Extracting Monitors from JUnit Tests

Christian Colombo[1], Jonathan Micallef[1], and Mark Micallef[1]

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 [1, 3] which provides assurance that a system's execution flow 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 with the local industry, one of the main reasons for this 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. One potential approach to counteract this issue would be to use the information available in tests to automatically obtain monitors [2]. The plausibility of this approach is the similarity between tests and runtime verifiers: tests drive the system under test and check that the outcome is correct while runtime verifiers let the system users drive the system under observation but still has to check that the outcome is as expected.

Notwithstanding the similarities, there a significant difference between testing and runtime verification which make the adaptation of tests into monitors a challenging one: tests are typically focused on checking very specific behaviour, rendering the checks unusable in a runtime verification setting where the behaviour is user-directed rather than test-specified. Test specificity affects a number of 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 would be for monitoring purposes.

Control flow The test assertions may be specific to the control flow as specified in the test's contest with the particular ordering of the methods and the test setup immediately preceding it. The control flow is particularly problematic if the assertion is control flow sensitive (e.g., checking the sequence of method calls called).

Data flow The test may also make assumptions of 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 external stateful elements (e.g., stateful communication, 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.

In view of these potential limitations, most unit tests do not provide useful information for monitoring purposes (due to their narrow applicability). Thus, one approach in this context would be to build sifting mechanisms to select the useful tests while avoiding others which would add monitoring overheads without being applicable in general. Unfortunately, deciding which tests are general enough for monitoring use is generally a hard if not impossible task to perform automatically since the assumptions the developer made when writing the assertion are not explicit. Our solution was to explicitly ask the tester for the assumptions made. In practise this would however be impractical.

A better approach might be to help the user express the assumptions the test makes up front whilst constructing the tests. Another way of looking at it would be to create help the testers create monitors first, i.e., the checking part of the test, and then create a test for the monitor. Later, upon deployment, the driving part of the test would be discarded and only the monitoring part would be kept.

In the future we aim to explore this further on two counts:

- Exploring tests (other than unit tests) which make less implicit assumptions on the context in which they run. For example, higher level tests such as Cucumber[1] tests are typically implemented for generic input with no assumptions on the order in which the tests are to be run.
- Possibly be creating a domain specific language which facilitates the create specification of monitors and tests at the same time, allowing the user to switch between tests and monitors at any time.

References

1. S. Colin and L. Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 525–555. Springer, 2005.
2. 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.
3. M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78:293 – 303, 2009.

[1] `http://cukes.info/`