Effective Model-Based Testing

Theo C. Ruys(✉) and Machiel van der Bijl

Axini B.V., Amsterdam, Netherlands
{theo.ruys,machiel.van.der.bijl}@axini.com
https://axini.com

Abstract. Model-Based Testing (MBT) has grown from testing infamous coffee machines to complex industrial size systems with many interfaces and parallel message exchanges.

Over the last decade Axini has developed a state-of-the-art platform, which can be used to model, analyze and test (very) large systems. We have successfully applied our platform to several medium to large applications in the finance and high-tech industry.

While modeling and testing such systems, we stumbled upon quality, management and engineering challenges to cope with the complexity of the whole MBT trajectory. We have developed several engineering guidelines to control both our modeling and testing trajectories.

Moreover, over the last few years, we have witnessed that model based working has gotten more acceptance within the industry. For MBT we see a shift of focus from testing to modeling. Originally, MBT would be applied when the development of the System Under Test was completely finished. Currently, we start the modeling during the analysis and design phase of the system. We have adapted our engineering methodology and guidelines accordingly.

We share our lessons learned and provide valuable insights for modeling and test engineers involved in model based testing and/or development.

Keywords: software testing · system testing · model-based testing · model-based development · test methodology

1 Introduction

Model-Based Testing (MBT) [21] is an established black-box testing approach for automatic generation of test cases [9]. MBT has grown from testing infamous coffee machines to complex industrial size systems with many interfaces and parallel message exchanges.

This paper is concerned with what could be called the methodology of MBT. Over the last decade, Axini has developed a state-of-the-art platform, which can be used to model, analyze and test (very) large systems. We have successfully applied our platform to several medium to large industrial applications. Recently, we have witnessed that model based working has gotten more acceptance within the industry under several different names (model based software engineering,

N. Jansen et al. (Eds.): Principles of Verification: Cycling the Probabilistic Landscape,
LNCS 15262, pp. 234–251, 2025.
https://doi.org/10.1007/978-3-031-75778-5_11

model driven design, model based testing, etc.). For MBT we see a *shift left* from testing to modeling.

Motivation. This work on a methodology for MBT has been motivated by the increasing complexity of the complete MBT trajectory and the increasing size of our models. With respect to the modeling and testing phase, we stumbled upon quality, management and engineering challenges to cope with the complexity of the whole MBT trajectory. Furthermore, over the years, our toolset has become more powerful which enabled us to include more behavior and data of the system in our models: our models have grown to over 10k lines of model. There was a strong need to standardize our way of working across our projects such that both our modelers and our clients could easily navigate through these large models.

Related Work. MBT has been around for half a century [7]. There have been numerous papers and surveys on 'MBT in Practice', which report on the experiences with applying some sort of MBT, e.g., [9,11,13]. Not much is published on how to manage and organize the MBT trajectory for large transition-based models, though.

Already in 1999, [6] reports on MBT to generate test cases to cover a broad coverage of the input domain. The *data model* is used to generate test cases to cover the input domain. In this paper we are concerned with a *behavioral model*: the model specifies the behavior of the SUT. We are also concerned, of course, to cover as much of the domain of the input data, but this is taken care of by the solver and strategies of our MBT tool. [14] reviews the research area of MBT and introduces several papers from ICTSS 2010. The discussed research is about new theoretical approaches and MBT tools from academy. This paper is concerned with the practical application of a mature MBT tool for industry-size systems and a methodology to effectively construct a behavioral model.

In [16] we reported on management problems regarding the verification trajectory in the realm of model checking. For MBT, the testing tools have matured and the software configuration management problems are solved: all testing artifacts are reproducible. In this paper we will show – similar to verification – that also in MBT different kinds of models have to be used for different purposes.

Contribution. We have developed several engineering guidelines to control both our modeling and testing trajectories. These guidelines help us to manage our MBT projects; not only for the first MBT test but also for any retest that may follow. We will explain the main ingredients of our MBT engineering methodology. Along the way we will report on our experiences and share several lessons learned. We also report on our experiences in the recent *shift left* of our MBT projects. The modeling is now already started during the analysis and design phase of the system. Our engineering methodology and guidelines have been adapted accordingly.

Overview. Section 2 discusses our definition of MBT. In Sect. 3 we briefly introduce our platform that we use in our MBT projects. Section 4 discusses some of

our early experiences with MBT and the lessons learned. Section 5 presents our methodology for our MBT projects and Sect. 6 takes a closer look at the most important phase of this methodology. Section 7 reports on the recent *shift left* from 'MBT afterwards' to 'MBT upfront'. Section 8 summarizes the paper and touches upon future work.

Lessons learned

The paper contains several *lessons learned* sections with a grey background that start with a light bulb.

2 Model-Based Testing

One could argue that all testing activities should be regarded as 'model-based': to test software, one has to make a model of the software, and hence all software testing is model-based [23]. In this section we briefly explain what we mean by 'model-based testing'. For our definition of MBT, we follow the tradition of *conformance testing* [4,19], where the SUT has to conform to a formal model.

For our notion of MBT, the SUT is treated as a black-box without knowledge about its internal structure. The only way a tester can control and observe an implementation is via its *interfaces*. The aim of testing is to check the correctness of the behavior of the SUT on its interfaces [20]. These interfaces are not limited to GUIs, but can be any communication interface, e.g., I/O interfaces (standard input/output, file systems, middleware, etc.), API calls, etc.

MBT is an automated testing technique where a program is used to *generate* the test cases. The behavior of the SUT is represented by a formal *model*, represented in a formal notation with a formal semantics, e.g., labeled transition systems or some other form of state-transition system. The approach uses a correctness notion – for example the **ioco** implementation relation [20] – to express conformance between implementation and the model. The model is used to automatically and systematically generate scenarios (test cases) to test the correctness of the SUT. Coverage criteria on the model steer the generation of these test scenarios.

MBT assumes that the implementation of the SUT is based on a *specification*, which explicitly describes what the SUT should do. A specification is typically a collection of documents which express in detail how the SUT should behave; such a specification is not only the blueprint for the development and implementation of the SUT but also the source of information for the formal model of the system: the model is thus an abstract description of the SUT. Testing then amounts to checking whether the behavior of the SUT 'conforms' to the behavior as specified by the abstract model.

Figure 1 gives an overview of an MBT-approach. Given a model of the SUT, the test generator generates a test suite of test cases. The test executor then attempts to execute each generated test case on the SUT. If a label in the test case is a *stimulus* (input), the adapter transforms the stimulus to a physical

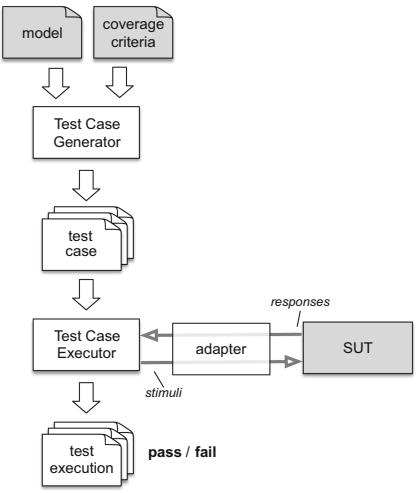

Fig. 1. Offline Model-Based Testing.

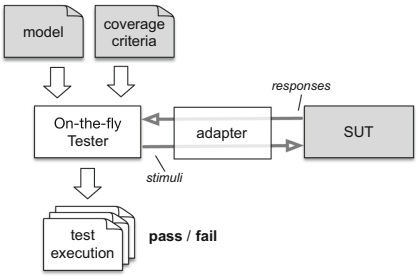

Fig. 2. Online Model-Based Testing.

label and offers the physical label to the SUT. If a label in the test case is a *response* (output), the test executor waits until it receives the logical response from the adapter. When the SUT returns a response which is not expected in the test case, it concludes that the test has *failed*. If the test executor can execute a complete trace of the test case against the SUT, it concludes that the test has *passed*. Figure 1 is regarded as an *offline* method: the set of test cases is generated *before* the actual testing takes place at the SUT.

Figure 2 shows an alternative MBT approach: *online* testing. Test cases are generated *on-the-fly* and partly, while testing the SUT. The on-the-fly Tester walks over the (internal representation) of the model offering stimuli to the adapter and observing the responses of the SUT. The Tester decides upon the next test-step after a stimulus or response (as opposed to generating all test-steps before testing). Typically, state and transition information is recorded during a test execution such that – e.g., for a next test execution – the Tester can decide what stimulus to choose to increase either the state, transition or path coverage of the model, or even a combination of these.

3 Axini Modeling Platform

This section introduces the Axini Modeling Platform (AMP), the platform developed by Axini [24]. AMP is an online MBT tool. The main contribution of this paper is the presentation of the methodology that we developed to systematically model and test technical systems in a structured way. It should be applicable to most MBT tools, especially online testing tools. Still, it might be interesting

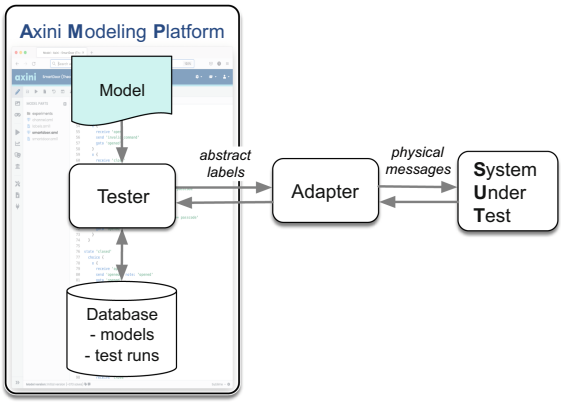

Fig. 3. Testing with the Axini Modeling Platform.

to be aware of specific characteristics of the tool that we are using for MBT. A more thorough introduction to AMP can be found in [15,23].

AMP is in active development for more than a decade and has been successfully applied to several dozens of projects. These projects can be divided into two areas: (i) MBT of financial applications and (ii) MBT of technical systems. This paper is mainly concerned with the latter: the MBT of complex reactive systems. These industrial size systems often have many interfaces over which – independently and concurrently – messages are being exchanged. The messages themselves are large, typically XML, or some binary format.

Figure 3 shows how AMP is used for modeling and testing. A model of the SUT is developed in the IDE of AMP. For the purpose of testing, the model is translated to an intermediate representation which is used by the Tester to generate test cases. An adapter is used to convert the physical labels of the SUT (e.g., XML messages) to abstract labels of the model, and vice-versa. All models and testing artifacts are saved in a database; all executed tests are fully reproducible and easily accessible. AMP is a web-application and can be accessed using a web browser. Additionally, AMP offers powerful exploration and visualization functions to analyze the models.

Figure 4 shows a screenshot of AMP when in modeling mode. AMP has a familiar IDE-like interface. On the left, the main functions of AMP are offered, e.g., visualizer, explorer, starting a test run, showing the history of previous test runs. The large window on the right is the modeling editor. In the top row, various model related functions are available, e.g., saving the files, starting a test run, undoing the last change(s), showing differences with previous versions, etc.

For each SUT one or more adapters need to be developed to connect AMP to the SUT. Although AMP eases the development of such adapters, the development of an adapter can still be laborious and time-consuming. This, however, applies to all forms of automated testing. We will not discuss the development of

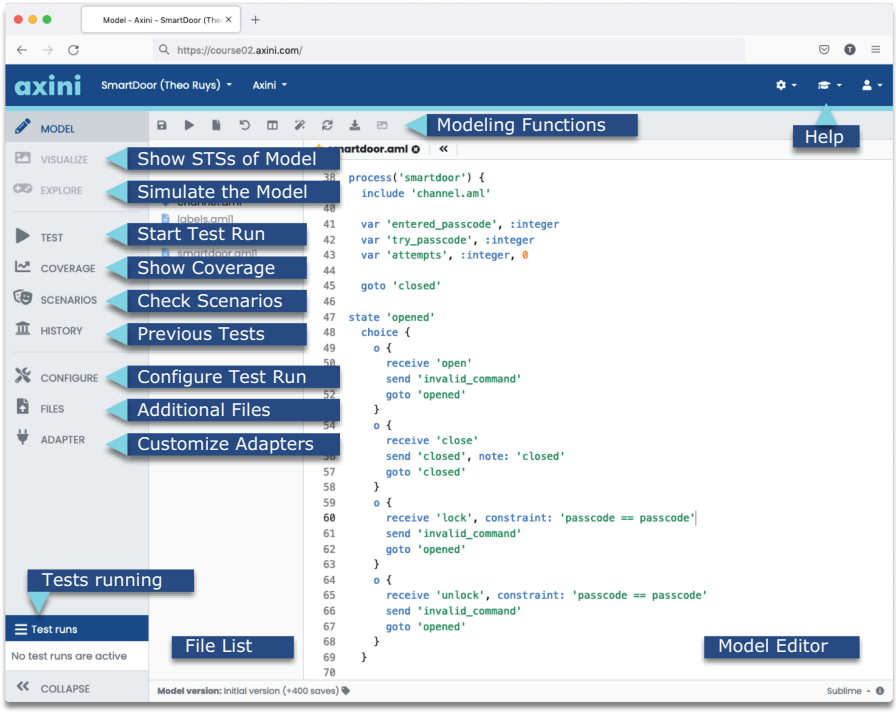

Fig. 4. Screenshot of Axini's AMP – Modeling View.

adapters in this paper, apart from the following lesson that we learned the hard way.

> *Keep adapters as thin as possible.*
>
> It is tempting to move behavior that is hard, boring or irrelevant for the testing from the model to the adapter, e.g., by introducing abstract labels which perform special actions within the SUT. This should be avoided: an adapter should not keep any state of the SUT. When models deviate too much from their specifications they become hard to understand and test-cases become hard to interpret, because of the hidden functionality within the adapter.

We have matured our modeling language and platform to the point that our adapters are a *thin layer* that only connect to the SUT and convert labels to SUT messages and vice versa. Almost all of it is standardized and can be generated.

The modeling language for AMP is the Axini Modeling Language (AML). An AML model consists of parallel processes, which together describe the behavior of the SUT. The processes can communicate with the environment and can exchange messages with each other. For the *behavioral* part, AML borrows from

PROMELA [10] and LOTOS [3]. For the *data* part, AML borrows from the Ruby programming language [12]: AML does not only support the usual simple types like boolean, integer, floats, and strings, but it also supports structured types like arrays, structs and hashes. Moreover, AML offers model structuring features like behaviors, macros and run-time functions.

Each process in an AML model is translated to a *symbolic transition system* (STS) [8], an extension of a labeled transition system (LTS). The semantics of an AML model is the cartesian product of the STSs of the processes. AMP's Tester is an online MBT tester (see Fig. 2). The implementation of this Tester is based on the **ioco** theory of Tretmans [20,22]. To cope with industrial size systems, AMP has been extended with several features, such as partial-order reduction, support for timed testing (i.e., expedited stimuli and urgent transitions) [17] and powerful coverage-improving strategies to steer the on-the-fly testing.

The size and complexity of the systems that we validate carry over to our models: our models are often larger than 10k lines of AML model. The duration of our MBT projects range from a few weeks to many months, to continuous MBT during the lifecycle of a system. AMP is used at several companies such as Dutch rail infrastructure manager ProRail, Thermo Fisher Scientific, a global supplier of analytical instruments, and the leading Dutch insurance company Achmea. AMP is also used at several Dutch universities in courses on software testing.

4 Lessons Learned

This section discusses some experiences during our first MBT projects with AMP, where we use a pragmatic and often improvised approach. We came to realize that we had to structure our modeling and testing trajectory differently.

Originally, most MBT projects that we were involved in were 'MBT afterwards' projects: the development of the application was already finished when we started the MBT of the application. We would often pair in a team of two persons: one person took care of the development of the adapters, the other one was responsible for the models.

Modeling is hard.

The *theory* of MBT seems simple: (i) construct the model, (ii) develop adapters to connect the MBT tool to the SUT, (iii) press the 'Test' button, and (iv) analyze the results afterwards. However, the *practice* of MBT has taught us that modeling is hard:

- What is the right level of abstraction for the model?
- How to structure the model into different processes?
- How to check that the model is correct?
- How to check that the model is complete?
- How to deal with errors in the SUT?

The modeler needs help when constructing models.

In the first years, we did not have any procedures, conventions or rules to standardize the modeling and testing trajectories. Most of our MBT engineers had a programming background. These engineers simply tried to project their programming and engineering expertise onto the modeling. When the MBT projects became more complex, this ad-hoc approach did not scale up. Furthermore, each engineer developed their own modeling style, which led to models which had less in common and were hard to understand by others.

Use model conventions.

Like computer programs, models should be understandable by other MBT engineers. Modelers should use the same procedures, modeling style and conventions to build their models. Preferably, the modeling language and the test tool should help the user.

One aspect of MBT was clear from the beginning: every test run should be reproducible. Our own MBT tool (AMP), for example, uses `git` and an SQL-database to store all models and all details of the test runs.

Pass = Pass, Fail = Fail.

A model should precisely describe the behavior of the SUT.

Sometimes, though, for complex systems, it is easier to construct a model which allows slightly more behavior of the SUT than the specification prescribes. This means, however, that we have to analyze each *passed* verdict afterwards, to check that the SUT indeed behaves as it should. Similarly, it is sometimes easier to allow less behavior in the model. In this case each *fail* of the test tool has to be analyzed.

We learned the hard way that additional analysis of test cases afterwards is tedious, error prone and hard to maintain. If the SUT passes against the model, it should always imply that no bugs have been found. And if the SUT fails on a model, a real defect has been identified.

The last lesson learned is especially important for a retest of an application, where any additional analysis of test cases should be avoided.

There will always be a retest.

We noticed that after a successful MBT test of an application, a client would later often request for a retest of the application. Sometimes this retest would be several years later. Clients expect that a retest is easy (and cheap): after all, the model of the previous test *only* has to be updated to the new version of the SUT, and one can press the 'Test' button again.

In practice, however, a retest would not be straightforward at all. There a several reasons for that:

– The starting point for the retest are the artifacts from the last MBT test: a test report, the model(s) used, the adapters, and a database of test results.

How to test which bugs have been fixed? How to test the new behavior of the application?
- The MBT team responsible for the re-test would often be different from the original team. Without modeling conventions, the new MBT team would have a hard time understanding the model and analyzing the test results.

We concluded that the testing trajectory should be organized with a retest in mind: the trajectory should be standardized in such a way that it would ease any future retest.

5 Model and Test Methodology

From Sect. 4 we learned that without a clear methodology, the modeling and testing trajectory is rather ad-hoc. The methodology presented here aims at a more controlled testing trajectory. The methodology has three main objectives:

- Provide a structured way of modeling and testing.
- Use the SUT as soon as possible.
- Ease the inevitable retest.

Especially the first objective is important: modelers should use the same concepts and terminology to enhance the interoperability between modelers.

The central philosophy of the methodology is borrowed from extreme programming [1]: *test early, test often*: from the beginning of the trajectory, one should validate the models against the SUT. The obvious and most important reason for this is that errors in the SUT can only be found at the SUT. But there is another pragmatic reason: experience has taught us that developing a model can be hard and non-trivial; it is convenient to use the test tool in combination with the SUT as a debugger for the model. Alternatively, we could have used a model-checker to automatically test properties of our models (without the SUT). Unfortunately, our models are too complex for current state-of-the-art model-checkers.

The methodology is geared towards online testing, where test results are immediately available after a failing test case. This approach fits nicely with incremental development approaches such as AGILE, SAFE and SCRUM.

The MBT methodology consists of the following phases.

1. *Study specifications.* Usually, several detailed specifications define the behavior of the SUT. These specifications not only form the basis for the implementation of the system but also for the models. When studying the documentation, ambiguities and under-specifications are often found.
2. *Construct a tracer bullet.* A tracer bullet[1] is a minimal model which touches all interfaces of the SUT (but with limited checks). A test run with the tracer bullet is typically fast; the tracer bullet can be seen as the smoke test for

[1] The term 'tracer bullet' is a familiar term in software engineering [18] and is borrowed from flammable ammunition that gunners use to plot the trajectory of their shots.

the model, adapters and the SUT. We will refer to the tracer bullet model as M_{tracer} or M_T.
Although limited in functionality, M_T is rather important: during the project it is used as a connection test for the SUT. Before each testing session, M_T is used to check whether the configuration and status of the SUT are still correct. If M_T leads to a fail, there is no point for further testing, as most test cases will probably fail.

3. *Incremental modeling and testing.* This is the most important phase of the process. We start with an abstract, minimal model of the system (usually just M_T) and gradually add more behavior and data to the model. We continuously use the MBT tool to check that the SUT still conforms to the extended model. Test runs are random and not deep. Errors found in the SUT are collected in a specific model. Section 6 discusses this phase in greater detail.
4. *Long and deep test runs.* When all behavior has been added to the model and all (random and shallow) test runs pass, it is time for the final phase. The MBT tool is configured to execute several long and deep test runs. We will switch from random testing to test strategies which focus on increasing the coverage (state, transition and/or sub-path) of the model.

Random testing is hard to beat.

It is common knowledge from any model-based way of working that the formalization of requirements already reveals bugs and inconsistencies in the specifications. Running the first random tests with M_T usually reveals several additional misinterpretations of the specifications as well.

Within our MBT projects, the majority of the bugs are found during step 3, where we use a *random* testing strategy. When there is an error, it is found fast and the error trace is not deep: although there may be only a single bug, many paths through the model lead to it.

When the model is complete and no more errors are being found in step 3, we will run long and deep test runs in step 4 with advanced coverage enhancing strategies. Typically, in this finalizing test phase, only a few new errors are found, e.g., errors which are time related or more intricate errors.

This seems to be consistent with the findings of Boehme and Paul [5] that it is hard to beat random testing.

6 Incremental Modeling and Testing

The central idea of this phase is to incrementally build the model of the SUT. Whenever we have added new, but complete functionality to the model, we use the SUT to test this new functionality. If we find a new error, we know that it is caused by the newly added functionality.

Figure 5 shows the cyclical nature of this test phase. Apart from M_T, the stable tracer bullet, we keep track of two models:

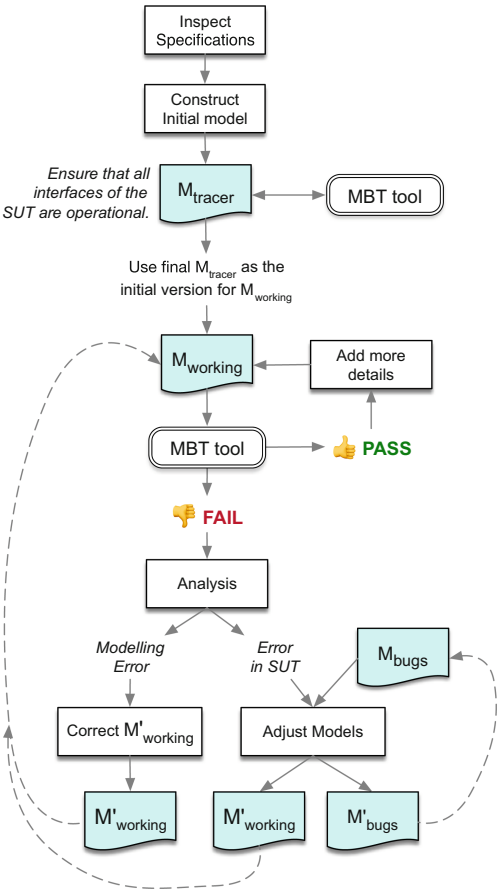

Fig. 5. Incremental modeling and testing: debugging at the SUT.

- $M_{working}$ (M_W), the working model of the SUT on the basis of the formal specification of the SUT. When errors are found, M_W is adjusted to the observed behavior of the SUT, though.
- M_{bugs} (M_B), which contains the scenarios to replay the errors found.

The *objective* of the cyclic 'debugging at the SUT' phase is to find as many errors as possible, as soon as possible, and include them in M_B. We ensure that M_W reflects the specification as closely as possible, except where bugs in the SUT or the specification require us to deviate. After this phase, M_W should always *pass*: M_W has become a validated model of the SUT. Similarly, all scenarios in M_B should *fail*: they constitute the errors found in the SUT. In this way we incrementally build a powerful regression test-set.

Figure 5 works as follows. We use M_T as our starting point: the initial version of M_W. We gradually add more details to M_W and use our MBT tool to validate

the SUT against $\mathsf{M_W}$. As long as the test cases *pass*, we keep adding more details to $\mathsf{M_W}$.

If one of the test cases *fails*, we need to analyze the source of the failure. We have to determine whether the fail is a modeling error, or a bug in the SUT or specification; this can be determined by analyzing the failing test case within the MBT tool. If the fail is a modeling error, we simply correct $\mathsf{M_W}$ and test the corrected version $\mathsf{M'_W}$ against the SUT. If the fail corresponds to an error in the specification or SUT, we have to update both $\mathsf{M_W}$ and $\mathsf{M_B}$ according to this new error. We need to construct a scenario that demonstrates the bug found. This scenario is subsequently included in $\mathsf{M'_B}$. $\mathsf{M_W}$ has to be updated to $\mathsf{M'_W}$ to behave exactly like the observed SUT; otherwise we cannot continue testing. Naturally, we mark the deviation of the specification in $\mathsf{M'_W}$.

The analysis of a *fail* can sometimes be time-consuming. The benefit of this approach, however, is that failing traces are all collected in $\mathsf{M_B}$, and $\mathsf{M_W}$ keeps being a correct model of the SUT.

Please note that – due to the non-deterministic nature of the SUT – $\mathsf{M_B}$ is not simply a collection of linear scenarios to replay a failing test case. Theses scenarios typically contain non-deterministic behaviors as well.

Retest. With the three resulting models, a retest of the SUT now becomes easy and straightforward:

1. Use $\mathsf{M_T}$ to test the connection with the new SUT.
2. Use $\mathsf{M_B}$ to see which bugs have been fixed.
3. Disable the fixed bugs in $\mathsf{M_W}$ and use $\mathsf{M_W}$ to check that the new system behaves correctly.
4. If new functionality has been added to the SUT, use the 'debugging at the SUT' approach to incrementally add the functionality to $\mathsf{M_W}$ and test it against the SUT.

Weather Models. Apart from $\mathsf{M_T}$, $\mathsf{M_W}$ and $\mathsf{M_B}$, we usually need additional models to model *unintended* behavior. We distinguish so-called good, bad and ugly weather behavior:

Good weather: *intended* behavior of the system. This consists of the intended interactions that are described by the specifications using valid data. Good weather is also known as *happy flow*. This is typically $\mathsf{M_W}$.

Bad weather: *unintended* behavior of the system. Behavior that has been specified, but should not be displayed by well behaved clients of the SUT under normal operating circumstances. Bad weather often ends up in $\mathsf{M_W}$ as well.

Ugly weather: *unspecified* behavior of the system. Modeled with the deliberate purpose of testing the robustness of the SUT. Examples are sending invalid, corrupt or very large messages. Usually, this kind of behavior needs support from the adapter. Moreover, we often use a special model to test this.

Use different models for different purposes.

Engineers new to MBT often have to get used to the concept of the *model* of MBT. Engineers with a *programming* background are used to a single program; they expect that there is also a single model, which contains all intended and unintended behavior. Engineers with a *testing* background are used to a collection of test cases; they expect that there is a model for each test purpose.

As we have seen, the answer lies in the middle. $\mathsf{M_W}$ is always a description of the behavior of the SUT. The MBT tool can be trusted to generate the relevant test cases. We maintain $\mathsf{M_T}$ and $\mathsf{M_B}$ to control the testing trajectory. And there may be additional weather models. Moreover, we often use dedicated models which zoom into certain aspects of the SUT while abstracting from others. Finally, we regularly use models which define several scenarios to observe special, predefined behavior of the SUT.

7 Model-Based System Engineering

In the AGILE/LEAN community there is a concept called *shift-left*, see for example SAFE [26]. This concept refers to shifting testing left in the development cycle, from late in the cycle to as early as possible.

One can characterize our search in the effective application of MBT as a shift-left as well. What started as a system/acceptance-test activity at the end of the development cycle is now a *Model-Based* activity that starts during requirements capturing and design, at the beginning of the development cycle.

Figure 6 shows the stages where we currently apply our *Model-Based* way of working. We have given different names to the activity based on where it is applied in the development phase. When shifting left, the *model* becomes the central ingredient of the activities; the testing keeps being very important, though. We distinguish the following *Model-Based* activities:

- MBT: the activity of modeling and testing at the end of the development phase,
- MBD (Model-Based-Development): the activity of modeling and testing during the development activities of the system where we focus on design, and
- MBSE (Model-Based System Engineering): the activity of modeling and testing from the start of the development cycle; in this case the focus is even more on the design of the system and there is more opportunity to prevent errors from entering the design.

7.1 MBT After Development

When we started with Axini a decade ago, the focus was on testing: the T of MBT. We focussed on creating a platform to automatically test systems as

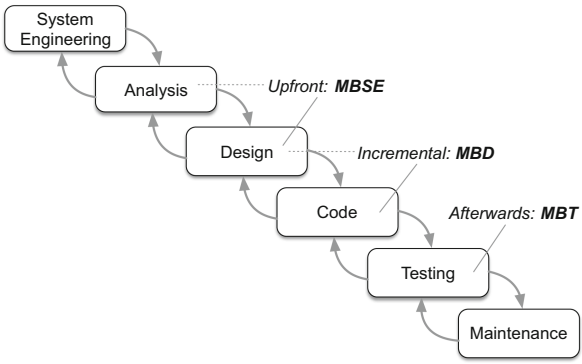

Fig. 6. Application of MBSE, MBD and MBT within the system development cycle.

thoroughly as possible. When needed, we added features to both the language and the toolset, necessary to model and test complex systems.

Our clients regarded MBT as a special kind of testing needed on special occasions (e.g., due to extensive change to the system) and/or for special systems (e.g., safety critical systems). We were asked to assess the quality of such systems as a second opinion. The MBT test was often done as an additional acceptance test, after delivery of the system by the development team.

It is clear that MBT will include the behavior of the complete system. Hence, MBT will overlap with some of the tests of the development team. If not coordinated, there will be duplication of work.

MBT *afterwards is an insurance premium.*

When MBT finds serious bugs, our clients are glad that they invested into MBT: it prevented destructive problems on production.

When the bugs found by MBT are less severe (e.g., corner cases, minor deviations from the specifications, deviations from the XSD), our clients are glad that the system is working correctly. After development, though, they are reluctant to fix these minor bugs. These bugs will often be put on the backlog for a next version of the system.

We observed that the insurance of MBT afterwards also often triggers the development team to be more wary about their test suite: it is not appreciated if MBT would expose some embarrassing bugs.

Aside from the insurance premium, there are systems where MBT is the only way to reliably test these systems, i.e., complex, multi-threaded systems where communication is concurrently over many different interfaces. To our knowledge there are no other test methods that can thoroughly test such reactive systems. If such a system has bugs, MBT is the only automated test approach to find them.

We set out to enhance our platform and MBT methodology such that (i) MBT could replace parts of the traditional testing, (ii) we could find errors earlier on in the project and (iii) that MBT was not only applicable for reactive systems, but for all systems.

7.2 MBD During Development

Applying MBD during the development cycle has a better business case. The investment stays roughly the same – i.e., the construction of the model – while the return on investment is higher. As described in Sect. 5 and 6 we have developed an MBD approach that fits the agile/incremental software development. MBD was still seen as an independent activity to ensure higher quality, but we were more integrated in the development team. In general, we would be one sprint behind the developers and errors that were found with MBT were picked up quickly and fixed in the same sprint.

MBD was a significant improvement. Most errors were fixed before the delivery date. We found that the testing power of MBD is far better than the traditional script-based testing of our clients; we found bugs that the developer tests did not find. And importantly, there were virtually no problems with the code that was tested with MBD, in acceptance or production.

We observed that the MBT team can help designers and developers because they have the same knowledge of the system. The models helped in discussions during sprint meetings. For example, in one project the development team accepted the solution from the MBT model and implemented it for the system when they were unable to get their own solution to work.

We observed that MBT took over some parts of the testing, which was normally carried out by the development or traditional test team. MBT during development also opened the door to continuously use MBT during the life cycle of the system. It was natural to preserve and continue to use the MBT models and test-ware.

Applying MBD during development made it clear that we could improve even further, especially in helping designers with the system design. While modeling, we found problems with the design, but often the design was already finished; sometimes already implemented in other systems.

7.3 MBSE at the Start of Development

As our toolset matured, and clients saw the benefits of using MBD, we noticed that the designers also benefited from modeling activities during the design phase. Improvements on the visualization and simulation made it possible to discuss and simulate design decisions and to simulate what-if scenarios. As a result more errors were detected and prevented early on.

From a Lean perspective this scenario makes the most sense: *find errors as early* on in the development cycle: first(er)-time-right. This will reduce the amount of rework later on, hence reducing wait time and speeding up the rate

of development. We also see this in the business cases that clients created: our approach speeds up development by 30–50% and it saves a lot in the testing hours. Just as an example, a bigger project was capable of saving at least 5000 testing hours [25].

Start modeling early in the development process.

The principal part of the investment in *Model-Based* working is modeling. The investment stays the same whether one does it early or late in the development process. Given the advantages that come with modeling it is better to do it as early in the process as possible.

Early modeling reveals errors as early in the process as possible. As we know from Lean software development and Boehms law [2] it is cheaper to fix errors early in the process.

Immediate feedback on conformance of the implementation with respect to its specification by thoroughly testing speeds up development tremendously. On top of that modelers can help out designers and developers.

Finally, continuous MBT will prevent most errors from entering the integration, acceptance and production phases.

8 Conclusion

In this paper we have reported on our experiences applying MBT to industrial size systems. We have presented several lessons learned.

The paper has two important contributions. We have presented a methodology to standardize and structure the modeling and testing trajectory. This methodology leads to more uniform models and is crucial when a system has to be retested. Furthermore we have reported on the *shift left* from 'MBT afterwards' to 'MBSE upfront'. This has several implications for the MBT trajectory and important advantages for our clients.

We are still extending and enhancing AMP in several ways. Current developments range from additions to the modeling language, improved multi-user support, to support CI/CD integration in existing development pipelines of our clients. Given the *shift left* in our MBSE projects we are also improving the means of analyzing our models, for example by model checking.

Acknowledgments. AMP is already in development for more than a decade. Many current and former employees of Axini have worked on our toolset, and made it possible to apply MBT to industrial size systems. The internal discussions on structuring and standardizing the MBT trajectory were the basis for this paper.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
2. Boehm, B.W.: Software engineering. IEEE Trans. Comput. **25**(12), 1226–1241 (1976)
3. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Comput. Netw. **14**, 25–59 (1987)
4. Brinksma, E., Alderden, R., Langerak, R., van de Lagemaat, J., Tretmans, J.: A formal approach to conformance testing. In: de Meer, J., Mackert, L., Effelsberg, W. (eds.) Proceedings of the IFIP TC 6 Second International Workshop on Protocol Test Systems, Berlin, FRG, 3–6 October 1989, North-Holland. Also: Memorandum INF-89-45, University of Twente, The Netherlands, pp. 349–363 (1990)
5. Böhme, M., Paul, S.: A probabilistic analysis of the efficiency of automated software testing. IEEE Trans. Software Eng. **42**(4), 345–360 (2016)
6. Dalal, S.R., et al.: Model-based testing in practice. In: Boehm, B.W. Garlan, D., Kramer, J. (eds.) Proceedings of the 1999 International Conference on Software Engineering, ICSE 99, Los Angeles, CA, USA, 16–22 May 1999, pp. 285–294. ACM (1999)
7. Elmendorf, W.R.: Automated design of program test libraries. Technical report TR 00.2089, IBM (1970)
8. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31848-4_1
9. Garousi, V., Keles, A.B., Balaman, Y., Güler, Z.Ö., Arcuri, A.: Model-based testing in practice: an experience report from the web applications domain. J. Syst. Softw. **180**, 111032 (2021)
10. Holzmann, G.J.: The Spin Model Checker - Primer and Reference Manual. Addison-Wesley, Boston, USA (2003)
11. Li, W., Le Gall, F., Spaseski, N.: A survey on model-based testing tools for test case generation. In: Itsykson, V., Scedrov, A., Zakharov, V. (eds.) TMPA 2017. CCIS, vol. 779, pp. 77–89. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-71734-0_7
12. Matsumoto, Y.: Ruby in a Nutshell. O'Reilly (2001)
13. Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech 2007), pp. 31–36 (2007)
14. Petrenko, A., da Silva Simão, A., Maldonado, J.C.: Model-based testing of software and systems: recent advances and challenges. Int. J. Softw. Tools Technol. Transf. **14**(4), 383–386 (2012)
15. Ruys, T.C.: Introduction to Model Based Testing and the Axini Modeling Language. Axini B.V., October 2021
16. Ruys, T.C., Brinksma, E.: Managing the Verification Trajectory. Int. J. Softw. Tools Technol. Transf. (STTT) **4**(2), 246–259 (2003)
17. Schmaltz, J., Tretmans, J.: On conformance testing for timed systems. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 250–264. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85778-5_18

18. Thomas, D., Hunt, A.: The Pragmatic Programmer (20th Anniversary Edition). Addison-Wesley (2020)
19. Tretmans, J.: A formal approach to conformance testing. In: Rafiq, O. (ed.) Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems, Pau, France, 28–30 September 1993, vol. C-19. IFIP Transactions, pp. 257–276. North-Holland (1993)
20. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_1
21. Utting, M., Legeaard, B.: Practical Model-Based Testing: a Tools approach. Elsevier (2010)
22. von Styp, S., Bohnenkamp, H., Schmaltz, J.: A conformance testing relation for symbolic timed automata. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 243–255. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_19
23. Vos, T.E., van Vugt-Hage, N.: Software Testing - Workbook for O/U course 'Software Testen', cursuscode IB3202. Open University, Heerlen, The Netherlands (2019)
24. Axini. https://www.axini.com
25. Bits&Chips. Model-based testing in safety-critical Scaled Agile. https://bits-chips.nl/artikel/model-based-testing-in-safety-critical-scaled-agile
26. SAFe. https://www.scaledagileframework.com