishing.

Mostowski, W., Poll, E., Schmaltz, J., Tretmans, J., and Wichers Schreur, R. (2009). Model-Based Testing of Electronic Passports. In Alpuente, M., Cook, B., and Joubert, C., editors, *Formal Methods for Industrial Critical Systems*, pages 207–209, Berlin, Heidelberg. Springer Berlin Heidelberg.

Naik, K. and Tripathy, P. (2018). *Software Testing and Quality Assurance: Theory and Practice*. Wiley Publishing, 2nd edition.

Roehm, H., Oehlerking, J., Woehrle, M., and Althoff, M. (2016). Reachset conformance testing of hybrid automata. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, HSCC '16, pages 277–286, New York, NY, USA. ACM.

Simão, A. d. S. and Petrenko, A. (2014). 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.*, pages 56–70. Accessed on: 2019-07.

Sipser, M. (2006). *Introduction to the Theory of Computation*. Course Technology, second edition.

Tretmans, J. (1993). A formal approach to conformance testing. In Rafiq, O., editor, *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, 28-30 September, 1993*, volume C-19 of *IFIP Transactions*, pages 257–276. North-Holland.

Tretmans, J. (2008). Model based testing with labelled transition systems. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer.