Automatically Checking Conformance on Asynchronous Reactive Systems

Camila Sonada Gomes*        Adilson Luiz Bonifacio†

Abstract

Software testing is an important issue in software development process to ensure higher quality on the products. Formal methods has been promising on testing reactive systems, specially critical systems, where accuracy is mandatory since 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 most challenges in model-based testing is the conformance checking of asynchronous reactive systems. The aim is to verify if an implementation is in compliance with its respective specification. In this work, we develop a practical tool to check conformance relation between reactive models using a more general theory based on regular languages. The approach, in fact, subsumes the classical $\text{ioco}$ conformance which is also available in our tool. In addition, we present some studies with different scenarios that are applied to practical tools with both notions of conformance.

1 Introduction

Automatic testing tools have been proposed to support the development process of reactive systems that are characterized by the 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 [7, 8, 16]. Input Output Labeled Transition Systems (IOLTSs) [8, 16, 4, 15] are traditional formalisms usually applied to modeling and testing reactive systems.

In model-based testing IOLTS specification can model desirable and undesirable behaviors of an implementation under test (IUT). The aim is finding faults in an IUT according to a certain fault model [14, 7, 5] to show if requirements are or not satisfied regarding its respective system specification. A well-established conformance relation, called $\text{ioco}$ and proposed by Tretmans [14], requires that the outputs produced by an IUT must be also produced by its respective specification following the same behavior. A more recent and general conformance relation, proposed by Bonifacio and Moura [7], 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. JTorx [2], another tool from the literature, also

*Computing Department, University of Londrina, Londrina, Brazil.
†Computing Department, University of Londrina, Londrina, Brazil.
implements a conformance testing verification process, but only based on the classical \textit{ioco} relation. Our tool comprises both the classical \textit{ioco} relation and also the more general conformance based on regular languages. Some practical sceneries are then run to evaluate aspects related to effectiveness and usability of both tools and both conformance theories.

We organize this paper as follows. Section 2 describes the conformance verification methods using regular languages and the \textit{ioco} relation. The practical tool which implements the more general method of conformance checking is presented in Section 3. Some applications and a comparative study are given in Section 4. Section 5 offers some concluding remarks.

2 Conformance Verification

The classical conformance relation \textit{ioco} establishes compliance between IUTs and specifications based on a fault model given by behaviors over a specific domain. Conformance testing verdicts are declared positive over \textit{ioco} relation if an output produced by an IUT is also specified in the specification after applying a sequence of input stimuli to it. In this context, we call by test case a sequence of input stimuli and by path a sequence of states induced by a test case over the formal model. Thus, when we apply a test suite to both a specification and an IUT, if for every test case 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 \cite{16}.

A more general conformance relation has been proposed by Bonifacio and Moura \cite{7}, where the fault model is defined by regular languages. Basically, desirable and undesirable behaviors are specified by regular expressions, $D$ and $F$, respectively. Given an implementation $I$, a specification $S$, and regular expressions $D$ and $F$, we need to check if $otr(I) \cap ([D \cap \overline{otr}(S)) \cup (F \cap otr(S))] = \emptyset$, where $otr(S)$ and $otr(I)$ denote the behaviors of $S$ and $I$, respectively, and $\overline{otr}(S)$ gives the complement of $otr(S)$. We declare that an implementation $I$ is in conformance to a specification $S$, if there is no test case of the test suite $T = ([D \cap \overline{otr}(S)) \cup (F \cap otr(S))]$ that is also a behavior of $T$ \cite{7}.

As stated by the authors \cite{7} the \textit{ioco} relation is, in fact, a particular case of the language-based conformance verification when the pair of languages $(D, F)$ is settled by $F = \emptyset \ e D = otr(S) L_U$, where $L_U$ is the set of outputs. That is, $I \ ioco \ S$ if and only if, $I \ conf_{D, \emptyset} \ S$, and the more general approach gives the \textit{ioco} relation using the language-based algorithm \cite{7}.

3 A Testing Tool for Reactive Systems

In this work we have developed a testing tool to automatically checking conformance of reactive systems modeled by IOLTS models. Our tool supports the more general notion of conformance based on regular languages and also the classical notion of \textit{ioco} relation. \textit{Eve}rs\textit{t} \cite{11} has been developed in Java \cite{12} using \textit{Swing} \cite{12} library to provide a yielding and friendly usability experience through a graphical interface.

Some features available on Everest tool are: (i) check conformance based on regular languages and \textit{ioco} relation; (ii) describe desirable and undesirable behaviors using regular expressions; (iii) specify formal models in Aldebaran format \cite{1}; (iv) generate test suites when non-conformance verdicts are obtained; (v) provide paths induced by test cases; and (vi) allow the graphical representation of the formal models.

\footnote{\textit{EVEREST} – \textit{conformancE Verification on \textit{t}Esting ReactivE Systems}}

\footnote{Available in \url{https://everest-tool.github.io/everest-site}}
The Everest tool development was arranged into four main modules as shown 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 computing results are represented by ellipses.

The View module implements an intuitive graphical interface with three different views: configuration; ioco conformance; and language-based conformance. In the first view we set the specification and implementation models, the model type (LTS or IOLTS) under test, and the partition into input and output labels in case of IOLTS models. In the ioco and language-based conformance views we run both verification processes and also graphically inspect the models to ascertain the configuration information. After finishing the testing process if the IUT does not conform to the specification then a negative verdict is displayed together with the associated test cases. Otherwise, the tool displays a positive verdict of conformance. Note that we can provide regular expressions to represent the desirable behaviors and the fault properties in the language-based conformance view.

The IUT and specification models are validated by the Parser module where data structures are constructed to internally represent them. 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 the regular languages. We remark that ordinary LTS models can be checked using the language-based conformance notion 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 Conformance Verification module provides all necessary operations over regular languages such as union, intersection, and complement. This module also constructs the finite automaton that represents the complete test suite. Next the module performs the conformance verification process.
4 Practical Application

In this section we describe some practical testing scenarios applied to the Everest tool and briefly compare to JTorx tool. Let $S$ be the IOLTS specification of Figure 2a and let $R$ and $Q$ be implementation candidates as depicted in Figures 2b and 2c. Also let $L_I = \{a, b\}$ and $L_U = \{x\}$ be the input and output alphabets, respectively. All models here are deterministic [9, 10] but we remark that our tool also deals with nondeterministic models.

In the first scenario we check if IUT $R$ conforms to specification $S$. Everest has returned a non-conformance verdict using $\text{ioco}$ relation and generated the test suite $\{b, \text{aa, ba, aaa, ab, ax, abb, axb}\}$. The subset of test cases $\{b, \text{aa, ba, aaa}\}$ induces paths from $s_0$ to $s_3$ in $S$ and from $q_0$ to $q_3$ in $R$, where the output $x$ is produced by $R$ but $S$ does not. Note that $s_3$ in $S$ is a quiescent state whence no output is defined on it. The subset $\{\text{ab, ax, abb, axb}\}$ induces paths to state $s_2$ in $S$ and $q_2$ in $R$. In this case, the output $\delta$ is produced by IUT $R$ whereas $S$ produces $x$, where $\delta$ is the quiescent label. That is, a fault is detected according to $\text{ioco}$ relation. We note that both tools modify the formal models in the presence of quiescence by adding self-loops labeled with $\delta$ [14].

The same scenario has been also applied to JTorx tool, resulting in the same verdict, as expected, but it generates the test suite $\{b, \text{ax, ab}\}$. Notice that the test suite generated by JTorx is a subset of that test suite generated by Everest. That is, Everest shows all test cases, and associated paths, related to a single fault according to a transition cover criteria over the specification whereas JTorx returns only one test case per fault. All this information may be useful and aid the tester in the fault mitigation process.

In the second scenario, checking the IUT $Q$ against $S$, the language-based conformance verification was able to detect a fault which was not detected by the $\text{ioco}$ conformance relation. 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 an external stimulus $a$ followed by an output $x$ produced in response. Since the only complete test suite is given by $[(D \cap \text{otr}(S)) \cup (F \cap \text{otr}(S))]$ and $F \cap \text{otr}(S) = \emptyset$, so we check the condition $D \cap \overline{\text{otr}(S)} \neq \emptyset$, i.e., a fault is detected when behaviors of $D$ are not present in $S$. Everest then results in a verdict of non-conformance and produces the test suite $\{\text{ababax, abaabax}\}$ reaching a fault that is not detected by JTorx using the $\text{ioco}$ relation.
5 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 have required high costs and resources in the software development. Therefore automation of the testing activity has become essential in this process. Several studies have addressed the testing of reactive systems [7, 8, 16, 3, 6], specially the conformance checking between IUTs and specifications based on appropriate formalisms to guarantee more reliability.

In this work we have developed an automatic tool for checking conformance on asynchronous reactive systems. We have implemented the more general relation based on regular languages and also the classical \textit{ioco} theory. One could observe that the Everest conformance verification process by means regular languages is more effective than the JTorx conformance checking. In the practical applications we could see that JTorx yielded a verdict of conformance whereas Everest could find a fault for the same scenario. We also remark that the main contribution of this work is the tool development together with the designed algorithms, and providing an intuitive graphical interface either for experts in the research area or for beginners with no specific knowledge over the conformance theories.

A new module of Everest tool is already being developed to provide a test suite generation in a black-box setting. We also intend to perform more experiments and comparative studies with similar tools from the literature in order to give a more precise analysis related to the conformance checking issue, specially, with respect to the usability and performance of these tools.

References

[1] AUT manual page. https://cadp.inria.fr/man/aut.html. Acessado em: 2019-02-28.

[2] JTorX a tool for model-based testing. https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/. Acessado em: 2018-03-23.

[3] Bernhard K. Aichernig, Elisabeth Jbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. \textit{Science of Computer Programming}, 97:383 – 404, 2015. Special Issue: Selected Papers from the 12th International Conference on Quality Software (QSIC 2012).

[4] Bernhard K. Aichernig and Martin Tappler. Symbolic input-output conformance checking for model-based mutation testing. \textit{Electronic Notes in Theoretical Computer Science}, 320:3 – 19, 2016. The 1st workshop on Uses of Symbolic Execution (USE).

[5] Bernhard K. Aichernig, Martin Weighhofer, and Franz Wotawa. Improving fault-based conformance testing. \textit{Electronic Notes in Theoretical Computer Science}, 220(1):63 – 77, 2008. Proceedings of the Fourth Workshop on Model Based Testing (MBT 2008).

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. \textit{Journal of Systems and Software}, 86(8):1978 – 2001, 2013.
[7] Adilson Luiz Bonifacio and Arnaldo Vieira Moura. Complete test suites for input/output systems, 2019.

[8] Adenilso da Silva Simão and Alexandre 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., pages 56–70, 2014.

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

[10] Bruno Legeard Mark Utting. practical model-based testing a tools approach. Elsevier, 1nd edition, 2007.

[11] Oracle. Java se development kit 8. http://www.oracle.com/technetwork/pt/java/javase/. Accessed: 2019-01-30.

[12] Oracle. Package javax swing. https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html. Accessed: 2019-01-31.

[13] Michael Sipser. Introduction to the Theory of Computation. Course Technology, second edition, 2006.

[14] G.J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. Number TR-CTIT-96-26 in CTIT technical report series. Centre for Telematics and Information Technology (CTIT), Netherlands, 1996. CTIT Technical Report Series 96-26.

[15] Jan Tretmans. Testing concurrent systems: A formal approach. In Jos C. M. Baeten and Sjouke Mauw, editors, CONCUR’99 Concurrency Theory, pages 46–65, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[16] Jan Tretmans. Model Based Testing with Labelled Transition Systems, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.