Combining Testing and Runtime Verification

Christian Colombo
Department of Computer Science
University of Malta
Email: christian.colombo@um.edu.mt

***Abstract*—Testing and runtime verification are intimately related: runtime verification enables testing of systems beyond their deployment by monitoring them under normal use while testing is not only concerned with monitoring the behaviour of systems but also generating test cases which are able to sufficiently cover their behaviour. Given this link between testing and runtime verification, one is surprised to find that in the literature the two have not been well studied in each other's context. Below we outline three ways in which this can be done: one where testing can be used to support runtime verification, another where the two techniques can be used together in a single tool, and a third approach where runtime verification can be used to support testing.**

I. Introduction

To date, testing is by far the most commonly used technique to check software correctness. In essence, testing attempts to generate a number of test cases and checks that the outcome of each test case is as expected. While this technique is highly effective in uncovering bugs, it cannot guarantee that the tested system will behave correctly under all circumstances. Typically, testing is only employed during software development; meaning that software has no safety nets during runtime. Conversely, runtime verification techniques are typically used during runtime to provide extra correctness checks but little effort has been done to integrate it with the software development life cycle. For this reason little or no use of it is made in contemporary industry.

At a closer look, the two techniques — testing and runtime verification — are intimately linked: runtime verification enables the checking of a system's behaviour during runtime by listening to system events, while testing testing is concerned with generating an adequate number of test cases whose behaviour is verified against an oracle. Thus, in summary one could loosely define runtime verification as *event elicitation + behaviour checking*, and testing to be *test case creation + behaviour checking*.

Through this brief introduction of testing and runtime verification one can quickly note that behavoiur checking is common to both. Given this overlap between two verification techniques, one is surprised to find that in the literature the two have not been well studied in each other's context. In this paper, we outline three ways in which this can be done: one where testing can be used to support runtime verification, another where the two techniques can be used together in a single tool, and a third approach where runtime verification can be used to support testing.

II. A Testing Framework for Runtime Verification Tools

Like any piece of software, runtime verification tools which generate monitoring code from formal specifications have to be adequately tested, particularly so because of their use to assure other software. State-of-the-art runtime verification tools such as Java-MOP [1] and tracematches [2] have been tested on the DaCapo benchmark [3]. However, the kind of properties which have been monitored are rather low level contrasting with our experience with industrial partners who seem more interested in checking for higher level properties (such as the ones presented in [4], [5]). Whilst we had the chance to test our tool Larva [6] on industrial case studies, such case studies are usually available for small periods of time and in limited ways due to privacy concerns. Relying solely on such case studies can be detrimental for the development of new systems which need substantial testing and analysis before being of any use.

For this reason, we aim to develop a testing framework which would provide a highly configurable mock transaction system to enable thorough validation of systems which interact with it. Such a testing framework would enable a user to easily mock different kinds of transactions, usage patterns in the system, etc. without issues of privacy, and enabling repeatability of results. Although not a replacement of industrial case studies, this would enable better scientific evaluation of runtime verification systems. This testing framework should have the following main aspects:

- **Language** The testing framework should provide an easy to use specification language which enables a user to specify:
  - Sequences of events which the mock system should be able to generate.
  - The time delays between such events, possibly varying randomly.
  - Whether or not these event sequences may be run concurrently or not.

  A mock system with these aspects would enable the user to test the detection capabilities of the monitoring system under test.
- **Profiling** The test framework should provide inbuilt means of measuring the processing and memory overheads incurred by the system. This would facilitate the measurement of the overheads induced by the monitoring system under test.
- **Repeatability** If unexpected behaviour is exhibited by the monitoring system while being testing on the mock system, then, being able to repeat the exact same behaviour would enable the user to understand the source of the problem.

III. Combining Testing and Runtime Verification Tools

While testing is still the prevailing approach to ensure software correctness, the use of runtime verification [7] as a form of post-deployment testing is on the rise. Such continuous testing ensures that if bugs occur, they don't go unnoticed. Apart from being complementary, testing and runtime verification have a lot in common: runtime verification of programs requires a formal specification of requirements against which the runs of the program can be verified [8]. Similarly, in model-based testing, checks are written such that on each (model-triggered) execution step, the system state is checked for correctness. Due to this similarity, applying both testing and runtime verification techniques is frequently perceived as duplicating work. Attempts [9] have already been made to integrate the runtime verification tool Larva [6] with QuickCheck [10] by enabling QuickCheck specifications to be compiled into Larva monitors. We plan to continue on this work by integrating the Larva tool with a Java model-based testing tool, ModelJUnit[1]. This time, the idea is to be able to convert a Larva specification into a ModelJUnit model which can be used to automatically generate test cases and check that the system under test is behaving as expected. Effectively, this would enable anyone with a Larva specification to be able to perform both model-based testing and monitoring without extra manual intervention.

While model-based testing is gaining popularity, unit testing is far more commonplace in contemporary industy. Thus, apart from looking at model-based testing for integration with runtime verification, we are also attempting to convert unit tests to monitors. Typically, unit tests include a short sequence of method calls followed by a number of assertions. If a monitor successfully observes the same sequence of method calls at runtime, then the unit test assertions should also hold. This approach might prove useful to introduce runtime monitoring to industry with the press of a button.

IV. Using Runtime Verification for Model-Based Testing Feedback

As software systems grow in size and complexity, it is becoming harder to ensure their correctness. In particular, complex security-intensive systems such as software handling online financial transactions, are particularly difficult to test for robustness due to the difficulty in predicting and mocking all the possible ways in which the system is expected to be used. To automate test case generation and ensure that the tests cover all salient aspects of a system, model-based testing [11], [12] enables the use of a model specification from which test cases are automatically generated. Although successful and growing in popularity, model-based testing is only effective in as much as the model is complete. Sequences of untested user interaction may lead to huge losses for the service-provider , and loss of trust from the users' end if any of these lead to a failure. Since coming up with a model for test case generation is largely a manual process [11], [12], it is virtually impossible to ensure the model is complete — the modeller may not anticipate all the ways in which the system will be used after deployment.

[1]http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/

In this context, we propose to use runtime information to detect incompleteness in the test case generation model: by considering the execution paths the system takes at runtime, a monitor checks whether each path (or a sufficiently similar one) is in the test case generation model. If the monitor detects a path which appeared at runtime but is not in the model, the modeller can be assisted in updating the model accordingly.

V. Conclusion

Given the intimate ways in which software testing and runtime verification are linked, there are several advantages in studying the interaction between them: (*i*) anyone using one of the technologies can be supported to automatically benefit from the other without any extra effort; and (*ii*) each technology provides aspects from which the other technology can benefit. In this paper, we have presented the general idea of ongoing work which attempts to provide (*i*) by combining existing tools to provide both testing and runtime verification capabilities; and (*ii*) by improving the quality of runtime verification tools through testing techniques, and improving testing coverage analysis through runtime information harvested through runtime monitors.

References

[1] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *JSTTT*, 2011, to appear.

[2] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with tracematches," *J. Log. Comput.*, vol. 20, no. 3, pp. 707–723, 2010.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: java benchmarking development and analysis," *SIGPLAN Not.*, vol. 41, no. 10, pp. 169–190, 2006.

[4] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *FMICS*, ser. LNCS, vol. 5596, 2008, pp. 135–149.

[5] C. Colombo, G. J. Pace, and P. Abela, "Compensation-aware runtime monitoring," in *RV*, ser. LNCS, vol. 6418, 2010, pp. 214–228.

[6] C. Colombo, G. J. Pace, and G. Schneider, "Larva — safer monitoring of real-time java programs (tool paper)," in *SEFM*, 2009, pp. 33–37.

[7] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., *RV*, ser. LNCS, vol. 6418, 2010.

[8] M. Leucker and C. Schallhart, "A brief account of runtime verification," *JLAP*, vol. 78, no. 5, pp. 293–303, 2009.

[9] K. Falzon, "Combining runtime verification and testing techniques," Master's thesis, 2011.

[10] J. Hughes, "Quickcheck testing for fun and profit," in *Practical Aspects of Declarative Languages*, ser. LNCS, 2007, vol. 4354, pp. 1–32.

[11] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Empirical assessment of software engineering languages and technologies*, ser. WEASELTech '07, 2007, pp. 31–36.

[12] I. K. El-Far and J. A. Whittaker, *Model-Based Software Testing*. John Wiley & Sons, Inc., 2002.