Model-based Testing in Agile Software Development

David Faragó (farago@kit.edu)
Karlsruhe Institute of Technology, Institute for Theoretical Computer Science

Abstract. With rising ubiquity of software, its quality is becoming more and more important, but harder to achieve. Model-based testing (MBT) and agile development (AD) are the two major approaches to solve this dilemma. We analyze their operational conditions and investigate how MBT can improve AD and vice versa. We conclude that strongly integrating both is the most fruitful combination. The two key requirements on MBT for AD are flexibility and rapid delivery. They can be met by underspecifying the models that MBT uses. But for current MBT techniques, underspecification has an adverse effect on efficiency, coverage and reproducibility. We believe all three aspects will be improved by a new method called lazy on-the-fly MBT, which we currently research.

1 Introduction

High quality of software is becoming more and more important, but difficult to achieve. Model-based testing (MBT) and agile development (AD) are the two main approaches to overcome these difficulties. Since both have shortcomings, we investigate whether they can benefit from one another.

In Section 2, we shortly introduce AD and describe its demands on testing. They motivate using MBT, which is introduced in Section 3. Section 4 shows that both MBT and AD can profit from one another. Since the two key requirements on MBT for AD are not yet efficiently handled by current MBT tools, Section 5 describes our currently researched method called *lazy on-the-fly MBT*, and how it can fulfill them better. The paper closes with a summary.

2 Testing in Agile Development

This section will shortly describe the main techniques of AD, with focus on the support by and demands on testing. We will derive two key requirements on MBT for AD. But before going into detail about the aspects of testing in AD, we give the big picture on how AD aims at better software development by quoting AD's *values*, stated in the *Manifesto for Agile Software Development* (cf. [14]):

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

Especially relevant for testing are: Firstly, being *flexible*, as result from value 1, 3 and particularly 4. Secondly, avoiding a big design up front (*BDUF*) by *rapidly delivering* working software, as result from value 2.

AD achieves rapid delivery by short (a few weeks) development iterations (called *sprints* in the agile method *Scrum*): In each sprint, the team implements a feature, formulated as a *user story*, which is a light-weight requirement - a few sentences in natural language. A user story is broken down into tasks, each completable within 1 or 2 person days. The team defines criteria when tasks and user stories are *done*.

To assure value 2 inspite flexibly responding to change, AD practices *continuous integration* (*CI*), i.e., controlling the quality continuously. Hence many tests should be automated for *regression testing*, which is one reason why testing plays such an important role in AD. AD focuses on unit tests and acceptance tests (cf. [9]). But since the developed software is often not sufficiently modular, integration and system tests also become necessary. When uncertainty demands that the next test steps depend on the results of the previous tests, software is also tested manually to guarantee rapid delivery of working software. AD favors *exploratory testing* for this, which is a sophisticated, methodic approach to ad hoc testing ([10]). Since manual testing is slow, it should not be used for regression tests.

So AD has strong demands on testing. I have experienced an exemplary trial of fulfilling them at WIBU-SYSTEMS AG, an SME developing DRM products. Its software department applies some agile techniques: focusing on rapid delivery, being open to changes, coordinating in daily standups, using a light-weight approach for specifications and rather concretizing them by communication with the product owner, i.e., the product manager. In this context, a BDUF for an automated, complete system test that checks all requirements was inapplicable. Many of the previous demands on testing were met by switching to semi-automated testing: Smaller, independent modules of automatic tests (scripted or in Java) were developed and then often called manually, performing semi-automated exploratory testing. This *on-the-fly* testing has proven to be very powerful and flexible: Adoption to changes in the requirements and maintenance of the small and decoupled test modules becomes easier. Hence early testing is possible. Additionally, the tester can react on-the-fly to dynamic information, e.g., suspicious signals. Analyzing the test log and locating errors is easier than in a large, fully automated test.

Since these semi-automated tests were also used for integration and system regression testing, some aspects can be improved: The effort for producing test

protocols (e.g., for ISO 9001), test coverage, time for test execution and reproducibility of errors. Combining the test modules into a fully automatic regression test is costly and destroys many of the achieved benefits from above. So when using traditional testing techniques, the concept of semi-automation is one of the most fruitful. Hence the next section will consider MBT as solution, which WIBU-SYSTEMS AG also starts using to some extent.

3 Model-based Testing

3.1 Introduction

MBT originates from black box conformance testing, but it can automate all kind of tests: unit, integration, system and acceptance tests. For unit tests, MBT's models must be sufficiently refined to give details at source code level. For acceptance testing, requirements must be integrated into the model.

MBT checks the conformance between a specification and the *system under test* (*SUT*). The specifications are mostly written in a process algebraic language which defines a model as a *Labelled Transition System* (*LTS*) of some kind ([3]). The specifications (plus some *adapters* to glue the MBT engine to the SUT) are used to derive tests via formal methods: Using algorithms from model checking, paths are traversed in the model defined by the specifications, and counterexamples to a considered requirement p (usually formulated as temporal logic formula) are returned. These yield test sequences: The inputs on the paths are used to drive the SUT, the outputs as *oracles*, i.e., to observe and evaluate whether the SUT behaves correctly. That way, MBT automatically generates superior black-box conformance tests compared to traditional, laborious testing techniques.

MBT methods can be formalized and compared using the *input output conformance theory* (*ioco*), a tool-independent foundation for conformance testing: LTSs with labels describing input, output, *quiescence* (aka *suspension*) or an internal action are used (cf. [15]). Then the *ioco* relation determines which SUTs conform to the specification (where out(x after σ) means all possible outputs of x after executing σ):

SUT s *ioco* model $m :\Leftrightarrow \forall$ suspension traces σ of m :
out(s after σ) $\subseteq$ out(m after σ).

This notion can be used in the test generation algorithm to derive a test suite from the specification to check the SUT for ioco.

3.2 Off-the-fly Model-based Testing

Off-the-fly MBT, used for instance by the tool TGV, first generates all tests using model checking and then executes them classically. The strict separation of test generation and test execution has several deficiencies: The high costs for intermediate representation, dynamic adaptations to the test generation are not possible, and non-deterministic SUTs cannot be processed effectively.

3.3 On-the-fly Model-based Testing

On-the-fly MBT, applied for instance by TorX, UPPAAL TRON and also Spec Explorer, uses the othere extreme of generating and executing tests strictly simultaneously by traversing the model and the SUT in lockstep. This eliminates all above deficiencies, but guidance and test selection is weakened. Section 5 will consider a solution.

4 Model-based Testing with Agile Development

Some previous work on MBT with AD are: [16] scarcely considers using AD to improve MBT and also MBT in AD, but suggests MBT outside the AD team, i.e., not strongly integrated. [12] aims to adapt MBT for AD and also shortly motivates using MBT within the AD team, but does not investigate in detail how to modify AD for fruitful integration, e.g., adjusting specifications and CI. It rather focuses on a case study, which empirically shows that abstraction is very important in MBT for AD. [2] gives a high-level overview on combining agile and formal methods, but only briefly considers testing. [11] uses a strict domain and a limited property language (weaker than the usual temporal logics). It uses very restricted models that are lists of exemplary paths. I do not know of any paper that differentiatedly investigates the requirements on MBT for AD, vice versa, and on both integrated strongly. Subsection 4.1 describes specifications for AD and for MBT. Subsection 4.2 investigates using AD for MBT, Subsection 4.3 MBT for AD. The section ends with a conclusion.

4.1 Specifications

This subsection looks closer at the specification types used in MBT and AD, and how they can be unified since multiple unrelated specifications are unnecessary costly, redundant (i.e., contradicting the DRY principle, [8]), and make strong integration of MBT and AD difficult. The arguments are similar to those of *agile modeling* (cf. [1]), which does not consider testing, though.

In AD, most specifications are very light-weight and mainly used to describe customer requirements. Often user stories are used (cf. Figure 1), which are too abstract to be understood on their own. They are great for communication, though, e.g. between customers and developers, which yields more detailed customer requirements. These can be put in the form of an acceptance test suite (e.g., with the framework FitNesse). This test suite is usually a list of independent, exemplary method calls and expected return values. Such an enumeration is an inefficient notation and usually leads to a bad coverage.

In MBT, specifications must be sufficiently detailed for deriving a test suite that is revealing. They are behavioral descriptions in *UML statecharts* or something

similar, e.g., Labelled Transition Systems or *Symbolic Transition Systems* (*STSs*) as depicted in Figure 1. These are very powerful, as they have precise semantics, variables, conditions in first order logic, and can describe the behavior of the SUT and requirements on several levels of abstraction (cf. Figure 1). Abstraction is achieved via underspecification by:

- allowing many non-deterministic choices for the SUT, using non-determinism in the specification or defining a real superset of outputs in a state, e.g., by defining abstracted oracles via relaxed conditions
- ignoring certain situations (e.g., hazards) by defining a real subset of inputs in a state

The level of abstraction can also be influenced by the mapping from abstract test sequences (paths of the model) to concrete test sequences (execution traces for the SUT).

Figure 1 gives simple exemplary specifications for a login web service, which is from the domain of *service-oriented architecture* (*SOA*). Such services can easily be tested via MBT and are frequently used in AD (and sometimes called *Agile Applications*), since their design concept supports AD: Services are simpler than monolithic systems and loosely coupled, assisting rapid delivery, fault tolerance and scalability.

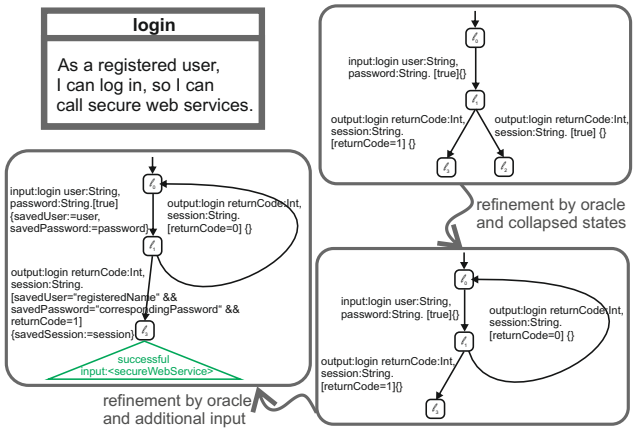

Fig. 1. Exemplary user story and STSs

The powerful specifications of MBT have the flexibility to be used for several purposes in AD (for business-facing as well as technology-facing, cf. [5]). They can particularly well replace the more precise models sometimes used in AD, e.g., UML statecharts and use cases (cf. [4]). Those are used when technical details need to be considered early; e.g., when complex business or product logic, business processes or more generally complex architectures need to be analysed, described or implemented. So prefering such a powerful specification over those used in AD leads to a unification with the following benefits:

- cost and redundancy are reduced
- AD can be applied to the refinement process of the models, i.e., when defining more detailed models by reducing the level of abstraction
- strong integration of MBT and AD is enabled, which is considered in the following sections.

4.2 AD for MBT

AD is so successful because, amongst others, it fixes time and cost, but can handle the scope and changing requirements flexibly. This is also important for MBT to avoid rigidness and a BDUF. In detail, many agile methods can be applied fruitfully to the processes and artifacts (models and adapters) used in MBT:

- pair programming, e.g., to increase the design quality of the models,
- starting with very abstract models, to support flexibility and efficient communication,
- iteratively refining aspects of the model within sprints, for rapid delivery.

Therefore, MBT profits from AD, but underspecification is necessary (especially in the first iterations). If we do not modify AD itself, though, i.e., do not integrate MBT and AD strongly with one another, we have some discrepancies, e.g., the specification languages and the definition of *done* does not match and CI (against a reference implementation, if need be) is not effective. Furthermore, the benefits described in the next section do not take effect.

4.3 MBT for AD

MBT is the solution to many problems of testing. Some also occur in agile development: deceptive and insufficient coverage, low flexibility and high maintenance. Additionally, AD requires rapid delivery and continous integration with regression tests. Hence MBT can profitably be applied to AD: Efficient tests can be generated and executed automatically with an appropriate coverage. The test suite can be changed flexibly by modifying the conscise models. This is especially important for *acceptance test driven development* (*ATDD*, see [7]), where automated acceptance tests are part of the definition of *done*.

Furthermore, advanced coverage criteria not only produce better tests, but also better measurements for quality management, e.g. for ISO 9001. For instance, andrena object's agile quality management ISIS (see [13]) is state of the art and considers many relevant aspects: Test coverage is only one of 10 metrics and measured using EclEmma. But that only allows limited coverage criteria, namely basic blocks, lines, bytecode instructions, methods and types (cf. [6]). Since automated tests are a central quality criterion, especially in AD, and since [17] shows that more sophisticated coverage criteria (e.g. MC/DC) are more meaningful, MBT in AD can also improve agile quality management.

But if MBT itself is not modified, we get again the problem of discrepancy (cf. previous subsection), and possibly a BDUF or a model that is too rigid for AD.

4.4 Conclusion

The last subsections have shown that MBT can profit from AD and vice versa, but without mutual adaptions, i.e., strong integration, both MBT and AD restrict each other. For the integration, abstract models are necessary, though, which current MBT tools cannot efficiently handle. Hence we will consider a new method in the next section.

5 Lazy On-the-fly Model-based Testing

5.1 Introduction

Using on-the-fly MBT solves several deficiencies (cf. Subsection 3.3), but backtracking in the model can no longer be used since the SUT usually cannot undo already executed test steps. Hence guidance and therefore test selection is weakened.

Our project MOCHA[1] aims at tackling this problem by designing a new method, *lazy on-the-fly MBT*, that fulfills ioco and can harness the advantages of both extremes, on-the-fly and off-the-fly MBT. It executes subpaths of the model lazily, i.e., only when there is a reason to, e.g., when a test goal, a certain depth, an inquiry to the tester, or some non-deterministic choice of the SUT is reached (a so-called *inducing states*). Hence the method can backtrack within the model's subgraphs bound by inducing states. While backtracking, the method can harness dynamic information from already executed tests, e.g., non-deterministic coverage criteria. As result, we expect strong guidance to reduce the state space and to produce fewer and more revealing tests with higher coverage. These main advantages over existing technology can also help improve reproducibility and efficiency of testing, especially for non-deterministic systems and abstract specifications.

5.2 Lazy On-the-fly MBT with AD

Underspecification supports AD, as it empowers flexibility and fast modelling for rapid delivery. As described in the previous subsection, one main advantage of lazy on-the-fly MBT is its guidance on subpath scale which uses dynamic information. It efficiently finds short and revealing tests with better coverage criteria and reproducibility - also for underspecified systems. This is particularly important for regression testing in CI. These coverage criteria can also improve measurements for quality management. Furthermore, the method can react to uncertainty, i.e., information not available prior to test execution (e.g., non-determinism) by adaptively taking into account dynamic results. Hence lazy on-the-fly MBT can be considered automated exploratory testing. The experience made at WIBU-SYSTEMS AG shows how important and powerful dynamic information is (cf. semi-automated exploratory testing in Section 2).

[1] funded by Deutsche Forschungsgemeinschaft (DFG)

Future work within MOCHA include modifying ioco for finer-grained refinements and identifying efficient heuristics for subpath selection as well as coverage criteria that incorporate dynamic information, e.g., non-deterministic choices.

6 Summary

We investigated how MBT and AD can be combined: MBT for AD improves flexibility, maintenance and coverage. AD for MBT avoids rigid models and a BDUF. By unifying the specifications and strongly integrating MBT and AD, we get the highest profits:

- reduced cost and redundancy
- effective CI and sensible definition of *done* that checks if the model is conform to the code
- both MBT and AD can unfold their full potential.

For this to work, we need abstract models. These can be processed more effectively with our new lazy on-the-fly MBT.

7 Acknowledgements

I would like to thank Peter H. Schmitt, chair of our Logic and Formal Methods Group, Alexander Schmitt, head of software development at WIBU-SYSTEMS AG, and Leif Frenzel from andrena objects AG.

References

1. Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
2. Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal versus agile: Survival of the fittest. *Computer*, 42:37–45, 2009.
3. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer Verlag, 2005.
4. Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
5. Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009.
6. EclEmma: Using the Coverage View. http://www.eclemma.org/userdoc/coverageview.html/. (April 2010).
7. Elisabeth Hendrickson. Driving development with tests: ATDD and TDD. *STARWest 2008*, 2008.
8. Andrew Hunt, David Thomas, and Ward Cunningham. *The Pragmatic Programmer. From Journeyman to Master*. Addison-Wesley Longman, Amsterdam, 1999.
9. J.B.Rainsberger. Integration tests are a scam. *Agile2009*, 2009.
10. Cem Kaner, Hung Q. Nguyen, and Jack L. Falk. *Testing Computer Software*. John Wiley & Sons, Inc., New York, NY, USA, 1993.
11. Mika Katara and Antti Kervinen. Making model-based testing more agile: A use case driven approach. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*. Springer, 2006.
12. Olli-Pekka Puolitaival. Adapting model-based testing to agile context. *ESPOO2008*, 2008.
13. Nicole Rauch, Eberhard Kuhn, and Holger Friedrich. Index-based process and software quality control in agile development projects. *CompArch2008*, 2008.
14. James Shore and Shane Warden. *The art of agile development*. O'Reilly, 2007.
15. Jan Tretmans. Model based testing with labelled transition systems. *Formal Methods and Testing*, pages 1–38, 2008.
16. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.
17. Yuen-Tak Yu and Man Fai Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software*, 79(5):577–590, 2006.