Automatically generating runtime monitors from tests*

Abigail Cauchi[1], Luke Chircop[1], Christian Colombo[1], Adrian Francalanza[1], Mark Micallef[1], and Gordon J. Pace[1]

Department of Computer Science, University of Malta

A large portion of the software development industry relies on testing as the main technique for quality assurance while other techniques which can provide extra guarantees are largely ignored. A case in point is runtime verification [3, 9, 7, 12] which provides assurance that a system's behaviour is correct at runtime. Compared to testing, this technique has the advantage of checking the actual runs of a system rather than a number of representative testcases.

Based on experience [6, 5] with local industry, one of the main reasons for the lack of uptake of runtime verification is the extra effort required to formally specify the correctness criteria to be checked at runtime — runtime verifiers are typically synthesised from formal specifications (e.g., Larva [7], JavaMOP [12]). One potential approach to address this issue is that of using the information available in tests to automatically obtain monitors [8]. The plausibility of this approach lies in the similarity between tests and runtime verifiers: tests drive the system under test and check that the outcome is correct; runtime verifiers also check that the outcome is correct but let the system users drive the system.

Notwithstanding the similarities, the fact that in runtime verification the users drive the system means that, while tests are typically focused on checking very specific behaviour, runtime verification assertions need to be able to handle the general case. This makes it hard to create runtime monitors (which are generic enough to be useful) from tests, in particular due to the following aspects:

- **Input:** The checks of the test may only be applicable to the particular inputs specified in the test. Once the checks are applied in the context of other inputs they may no longer make sense. Conversely, the fewer assumptions on the input the assertion makes, the more useful the assertion is for monitoring purposes.
- **Control flow:** The test assertions may be specific to the control flow as specified in the test's context with the particular ordering of the methods and the test setup immediately preceding it.
- **Data flow:** The test may also make assumptions on the data flow, particularly in the context of global variables and other shared data structures — meaning that when one asserts the contents of a variable in a different context, the assertion may no longer make sense.
- **External state:** A similar issue arises when interacting with stateful external elements (e.g., stateful communication with a third party, a database, a file, etc.): if a test checks state-dependent assertions, the runtime context might be different from the assumed state in the test environment.

The idea of translating tests into test models is not new and the challenges highlighted above have been addressed in one way or another in previous work: Arts et al. [1] attempt to generate QuickCheck automata from EUnit tests. We note that QuickCheck automata are used for testing rather than runtime checking and therefore the approach is mainly to learn patterns from existing tests and explore their generalisation to push testing further.

* Project GOMTA financed by the Malta Council for Science & Technology through the National Research & Innovation Programme 2013

Similarly — in that the translation is from testing to more generalised testing — in previous work [4], the authors show how to generate QuickCheck automata from Gherkin specifications. The main difference from the work by Arts et al. is the level of abstraction of the starting point, i.e., Gherkin specification which unlike unit tests in EUnit, are system level tests. This means that such tests make less implicit assumptions on the context in which they run. This enabled us to not only extract sequences of (non)acceptable actions as in [1], but also conditions on the system state.

Closer to the approach being proposed in the extended abstract is the work by Pace and Falzon [8] translating QuickCheck automata into runtime verification monitors. We note that this work overcomes the challenges highlighted above by starting from a testing model whose oracles are already significantly generic to handle the wide range of tests which can be generated. The downside of the approach is that like runtime verification, model-based testing is also not as widespread in industry.

With the aim of being as relevant as possible to industry, we aim to take unit tests and through machine learning techniques, infer the generic monitor from specific low-level oracles. For example, observing that in all the tests of the *deposit* method the resulting balance is always the start balance plus the deposited amount, one can generate a monitor to ensure this also holds during runtime. Inferring invariants from a number of observed system executions is not new [2, 10, 11, 13, 14]. However, this line of work focuses on generating invariants to be used during regression testing or debugging purposes and not to be deployed with the system. The consequence is that the inferred invariants might still be too narrow, catering only for runs manifesting themselves during the testing phase.

Our aim for the future is to explore ways in which invariants can be automatically filtered to avoid having false negatives at runtime. To this extent we will be exploring a number of options: focus on invariants dealing with interfacing invariants, i.e., those passed around as parameters, as these would hopefully better at revealing a protocol amongst the various methods using the interface. Another option might be to focus on tests which call more than one method in their testing sequence; again, a possible indication of a higher level of abstract (integration test rather than unit test). Another indication of abstraction level might be the privacy qualifier of variables, i.e., public as opposed to private variables, with *public* possibly indicating a higher one. We hope to share the findings of these possibilities once we carry out sufficient experimentation.

References

1. T. Arts, P. L. Seijas, and S. Thompson. Extracting quickcheck specifications from eunit test cases. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 62–71. ACM, 2011.
2. A. Babenko, L. Mariani, and F. Pastore. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 237–248, New York, NY, USA, 2009. ACM.
3. S. Colin and L. Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 525–555. Springer, 2005.
4. C. Colombo, M. Micallef, and M. Scerri. Verifying web applications: From business level specifications to automated model-based testing. In *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.*, volume 141 of *EPTCS*, pages 14–28, 2014.
5. C. Colombo, G. J. Pace, and P. Abela. Compensation-aware runtime monitoring. In *Runtime Verification - First International Conference, (RV)*, volume 6418 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2010.
6. C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149, L'Aquila, Italy, 2008.

7. C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE, 2009.
8. K. Falzon and G. J. Pace. Combining testing and runtime verification techniques. In *Model-Based Methodologies for Pervasive and Embedded Software*, volume 7706 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2013.
9. M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78:293 – 303, 2009.
10. L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126, Nov 2008.
11. L. Mariani, F. Pastore, and M. Pezze. Dynamic analysis for diagnosing integration faults. *IEEE Trans. Softw. Eng.*, 37(4):486–508, July 2011.
12. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 14:249–289, 2012.
13. F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler. Dynamic analysis of upgrades in c/c++ software. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering*, ISSRE '12, pages 91–100, Washington, DC, USA, 2012. IEEE Computer Society.
14. F. Pastore, L. Mariani, A. E. J. Hyvärinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad. Verification-aided regression testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 37–48, New York, NY, USA, 2014. ACM.