JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution

Axel Belinfante

Formal Methods and Tools, University of Twente, The Netherlands
Axel.Belinfante@cs.utwente.nl

Abstract. We introduce JTorX, a tool for model-driven test derivation and execution, based on the ioco theory. This theory, originally presented in [12], has been refined in [13] with test-cases that are input-enabled. For models with underspecified traces [3] introduced uioco.

JTorX improves over its predecessor TorX [14] by using uioco and this newer ioco theory. By being much easier to deploy, due to improved installation, configuration and usage. And by integrating additional functionality, next to testing: checking for (u)ioco between models [6]; checking for underspecified traces in a model; interactive or guided simulation of a model. This makes JTorX an excellent vehicle for educational purposes in courses on model-based testing, as experience has shown – and its usefulness is not limited to education, as experience has shown too.

1 Introduction

Ten years ago we presented TorX, a tool for model-based testing. Its main focus was on *on line* test derivation and execution, i.e. a test is derived on demand while it is being executed (for *off line* execution a test case was treated as a special kind of model; only much later TorX was extended with (experimental) *off line* test derivation). As we wrote in [1], important features of TorX are flexibility and openness. However, it turned out that in our attempt to obtain these features we sacrificed ease of deployment, in particular ease of configuration, and, on Windows, ease of installation. For case studies that was not an issue because 1) usually one can choose where to run them (on a Unix system), 2) typically they were done by TorX-experts, and 3) the configuration overhead was small compared to the overall effort of setting up a case study anyway. However, it was an issue for transfer of the tool to non-TorX-experts, like students who have to install and use the tool to do tool-based exercises, or staff members who want to use it to show the idea of model-based testing.

JTorX is our answer to this problem. JTorX is a re-implementation of the main functionality of TorX in Java. As a consequence, installation of JTorX is rather simple, also on Windows. Configuration is much simpler than in TorX, because all of it can be done via the JTorX Graphical User Interface (GUI).

Not only is JTorX easier to deploy, we also used the opportunity to catch up with theoretical progress, and to add features that are helpful for education and for quick impromptu demonstrations. These are discussed in the next section.

JTorX is, under BSD-style license, available for free at [17].

J. Esparza and R. Majumdar (Eds.): TACAS 2010, LNCS 6015, pp. 266–270, 2010.
© Springer-Verlag Berlin Heidelberg 2010

2 Features

Catching up with theoretical progress led to the following. The initial design of TorX, based on the *ioco* theory of [12], made use of the fact that in that theory test cases are non input-enabled (once the tester has decided to apply a stimulus it will not look at output that might be produced by the system under test (SUT) until the stimulus has been applied). This was already revised when we started to experiment with timed testing [5]. The design of JTorX is based on the refined *ioco* theory of [13] in which test cases are input-enabled. In addition, JTorX allows testing for *uioco* (introduced in [3]), a weaker relation than *ioco* developed for models that contain underspecified traces. In JTorX the ability to support off line test derivation has been taken into account from the start.

The following features were added in JTorX to ease educational use, next to the functionality for on line test derivation and execution: a checker to find underspecified traces in a given model; a checker that checks (instead of tests) whether two models are (u)ioco-related [6]; a simulator for manual exploration of a model (or *suspension automaton*, see Section 3) or guided simulation of a given trace, e.g. produced by one of the checkers. The test run can be guided, using such trace, or a test purpose. To access models, JTorX has built-in support for: GRAPHML [8] (to allow the use of graph editor yEd [15] to draw a model as automaton), the Aldebaran (.aut) file format, the Jararaca [16] file format (to allow a regular expression-style specification of traces to guide a test run), and the TorX Explorer protocol to allow model access via the mcrl2 [9], LTSmin [4] and CADP [7] tool environments. To connect to a SUT JTorX has built-in support for: use of a simulated model; use of a real program that communicates using labels of the model, either on its standard input and output, or over a single TCP connection; use of the TorX Adapter protocol (for backwards compatibility). Each test run is reported in a (text) log shown in the GUI (and as in TorX visualized in a dynamically updated *message sequence chart*). During a test run (as in TorX), and during simulation, progress through models and suspension automaton is visualized in dynamically updated automaton viewers.

3 Architecture

The flexibility and openness of TorX were obtained by having a modular tool architecture. JTorX inherits this architecture (but not the deployment issues, by having a different implementation). A typical (J)TorX configuration, depicted in Fig. 1, contains at least the following components. (Components that are only used in guided test runs appear dotted in this figure.) An Explorer provides uniform access to the (labelled transition system) state space of the Model (or Test Purpose, in case of a guided test run). A Primer provides access to the *suspension automaton* (see [13]) of the Model (or Test Purpose) accessed via its Explorer, i.e. it determinizes and marks *quiescent* states (in which the SUT is expected to stay silent) with δ-labelled selfloops. It does this on demand. To avoid storing states unnecessarily Primer and Driver tell their resp. Explorer

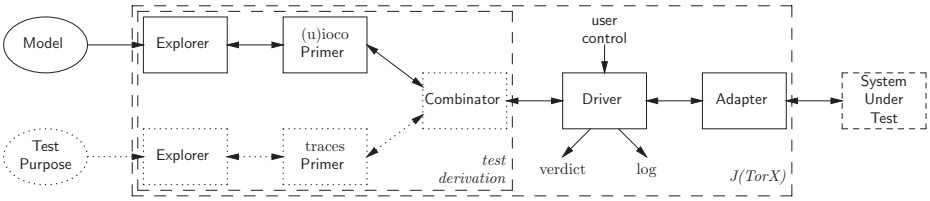

Fig. 1. Tool components of a typical (J)TorX configuration. Items TP, Explorer, Primer and Combinator in the dotted boxes are only present in a guided test run.

and Primer which states (reached by stimulus or observation not in the test run) to forget. An Adapter provides uniform access to the SUT. The Driver controls a test run, and decides whether to obtain and apply a stimulus, whether to obtain and check an observation, or to stop the test run. Additional components, like the Combinator, are used e.g. to guide a test run using a Test Purpose. Typically, an Explorer is modeling-formalism dependent, but model-independent; an Adapter is model-dependent, and specific for a particular (family of) SUT. The other tool components are model-, formalism- and SUT-independent.

4 Usage

At four universities students have used JTorX in courses on testing techniques, to compare models, and to test a real program w.r.t. a model that they developed themselves. Doing these exercises gave the students a deeper understanding of the ioco theory and its test derivation algorithm, and allowed them to experience model-based testing in practice. For the tutors, the use of JTorX greatly reduced the effort needed to set up the exercise class, compared to the use of TorX in previous years. Moreover, it encouraged developing more elaborate exercises – for example, testing of a real program (now facilitated by JTorX built-in standard i/o Adapter) was not done in previous years. For the students, JTorX clearly provided a better user experience – with TorX there typical were complaints, e.g. about the GUI, but with JTorX there were none. As a result, JTorX will continue to be used in these courses.

JTorX found a real, unintended error in a Java program developed as SUT for one of the courses. The program occasionally lost inputs, because its input handling was initialized *inside* (instead of before) its main input-processing loop.

In an internship a student used Unit Testing and then JTorX to test the program he developed. JTorX found five errors, some of which rather subtle – these might not have been found without JTorX, even when the time invested in model-based testing would have been spent on manual testing instead [11].

5 Future Work

We foresee improvements to JTorX in two directions: improvement of the user experience e.g. by professionalization of the user interface, and extension of the

functionality. Implementation in JTorX of TorX' ability to deal with parameterized action labels will enable use of its real-time Explorer [5] and its Explorer for Promela models. Also, connection to TorXakis [10] is planned.

6 Related Work

We are aware of existing tools for model-based testing like those discussed in [2], but to our knowledge, none of them posseses a similar suitability for education.

Acknowledgements

Our gratitude goes to Mark Timmer for helping with the testing techniques practical exercises at University of Twente, to Jan Tretmans for trying JTorX and providing feedback, and to Jaco van de Pol for feedback on this article.

References

1. Belinfante, A., et al.: Formal test automation: A simple experiment. In: 12^{th} Int. Workshop on Testing of Communicating Systems, pp. 179–196. Kluwer, Dordrecht (1999)
2. Belinfante, A.F.E., Frantzen, L., Schallhart, C.: Tools for test case generation. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 391–438. Springer, Heidelberg (2005)
3. van der Bijl, H.M., Rensink, A., Tretmans, J.: Compositional testing with ioco. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
4. Blom, S.C.C., van de Pol, J.C., Weber, M.: Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, Centre for Telematics and Information Technology, University of Twente, Enschede (2009)
5. Bohnenkamp, H.C., Belinfante, A.F.E.: Timed testing with TorX. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 173–188. Springer, Heidelberg (2005)
6. Frantzen, L.: iocoChecker (2008), `http://www.cs.ru.nl/~lf/tools/iocochecker`
7. Garavel, H., et al.: Cadp 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
8. GraphML work group: GraphML file format, `http://graphml.graphdrawing.org`
9. Groote, J.F., et al.: The mcrl2 toolset. In: Proc. International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008 (2008)
10. Mostowski, W., Poll, E., Schmaltz, J., Tretmans, J., Schreur, R.W.: Model-based testing of electronic passports. In: Alpuente, M. (ed.) FMICS 2009. LNCS, vol. 5825, pp. 207–209. Springer, Heidelberg (2009)
11. Sijtema, M.: Developing XBus2 — a software bus, that is maintainable, testable and backwards-compatible. Internship Report, University of Twente (2009)
12. Tretmans, J.: Test generation with inputs, outputs, and repetitive quiescence. Software - Concepts and Tools 17(3) (1996)

13. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
14. Tretmans, J., Brinksma, H.: TorX: Automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, Nuremberg, Germany, December 2003, pp. 13–43 (2003)
15. yWorks: yEd, `http://www.yworks.com/en/products_yed_about.html`
16. Jararaca manual, `http://fmt.cs.utwente.nl/tools/torx/jararaca.1.html`
17. JTorX website, `http://fmt.cs.utwente.nl/tools/jtorx/`