Improved Underspecification for Model-based Testing in Agile Development

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

Abstract: Since model-based testing (MBT) and agile development are two major approaches to increase the quality of software, this paper considers their combination. After motivating that strongly integrating both is the most fruitful, the demands on MBT for this integration are investigated: The model must be underspecifyable and iteratively refineable and test generation must efficiently handle this. The theoretical basis and an example for such models is given. Thereafter, a new method for MBT is introduced, which can handle this more efficiently, i.e., can better cope with nondeterminism and also has better guidance in the model traversal. Hence it can be used in agile development, select more revealing tests and achieve higher coverage and reproducibility.

key words: model-based testing; agile development; iterative refinement; nondeterminism; ioco; on-the-fly; off-the-fly;

1 Introduction

Model-based testing (MBT) and agile development (AD) are two main approaches to overcome the difficulties of reaching high quality in complex, ubiquitous software. As AD's strength is validation and MBT's strength is verification, we investigate how they can benefit from one another.

Section 2 shortly introduce AD, describe its deficits and its implications on verification. Section 3 introduces MBT with an underlying conformance testing theory as suitable formal method for this verification. Thereafter, the state of the art of MBT tools is described. Section 4 introduces related work and then shows the benefits of strongly integrating MBT and AD, as well as its requirements. In the last subsection, the requirements on the specification are solved by introducing Symbolic Transition Systems (STSs) and their underspecification and refinement possibilities. The end of the subsection concludes how MBT and AD are technically combined. Section 5 describes improved techniques for MBT in this combination with AD. This is our currently researched method, which efficiently processes the underspecified models and has further advantages. The conclusion gives a summary and future work.

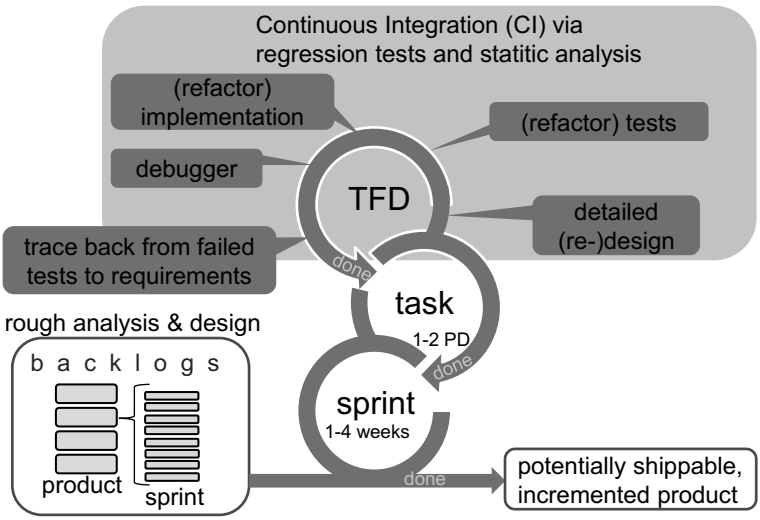


Figure 1: Exemplary agile method: XP and Scrum

2 Agile Development

2.1 Introduction

This section will briefly describe the main techniques of AD (cf. [SW07]), Subsection 2.2 AD's deficits and 2.3 the implications on verification.

In short, AD is iterative software development, such that requirements and solutions can evolve. This is supported by a set of engineering best practices, such that high-quality software increments can be delivered rapidly.

The big picture on how AD aims at better software development is given by AD's *values*, stated in the *Manifesto for Agile Software Development* (see [FH01]):

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

Figure 1 sketches an example of how software development achieves these values. Two of the most prominent agile methods are used, which can be combined easily: Extreme Programming (XP) [Tea98, JAHL00] and Scrum [TN86]. Other agile methods (e.g., the Agile Unified Process [Amb02] or Feature Driven Development [PF02]) lead to the same implications on verification.

AD achieves rapid delivery (see value 2) by short (a few weeks) development iterations (often called *sprints*): In each sprint, the team implements one ore more features from

the product backlog. These are often formulated with *user stories*, which are light-weight requirements - a few sentences in natural language. The user stories are broken down into tasks, which are put into the sprint backlog. By iterating over all tasks, the sprint is implemented. The result is a potentially shippable, incremented product.

In more detail, each task should be completable within 1 or 2 person days. Within a task, the developer practices even shorter iterations using *test-first development (TFD)*: after refining (or refactoring) the design, according test cases are specified (or refactored). Only thereafter the feature is implemented, using the IDE's semi-automatic features, such as method creation and refactoring. For more complex test case failures, the developer can use debugging and trace back from the test case to the according feature.

To assure value 2 in spite of flexibly being able to respond to change, AD practices *continuous integration* (*CI*), i.e., controlling the quality continuously. This is performed automatically using a CI framework in the background, e.g., *Hudson* or *CruiseControl*, which can use automated *regression tests*. AD's main focus on unit tests and acceptance tests (cf. [Rai09]). If less modular software is developed, more tests have to be performed by integration instead of unit tests. Unfortunately, this is often the case in practice. The CI framework can also use static analysis, e.g., to detect code smells and too high cyclomatic complexity. The team defines exit criteria, e.g., when tasks and sprints are done. These *definitions of done* are then given to the CI framework to be checked automatically.

Using agile processes, rapid delivery of high quality software increments can be achieved. These can be shown to the customer and her feedback can be flexibly incorporated in the following iterations. Hence AD is strong on validation. Therefore, a large fraction of organizations have adopted AD: two out of three according to the study [Amb08] together with Dr. Dobb's magazine, one out of three according to [For09]. Furthermore, the *Agile Conference* had a growth of 40% in 2009.

2.2 Deficits of Agile Development

The main deficit of AD is that too little specification and documentation is delivered. Specifications are becoming more and more important, e.g., for certificates and in todays component-oriented software development, since components need to be specified to reuse and distribute them. But more precise documentation is also needed by developers so they have direction and knowledge of the purpose while navigating through code, e.g., in pair programming.

AD also has some difficulties in testing, which is an integral part of AD (cf. the previous subsection): The used test coverage is often insufficient, e.g., 60%, and deceptive, e.g., statement coverage (cf. [LCBR05]). . Directly written test cases are less flexible and require more maintenance than the specifications for MBT. For instance, if exception handling is refined (as with the concise changes from Figure 2c to Figure 2d), a lot of test cases might have to be modified to incorporate this. Finally, tracing back from failed test cases to high level requirements is often difficult in AD.

Before the next section will show how MBT can help to overcome these deficits in AD,

the following subsection will describe in general the implications of AD on verification.

2.3 Verification

All agile processes require the following, which is relevant for verification: 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.

So specification should not obstruct AD's frequent iterations and support flexibility. To be able to integrate the verification process into the CI, it should be automatic and fast, such that developers get quick feedback. Optimally, a *10-minute build* (including CI) should be reached.

Formal methods (FM) can be used for the verification in AD to increase the degree of automation (e.g., using static analysis or model-based testing), the quality of the software, as well as the confidence in the quality. These aspects are especially important in safety-critical domains.The combination of AD and safety-critical software is being investigated in the project *AGILE* (see [Ope09]). It started in 2009 and firstly considers DO178B certifications for confidence in the quality, but other certifications will follow.

Investigating the combination of FM and AD has started some years ago, [BBB+09] gives a good overview.

J.B. Rainsberger stated on the *Agile Conference 2009* (see [Rai09]) that contract-based testing should replace integration tests in agile development. Several languages and tools exist for this, for instance JMLUnit [ZN10] and the MBT tool Spec Explorer [VCG+08].

Because of the above requirements on verification, light-weight formal methods are suited best. An example is the tool *FindBugs*, which is a static analysis tool on bytecode-level that detects certain bug patterns [APM+07]. It can also be plugged into Hudson. Unfortunately, it produces many false negatives.

MBT is a light-weight formal method that generates tests from specifications. Hence the whole system is still checked, and the formal method can easily be integrated into the development process - technically as well as psychologically, since agile teams are accustomed to testing. Hence this paper focuses on MBT.

3 Model-based Testing

3.1 Introduction

MBT can be used for conformance testing, i.e., to automatically check the conformance between a specification and the *system under test* (*SUT*), which is formally described in Subsection 3.2. To avoid error-prone redundancy and additional work, this paper considers MBT that automatically generates tests from the product's specifications, i.e., no

additional, explicit test specifications are necessary. MBT is mainly a black box technique and can automate all kind of tests: unit, integration, system and acceptance tests. For unit tests, MBT's models must be sufficiently refined to give low-level details. For acceptance testing, requirements must be integrated into the model.

The specifications are mostly written in a process algebraic language which defines a model as a *Labelled Transition System* (*LTS*) of some kind ([BJK+05]). The MBT considered in this paper uses *model checking* algorithms to derive test cases from the specifications: Paths are traversed in the model defined by the specifications, and witnesses to a considered requirement p, usually formulated as temporal logic formula, are returned. These witnesses (counterexamples for $\neg p$) yield test sequences: The inputs on the paths are used to drive the SUT, the outputs to observe and evaluate whether the SUT behaves correctly, i.e., as *oracles*. That way, MBT automatically generates superior black-box conformance tests compared to traditional, laborious testing techniques.

3.2 Ioco

MBT methods can be formalized and compared using the *input output conformance (ioco) theory*, a tool-independent foundation for conformance testing. The *ioco* relation determines which SUTs conform to the specification. The specification is given as LTS L describing the input and output (*i/o*). More precisely, $L = \langle Q, L_I, L_U, T, q_0 \rangle \in LTS(L_I, L_U)$, i.e., labels describe inputs L_I, outputs L_U, *quiescence* δ (aka *suspension*) or the internal action τ, cf. [Tre08]. L_I, L_U and the states Q are non-empty countable sets, $q_0 \in Q$ is the initial state, and $T \subseteq Q \times (L_I \cup L_U \cup \{\delta, \tau\}) \times Q$ the transition relation. The *suspension traces* of L, $Straces(L)$, is the set of all paths of L, the labels of T^* (T's reflexive, transitive closure), but all τ removed. The SUT is considered as *input output transition system* $\in IOTS(L_I, L_U)$, i.e., an input-enabled LTS $\in LTS(L_I, L_U)$, meaning that all inputs are enabled in all reachable states. The relation $ioco \subseteq IOTS(L_I, L_U) \times LTS(L_I, L_U)$ is defined as follows: $i \; ioco \; s :\Leftrightarrow \forall \sigma \in Straces(s) : \;\; \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$, where $\text{out}(x \text{ after } \sigma)$ means all possible outputs (or δ) of x after executing 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: By traversing s and nondeterministically choosing amongst all controllable choices, but keeping the branchings that are uncontrollable choices the SUT can take nondeterministically, the ioco algorithm generates test cases which are themselve LTSs $\in IOTS(L_U, L_I)$: they are output-enabled, cycle-free, deterministic, free of τ, finite, singular input- or δ-enabled (i.e., all states have exactly one input or δ enabled), and the leafs are exactly the verdicts. These test cases are *sound*, i.e., they do not report false negatives with respect to the ioco relation. The test suite containing all possible test cases is *exhaustive*, i.e., for each SUT that is not ioco-correct it contains a failing test case. A test case is run by executing it synchronously in parallel with the SUT.

3.3 State of the Art

The techniques that existing MBT approaches implement can be divided into two types:

- *Off-the-fly MBT* (also called *offline MBT*): First generate the complete test suite using model checking and then test the SUT classically by executing the generated test suite.
- *On-the-fly MBT* (also called *online MBT*): Generate and execute tests strictly simultaneously by traversing the model and the SUT in lockstep.

The MBT community has realized the need to also consider nondeterministic SUTs. This is particularly important for complex, e.g. distributed, systems, where the tester does not have full control (for instance because of race conditions and exceptions). Specifications that also offer nondeterminism can cover this situation and also enable abstractions helpful for AD (cf. Section 4.3). How effective nondeterministic specifications can be processed depends on the technique employed.

3.3.1 Off-the-fly Model-based Testing

Off-the-fly MBT (also called offline) used for instance by the tool TGV (cf. [BJK+05, JJ05]), 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 nondeterministic SUTs cannot be processed effectively.

3.3.2 On-the-fly Model-based Testing

On-the-fly MBT (also called online) is being applied, for instance, by TorX [TBR03], UPPAAL TRON [LMN04] and also Spec Explorer [VCG+08, BJK+05, UL07]. It uses the other extreme of generating and executing tests strictly simultaneously by traversing the model and the SUT in lockstep. This eliminates all above deficiencies:

- Overhead in time and memory can be reduced since we no longer require an intermediate representation, such as the costly representation of the specification in the test suite generated by the ioco algorithm.
- Dynamic information from test execution can be incorporated in other heuristics, especially guidance, so that the MBT process can adapt to the behavior of the SUT at runtime and therefore generate more revealing tests. For instance:
  - Investigating the part of the state space around faults already found is a promising strategy because faults tend to occur in cliques. For instance, the coverage criteria can be strengthened for components in which many faults were found.
  - In contrast, if we can identify the causality between faults, we can avoid pursuing consequential faults.
  - When generating a test sequence, we can try to discharge as many test goals as possible in each state. This can be too liberal a strategy if dynamic adaptations are not available. If they are, we can subsequently generate more tests to isolate

the function deviating from its requirement.

- On-the-fly MBT can also process nondeterministic SUTs by instrumenting them to track the actual choices they made at runtime.

The directly executed tests can also be recorded, yielding a classical test suite that can later be re-executed by common testing tools without formal methods. Being able to *reproduce* a test execution helps to check whether a fault has been fixed or to provide repeatable evidence of correctness. If the SUT is nondeterministic, recorded test suites no longer guarantee reproducibility. The main drawback of on-the-fly MBT, however, is its weak guidance used for test selection. A solution is considered in Section 5. It also enables stronger utilization of coverage metrics as guidance while traversing the model graph, so that newly generated tests really raise the coverage criterion. As formal specifications contain control flow, data and conditions, the same coverage metrics can be applied as on source code level, for instance statement, transition or MC/DC coverage. Which coverage criterion is best on the specification level varies.

4 Model-based Testing and Agile Development

4.1 Related Work

Some previous work on MBT with AD are: [UL07] scarcely considers using AD to improve MBT and also MBT within AD. It suggests MBT outside the AD team, i.e., not strongly integrated. [Puo08] 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. [KK06] 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.

[Rum06] gives a good overview of MBT when evolution (as in AD) is involved. It uses the same modelling language for the production system and the tests, but not the same models. As mentioned in Subsection 3.1, our kind of MBT generates tests from the product's specifications to reduce work and redundancy.

No other author (I know of) differentiatedly investigates the requirements on our kind of MBT for AD, vice versa, or on both integrated strongly (which therefore covers both verification and validation). Such investigations have been made in [Far10] and are described in the following subsection.

4.2 Strongly Integrated

4.2.1 MBT for AD

The general approach of integrating MBT and AD is changing the TFD cycle in Figure 1 (cf. Figure 3): CI then uses MBT for regression testing and metrics such as various specification and code coverages. Therefore, specifications instead of tests are written (or refactored) - so this is rather a *specification-first development (SFD)* cycle. These are read by the MBT tool. Developers can now trace back from failed test cases to the corresponding specification, which is easier than the original tracing back to the corresponding requirement. The specifications are additional deliverables of the development process. Using MBT also counters 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 concise models. This is especially important for configuration management and *acceptance test driven development* (*ATDD*, see [Hen08]), 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 [RKF08]) 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. [ecl]). Since automated tests are a central quality criterion, especially in AD, and since [YL06] shows that more sophisticated coverage criteria (such as MC/DC) are more meaningful, MBT in AD can also improve agile quality management.

If we do not modify MBT 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* do not match, so CI is not very effective. Furthermore, the benefits described in the next section do not take effect. Additional specifications for MBT are required, and must be flexibly changable. Finally, MBT must work fast: during specification to avoid a BDUF and during verification for a quick CI.

4.2.2 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 (cf. previous Subsection). For this, the iterative and incremental processes in AD can be applied on the specification-level:

- starting with very abstract models, to support flexibility,
- iteratively refining aspects of the model within sprints, for rapid delivery,
- using the refined specifications that are sufficiently detailed for MBT.

Furthermore, agile methods as pair programming, reviews, but also CI help find defects in the specifications early. Because of AD's strong validation, differences between the specification and the customers expectations are also fixed. Finally, clearer and more modular source code that is packaged in increments can improve MBT's efficiency.

Therefore, MBT profits from AD, but powerful underspecification is necessary (especially in the first iterations).

4.2.3 Conclusion

The last subsections have shown that AD can profit from MBT and vice versa, but without mutual adaptions, i.e., strong integration, both MBT and AD restrict each other. For the integration, a unified specification that offers flexible underspecification is necessary, which will be presented in the following subsection. Since current MBT tools cannot efficiently handle these (cf. Section 3), the next section will introduce a new MBT method for this.

4.3 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, [HTC99]), and make strong integration of MBT and AD difficult. The arguments are similar to those used for *agile modeling* (cf. [Amb02]), which does not focus on testing, though.

In AD, most specifications are very light-weight and mainly used to describe customer requirements. Often user stories are used, 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 are usually put in the form of an acceptance test suite with a framework as FitNesse. Such test suites are often more data-oriented than flow-based, and the maintenance of test scenarios is difficult, as is achieving high 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 2. These are very powerful, as they have precise semantics, i/o and location (i.e. model-) variables, and conditions in first order logic. The labels of the ioco theory (cf. Subsection 3.2) are lifted and now have the form: `{input,output,internal,` δ`}:name.[guard]{update}`, with `name` being a method name, `guard` a formula in first order logic and `update` a term over all location variables and the i/o variables of `name`. The precise semantics are given in [FTW06]. STSs can describe the behavior of the SUT and requirements on several levels of abstraction. The most abstract level (cf. Figure 2a) can be used for communication and to give an overview. Although an abstract STS is less intuitive than a user story, the higher precision is more important, especially in safety-critical domains. Abstraction is achieved via underspecification by:

- allowing many nondeterministic choices for the SUT, using nondeterminism 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 could also be influenced by the mapping from abstract test sequences (paths of the model) to concrete test sequences (execution traces for the SUT), but to be able to formalize underspecification and refinement, we use the underspecification techniques within the specifications. Therefore, we have for specifications $s_1, s_2 \in LTS(L_I, L_U)$: s_1 *refines* $s_2 \ :\Leftrightarrow \forall \sigma \in Straces(s_1) \cap Straces(s_2) : \ \text{in}(s_1 \text{ after } \sigma) \supseteq \text{in}(s_2 \text{ after } \sigma) \ \wedge \ \text{out}(s_1 \text{ after } \sigma) \subseteq \text{out}(s_2 \text{ after } \sigma)$.
Extending the ioco relation to $LTS(L_I, L_U)^2$, this can be written as s_1 accepts more inputs than s_2 and $s_1 \ ioco_{Straces(s_1) \cap Straces(s_2)} \ s_2$. With this, the following implication in relation to the SUT i holds: $i \ ioco \ s_1 \implies i \ ioco \ s_2$. So using the most refined specification for MBT within CI also guarantees the correctness up to the most abstract specification (that replaced user stories).

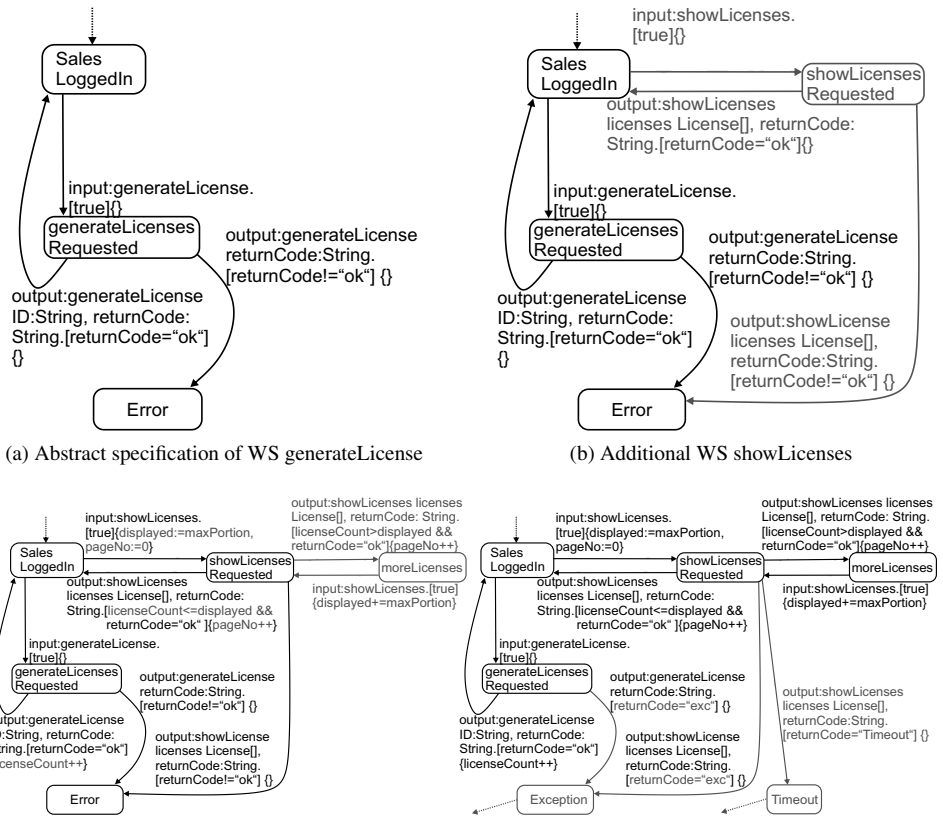
Figure 2 gives simplified exemplary specifications for web services generating licenses from WIBU SYSTEM AG's *License Central*, which is from the domain of *service-oriented architecture* (*SOA*). Such services can easily be tested via MBT and are frequently used in AD (and sometimes called *Agile Applications*), since their design concept supports AD: Services are simpler than monolithic systems and loosely coupled, assisting rapid delivery, fault tolerance and scalability.

Figure 2b refines Figure 2a by specifying more inputs. Figure 2c describes a different functionality than Figure 2b and therefore is not a refinement (state *moreLicenses* sometimes replaces state *loggedIn* but offers less inputs). Figure 2d refines Figure 2c by specifying less outputs.

All in all, 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. [CG09]). They can particularly well replace the more precise models sometimes used in AD, e.g., UML statecharts and use cases (cf. [Coc00]). 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. The resulting workflow as extension to Figure 1 is depicted in Figure 3.

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.



(a) Abstract specification of WS generateLicense

(b) Additional WS showLicenses

(c) More functionality for showLicenses

(d) Refined exception handling

Figure 2: Exemplary STS specifications for web services generating licenses

5 Lazy On-the-fly Model-based Testing

5.1 Introduction

Using on-the-fly instead of off-the-fly MBT solves several deficiencies (cf. Subsection 3.3.2).

The main drawback of on-the-fly MBT, however, is its inability to use the backtracking capabilities of the model checking algorithms: Each step of the model traversal is executed strictly simultaneously in the SUT, which cannot undo these steps. The lack of backtracking complicates test selection since different subpaths (longer than one transition) cannot be chosen amongst. Instead, test selection has to be performed at the early stage of deciding which transition to traverse. All mentioned on-the-fly tools have the major disadvantage of weak guidance used for test selection.

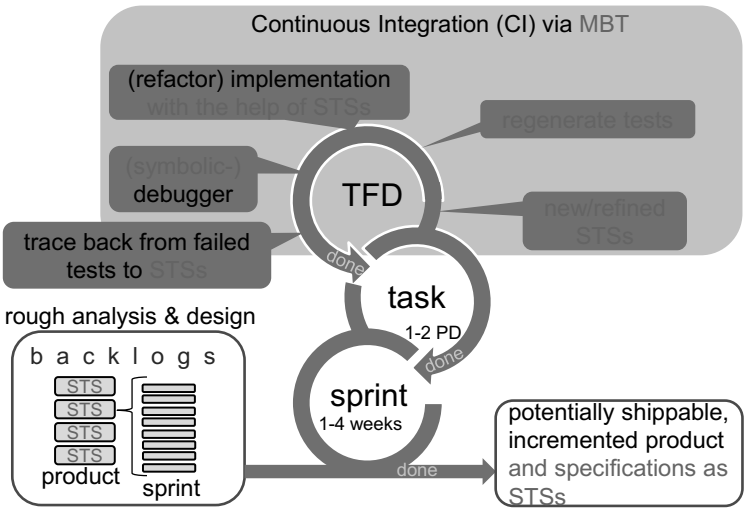


Figure 3: MBT with STSs in our exemplary agile method

To preserve the advantages of the two approaches and avoid their disadvantages, our project MOCHA[1] offers a novel method that synthesizes these contrary approaches: It 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 as well as off-the-fly MBT. It executes subpaths of the model lazily on the SUT, 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 nondeterministic choice of the SUT is reached, a so-called *inducing state*. Therefore, we do interleave model traversal and execution of the SUT, but not strictly after each test step, only loosely after several steps.

So the top-level algorithm for lazy on-the-fly MBT repeatedly compiles and executes subpaths in the following way:

1. $currentState :=$ initial state;
2. Traverse the subgraph containing all subpaths from $currentState$ to the next inducing states;
3. Select one subtree π;
4. Execute π in the SUT (as determined by the ioco theory) and log the results;
5. If the termination criteria are not yet met:
   if π ends with a terminal: GOTO 1;
   else $currentState :=$ last state of π; GOTO 2;

Lazy on-the-fly MBT can exploit the advantages of both extremes to a large extent:

- As in off-the-fly MBT, backtracking is again possible, now within the model's subgraphs that are bounded by the states where test execution is induced (*inducing*

[1] funded by Deutsche Forschungsgemeinschaft (DFG)

states). So from within a subgraph, we do not look backward or forward across inducing states, but we can search the complete subgraph for charged test goals, to choose the most promising subtree, e.g., the one passing the most charged test goals with the fewest possible steps. Hence lazy on-the-fly MBT is strongly guided by prioritizing subpaths or subtrees (*guidance on subpath scale*). This powerful guidance can easily incorporate other heuristics, enables new heuristics, harness dynamic information from already executed tests, e.g., nondeterministic coverage criteria, and therefore help generate and execute tests faster and more flexibly.

- The cost for local intermediate representations can strongly be bounded since only the current subtree is passed to the test driver controlling the SUT. If the SUT is nondeterministic and paths are pruned when nondeterministic choices are made, costly bifurcation is avoided.
- Nondeterministic SUTs can be processed easier, e.g., by inducing test execution at their nondeterministic choice points and using the execution result dynamically.

As result, we expect strong guidance to reduce the state space and to produce fewer and shorter tests with higher coverage, to reveal more relevant faults, for instance those which only occur after long sequences of events.

An example is testing `showLicenses` over multiple pages (cf. Figure 2d), which corresponds to the temporal formula $(\diamond pageNo > 1)$. This requires a test sequence that calls `generateLicense` more than `maxPortion` times and the `showLicenses` two times. Using off-the-fly MBT for this test does not work, since the variables in the STSs cause a potentially infinite state space and the nondeterministic choices that must be passed on the test sequence cause a huge test tree, e.g., for exception handling. Using on-the-fly MBT works better, but since it has weak guidance and the test sequence is so long, it is very inefficient: many `showLicense` calls (or other WS calls) will likely occur until `generateLicense` is called more than `maxPortion` times. With lazy on-the-fly MBT, we can choose error states and the test goal as inducing states and therefore get via better guidance a test tree that is very similar to the test sequence $\texttt{generateLicense}^{\texttt{maxPortion}+1}\texttt{showLicenses}^{2}$. It only has bifurcations of length one because after the alternative outputs for exceptions or timeouts, these paths are immediately pruned.

5.2 Within AD

The main advantages described in the previous subsection can improve the efficiency of testing, especially for abstract specifications and nondeterministic systems. This is particularly useful in AD, since underspecification supports AD, as it empowers flexibility and fast modelling for rapid delivery. Another advantage of lazy on-the-fly MBT is its strong guidance, which uses dynamic information. It efficiently finds short and revealing tests with better coverage criteria and helps reproducibility - also for nondeterministic systems. This is particularly important for automated regression testing in CI, which therefore performs continuous ioco checks. Finally, these coverage criteria can also improve measurements for quality management.

6 Conclusion

6.1 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 with reduced cost and redundancy and with effective CI and sensible definition of *done* that continuously checks if the model is conform to the code. For this to work, we need abstract models. STSs on several abstraction levels are suitable. These can be processed more effectively with our new lazy on-the-fly MBT: It chooses partial execution at the most sensible point in time and can therefore cope with nondeterminism and leverage backtracking and dynamic information. This leads to better testing and reproducibility in AD.

6.2 Future Work

The main future work is implementing lazy on-the-fly MBT and conduction case studies.

Theoretical future work within MOCHA include modifying ioco for finer-grained refinements that are compatible with robustness tests, and identifying efficient heuristics that incorporate dynamic information, e.g., nondeterministic choices.

7 Acknowledgements

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

References

[Amb02] Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Amb08] Scott Ambler. Has agile peaked? *Dr. Dobb's*, 2008.

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using FindBugs on production software. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 805–806. ACM, 2007.

[BBB+09] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *Computer*, 42(9):37–45, 2009.

[BJK+05] 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.

[CG09] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[ecl] EclEmma: Using the Coverage View. http://www.eclemma.org /userdoc/coverageview.html/. (April 2010).

[Far10] David Faragó. Model-based Testing in Agile Software Development. In *30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Testing meets Agility*, to appear in Softwaretechnik-Trends, 2010.

[FH01] M. Fowler and J. Highsmith. The Agile Manifesto. In Software Development, Issue on Agile Methodologies, `http://www.sdmagazine.com`, last accessed on March 8th, 2006, August 2001.

[For09] Agile Development Method Growing in Popularity. *Forrester*, 2009.

[FTW06] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A Symbolic Framework for Model-Based Testing. In Klaus Havelund, Manuel Nez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006.

[Hen08] Elisabeth Hendrickson. Driving Development with Tests: ATDD and TDD. *STARWest 2008*, 2008.

[HTC99] Andrew Hunt, David Thomas, and Ward Cunningham. *The Pragmatic Programmer. From Journeyman to Master*. Addison-Wesley Longman, Amsterdam, 1999.

[JAHL00] Ronald E. Jeffries, Ann Anderson, Chet Hendrickson, and Chapter Circle Of Life. Extreme Programming Installed, 2000.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[KK06] 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.

[LCBR05] Joseph Lawrance, Steven Clarke, Margaret Burnett, and Gregg Rothermel. How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 53–60, Washington, DC, USA, 2005. IEEE Computer Society.

[LMN04] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-time Systems Using Uppaal. In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2004.

[Ope09] Project AGILE, 2009.

[PF02] Stephen R. Palmer and John M. Felsing. *A Practical Guide to Feature-Driven Development (The Coad Series)*. Prentice Hall PTR, February 2002.

[Puo08] Olli-Pekka Puolitaival. Adapting model-based testing to agile context. *ESPOO2008*, 2008.

[Rai09] Joseph B. Rainsberger. Integration Tests Are a Scam. *Agile2009*, 2009.

[RKF08] Nicole Rauch, Eberhard Kuhn, and Holger Friedrich. Index-based Process and Software Quality Control in Agile Development Projects. *CompArch2008*, 2008.

[Rum06] Bernhard Rumpe. Agile Test-based Modeling. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 10–15. CSREA Press, 2006.

[SW07] James Shore and Shane Warden. *The art of agile development*. O'Reilly, 2007.

[TBR03] Jan Tretmans, Ed Brinksma, and Cote De Resyste. TorX: Automated Model Based Testing - Cte de Resyste, 2003.

[Tea98] The "C3 Team". Chrysler Goes to "Extremes". *Distributed Computing*, pages 24–26, 1998.

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, 1986.

[Tre08] Jan Tretmans. Model Based Testing with Labelled Transition Systems. *Formal Methods and Testing*, pages 1–38, 2008.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

[YL06] 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.

[ZN10] Daniel M. Zimmerman and Rinkesh Nagmoti. JMLUnit: The Next Generation. In *FoVeOOS 2010*, pages 20 – 29, 2010.