2018 IEEE International Conference on Software Testing, Verification and Validation Workshops

Test Automation—Automation of What?

Yvan Labiche
Systems and Computer Enginering, Carleton University, Ottawa, Canada
yvan.labiche@carleton.ca

***Abstract*—Taking a birds-eye view at the different activities that take place when someone engages in software testing, we discuss automation problems and some deployed solutions to the broad notion of software test automation. In doing so, we discover engineering/deployment problems as well as more fundamental scientific/research issues.**

Keywords—software testing, automation

I. Software Testing Tasks

Testing is the most common verification strategy deployed. It entails a number of tasks. Although they can be envisioned at different granularity, they at least include: **(1)** Identifying what needs to be tested, the information to use to create tests (test model) and how to use it (selection criterion); **(2)** Constructing test data by applying a criterion on the test model, along with test oracle data, that is data to identify whether the response of the piece of code under test is adequate when presented with test data; **(3)** Constructing test code that implements test cases: test data plus oracle data. This may include the construction of test drivers (or test scripts), test stubs (or mock-ups), mechanisms to collect execution information to be compared with expected ones (oracle data) in an implemented test oracle; **(4)** Executing tests (driver/stub/oracle code), which may require additional set-up and tear-down (i.e., code); **(5)** Maintaining tests to accommodate changes to the piece of code under test as well as changes to test activities (e.g., to fix a fault in a driver/stub/oracle, to trim a too large set of test cases).

Organizations try to automate these activities and academics research solutions that are automated (or automatable). This extended abstract looks at challenges and solutions for automating those tasks, specifically tasks 2 to 5. This document is not meant to be an exhaustive discussion (that would require a textbook). The intent is rather to point to some salient issues that are sometimes overlooked.

II. Test and Oracle Data Construction

The construction of test and oracle data heavily depends on the test model and associated selection criterion. We discuss a few well-known models from the automation viewpoint.

A. Plain language descriptions

When one derives tests from a plain language description of some functionality, be it a paragraph in a general requirement document, a use case description, or a function specification, one typically relies on equivalence class partitioning, boundary value analysis, or a combination of those into category partition or decision trees [1].

This work is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

These techniques are typically performed by hand, unless some tool support is provided so the user can specify equivalence classes, boundaries … and rely on combinatorics and solvers to obtain test case inputs [7]. Random selection is also easy to automate, though whether random testing is effective is still an open debate [3]. Identifying oracle data for inputs generated by these techniques is also an automation challenge.

Such solutions have also one important drawback, common to all black-box solutions: the level of abstraction of the test model hinders the construction of test scripts for those test inputs. Automating the construction of the test model (e.g., equivalence classes, boundaries) is also an interesting problem.

B. Models of State-Based Behaviour

Numerous models representing state-based behaviour have been described in the literature, along with selection criteria [1, 15, 16]. Constructing test paths, i.e. sequences of states and transitions, from these test models is relatively easy to automate because we can rely on graph algorithms.

Transforming such test paths into executable test cases is not necessarily as easy because of two main issues. First, events triggering transitions may accept input data. Automatically identifying such input values is akin to the discussion we had in the previous section. Can we identify ranges of values that are more adequate/interesting, from a testing point of view, than others? Selecting data individually for events in a test path may not work if events are related to each other in complex state-based behaviour: e.g., selecting an input value for an event in a test path limits the choice of input values in a sub-sequent event in that path. Automatically combining existing techniques with category partition may work, similarly to other models [11], when searching for interesting input values, as long as automation can rely on (automatically or manually defined) equivalence classes and boundaries (cf. previous discussion). Second, some state machines have guard conditions on transitions, introducing a so-called counter problem [5]. Constructing test paths that are executable is akin to the path-sensitization problem, which is known to be un-decidable in the general case [6], although some data flow analysis and heuristics can help solve the problem automatically [4, 12].

Regardless of the automation of (feasible) test path construction, two issues are worth discussing, though they relate to test scaffolding more so than to test case construction. The first issue is due to the level of abstraction of the test model. Events triggering transitions, along with test data, need to be transformed into calls in a test script using the API of the piece of code under test and this transformation is not

0-7695-6432-1/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSTW.2018.00037

necessarily straightforward (or automatable). Second, one has to decide what information to collect for the oracle, and at what point(s) during a test case execution to collect it. Automating these activities can be challenging (e.g., observability issues).

C. Source code

A number of automated solutions exist to derive test cases from source code: e.g., combining formal methods with testing [10], combining testing with symbolic execution [13], using meta-heuristic to find test inputs [9]. One general issue with those techniques is that they verify that the code performs as expected, as described by … the code itself. There is a risk that, should the code be faulty (which is likely), the tests pass thereby simply confirming that the code is correctly faulty. There are of course attempts to address this issue, such as presenting to the developer assertions inferred during test case executions [9], but checking that those assertions make sense cannot be automated and can be time consuming, and there is the issue of whether the test inputs are sufficiently varied to consider the inferred assertions to be accurate [14].

Instead of inferring oracle assertions from test case executions, one can rely on assertions already embedded into the code, a.k.a. contracts. Although contracts are seldom used in practice, then are extremely useful when constructing and maintaining software and tend to be reasonable alternatives to creating custom oracles [2, 8], possibly helping solve the oracle problem [17], thereby reducing test automation costs.

Code coverage tools also facilitate automation, though the more interesting coverage criteria (e.g., data flow ones) are seldom supported and it is still not clear whether those criteria are anyway reliable indicators of test suite quality.

III. Test scaffolding

A number of technologies have been created to automate the construction of the test scaffolding. They are often programming language specific. The work started with Java, due to its popularity, and then extended to other programming languages. Technologies include those for constructing test scripts (drivers) such as JUnit (junit.org) or test stubs such as JMock (jmock.org). Other technologies are programming language agnostic, such as TTCN-3 (ttcn-3.org). Other automated solutions are proprietary. These technologies are extremely useful since they provide languages to create the test scaffolding, they facilitate test cases (re) executions, they allow the evaluation of oracle assertions. They however do not help the user solve important problems: see previous discussions.

IV. Maintenance

A lot of work has been published on regression testing and made its way into automated tool support, including the possibility to fix tests (instead of simply discarding them) when the application code changes. One maintenance problem that has been overlooked, but is now stressing because of the ever growing code base of tests, is the maintenance of the test code itself. Since test code is source code, and even though authors advocate test code should be as simple as possible [6], test code of commercial software is very complex and we can expect such test code to be plagued by issues similar to application code: e.g., high complexity to the extent this affect understanding/maintenance, lack of documentation such as rationale for tests, clones.

V. Conclusion

The extent of automation available for constructing test scaffolding, specifically automation of executions, while extremely useful, sometimes leads to miss-conceptions about the level of test automation achieved in software development projects. Some practitioners are tempted to claim that they conduct automated testing because they use those technologies. I tend to oppose this claim because these technologies automate somewhat the easiest activities. Indeed, they do not help solve harder problems such as (1) what are (beyond mere code construction) start-up and tear-down activities of tests, (2) what are interesting test data, (3) what expected behaviour to check in oracle assertions. Unfortunately, as we have briefly discussed, there is seldom any general automated solution for these hard problems. To conclude, there are still many aspects of software testing that can (and should) be automated.

References

[1] Ammann P. and Offutt A. J., *Introduction to Software Testing*, Cambridge University Press, 2008.

[2] Araujo W., Briand L. C. and Labiche Y., "On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software," *IEEE TSE*, 40 (10), 2014.

[3] Arcuri A. and Briand L., "Adaptive random testing: an illusion of effectiveness?," *Proc. ISSTA*, pp. 265-275, 2011.

[4] Asoudeh N. and Labiche Y., "Multi-objective construction of an entire adequate test suite for an EFSM," *Proc. ISSRE*, pp. 288-299, 2014.

[5] Asoudeh N. and Labiche Y., "On the effect of counters in guard conditions when state-based multi-objective testing," *Proc. IEEE QSIC, Reliability and Security-Companion*, pp. 105-114, 2015.

[6] Beizer B., *Software Testing Techniques*, Van Nostrand Reinhold, 2nd Edition, 1990.

[7] Briand L. C., Labiche Y., Bawar Z. and Spido N., "Using machine learning to refine Category-Partition test specifications and test suites," *IST*, 51 (11), 2009.

[8] Briand L. C., Labiche Y. and Sun H., "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *Software - Practice and Experience*, 33 (7), 2003.

[9] Fraser G. and Arcuri A., "EvoSuite: automatic test suite generation for object-oriented software," *Proc. ACM SIGSOFT FSE*, pp. 416-419, 2011.

[10] Fraser G., Wotawa F. and Ammann P., "Testing with model checkers: a survey," *STVR*, 19 (3), 2009.

[11] Hartmann J., Vieira M., Foster H. and Ruder A., "TDE/UML: A UML-based Test Generator to Support System Testing," *Proc. Annual International Software Testing Conference in India*, 2005.

[12] Korel B., Tahat L. H. and Vaysburg B., "Model-Based Regression Test Reduction using Dependence Analysis," *Proc. IEEE ICSM*, pp. 214-223, 2002.

[13] Qu X. and Robinson B., "A Case Study of Concolic Testing Tools and Their Limitations," *Proc. IEEE ESEM*, pp. 117-126, 2011.

[14] Rahman F. and Labiche Y., "A comparative study of invariants generated by Daikon and user-defined design contracts," *Proc. IEEE QSIC*, pp. 174-183, 2014.

[15] Utting M. and Legeard B., *Practical Model-Based Testing: A Tools Approach*, Morgan-Kaufmann, 2006.

[16] Utting M., Pretschner A. and Legeard B., "A taxonomy of model-based testing approaches," *STVR*, 22, 2012.

[17] Weyuker E. J., "On Testing Non-testable Programs," *The Computer Journal*, 25 (4), 1982.