Model-Based Testing in the Field: Lessons Learned

Wolfgang Grieskamp
Microsoft Research
wrwg@microsoft.com

Abstract: For half a decade model-based testing has been applied at Microsoft in the internal development process. Though a success story compared to other formal quality assurance approaches like verification, a break-through of the technology on a broader scale is not in sight. What are the obstacles? This paper describes the application of MBT at Microsoft, discusses the problems in the adoption, and draws some conclusions for the design of MBT tools.[1]

1 Introduction

Testing is one of the most cost-intensive activities in the industrial software development process. Yet, not only is current testing practice laborious and expensive but often also unsystematic, lacking engineering methodology and discipline, and adequate tool support.

Model-based testing (MBT) is one of the most promising approaches to address these problems. At Microsoft, MBT technology has been applied in the production cycle since 1999 [Rob, BGN$^+$03, Sto04, CGN$^+$05]. One key for the relative success of MBT at Microsoft is its attraction for a certain class of well-educated, ambitious test engineers, to which it is one way to raise testing to a systematic engineering discipline.

However, at the larger picture, an estimate based on the number of subscriptions to internal mailing lists for MBT would count only about 10% of product teams which are using or have tried using MBT for their daily tasks. While these numbers can be considered a success compared to other formal quality assurance approaches like verification, they are certainly not indicating a break-through. So what are the obstacles in applying MBT, and how can a larger group of users be attracted to the technology?

This paper attempts to answer this question, based on feedback of the user base of the Spec Explorer tool [CGN$^+$05], its predecessor AsmL-T [BGN$^+$03], and other internal MBT tools at Microsoft. The major issues, apart of the ubiquitous problem in the industry that people do not have enough time to try out new technology and educate themselves, seem to be the steep learning curve for modeling notations together with the lack of state-of-the-art authoring environments, missing support for scenario-based (interaction-based) modeling, thus involving not only the test organization but also other stakeholders in the process, poor documentation of the MBT tools, and last not least technical problems like dealing with state explosion, fine-grained test selection, and integration with test management tools.

The analysis presented led to the design of a new model-based testing tool generation at Microsoft Research. This tool tries to address the identified challenges by providing a full

[1] An extended version of parts of this paper appears in [Gri06]

integration into the development environment of Visual Studio, providing users an authoring experience they are used to. The new tool moreover emphasizes a *multi-paradigmatic* approach to MBT, allowing to describe models on different levels of abstraction, using scenario and state oriented paradigms as well as diagrammatic and programmatic notations, and enabling the combination of those diverse artifacts for a given modeling and testing problem. A full discussion goes behind the scope of this paper; more details can be found in [Gri06], including the semantic foundations in [GKT06].

2 Model-Based Testing at Microsoft

MBT has a long application tradition at Microsoft, and various tools have been and are in use. The first tool, the Test Modeling Toolkit (TMT), was deployed in 1999, and is based on extended finite state machines (EFSM). Microsoft Research deployed two tools, AsmL-T in 2002 [BGN$^+$03] and Spec Explorer in 2004 [CGN$^+$05], both using executable specification languages based on the the abstract state machine paradigm (ASM) [Gur95] as the modeling notation. Other internal tools which have not been published are also around. The general mailing alias used for internal discussion of MBT issues at Microsoft currently exceeds 700 members.

All these tools, though quite different in details and expressiveness, share some common principles. Models are described by *guarded-update rules* on a global data state. The rules describe transition between data states and are labeled with *actions* which correspond to invocations of methods in a test harness or in the actual system-under-test (SUT). Rules can be parameterized (and the parameters then usually also occur in the action labels). A user provides value domains for the parameters, using techniques like pairwise combination or partitioning. In the approach as realized by AsmL-T and Spec Explorer, the parameter domains are defined by expressions over the model state, such that for example they can enumerate the dynamically created instances of an object type in the state where the rule is applied.

Sample A very simple example to demonstrate the basic concepts as they appear in Spec Explorer today is considered. The model describes the *publish-subscribe* design pattern which is commonly used in object-oriented software systems. According to this pattern, various subscriber objects are registered with a publisher object to receive asynchronous notification callbacks when information is published via the publisher object (in fact, the subscribers can dynamically register and unregister at a publisher, but this aspect is simplified here.) Thus this example includes dynamic object creation as well as reactive behavior.

The model is given in Fig. 1 (top). The state of the model consists of publisher and subscriber instances. A publisher has a field containing the set of registered subscribers, and a subscriber has a field representing the sequence of data it has received but not yet handled (its “mailbox”). The model simply describes how data is published by delivering it to the mailboxes of subscribers, and how it is consumed by a subscriber in the order it was published. The precondition of the handling method of the subscriber enables it only if the mailbox is not empty, and if the data parameter equals to the first value in the mailbox. Note that the `Handle` method is an *observable* action, which comes out as spontaneous output from the system under test (SUT).

```
class Publisher {
  Set<Subscriber> subscribers = Set{};
  [Action(ActionKind.Controllable)]
  Publisher(){}
  [Action(ActionKind.Controllable)]
  void Publish(object data)
  {
    foreach (Subscriber sub
                      in subscribers)
      sub.mbox += Seq{data};
  }
}
```

```
class Subscriber {
  Seq<object> mbox = Seq{};
  [Action(ActionKind.Controllable)]
  Subscriber(Publisher publisher)
  {
    publisher.subscribers += Set{this};
  }
  [Action(ActionKind.Observable)]
  void Handle(object data)
  requires mbox.Count > 0 &&
         mbox.Head.Equals(data);
  {
    mbox = mbox.Tail;
  }
}
```

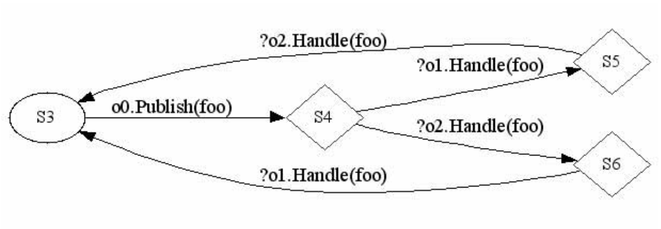

Figure 1: Publisher-Subscriber Model

Fig 1 (bottom) shows an excerpt from the state graph generated by Spec Explorer from this model. This kind of graph corresponds to an *interface automaton* [dAH01]. In this fragment, one publisher and two subscribers are configured (the state graph omits the configuration phase). From state `S3`, a `Publish` invocation is fired, leading to state `S4`, which is an *observation* state where the outgoing transitions are observable actions. The meaning of an observation state is that the SUT has an internal choice to do *one* of the outgoing transitions, as opposed to a control state (`S3`) where it must accept *all* of the outgoing transitions. Thus, effectively, the model gives freedom to an implementation to process the subscribers of a publisher in any given order.

In order to generate the state graph, the model was augmented with further information: the parameters passed to the `Publish` method have been specified (here, `"foo"`), the number of publishers and subscribers to be created has been bounded, as well as the number of messages in the mailbox of a subscriber.

Such state graphs are then input to traversal algorithms to generate a test suite which can be executed offline, or are dynamically traversed using online/on-the-fly testing. For both cases, the test execution environment takes care of binding object values in the model to objects in the implementation, as well as queuing asynchronous observable action invocation events in the SUT for consumption by the model. For details, see [CGN$^+$05].

In practice, models written with Spec Explorer are significantly larger than this simple example; yet they are rarely on the scale of real programs. In the applications at Microsoft, models typically introduce about 10 to 30 actions, with up to 2000 lines of model code, in exceptions over 10000 lines. Yet, these models are able to test features with a code-base which is larger by an order of magnitude or more. This stems from the level of abstraction chosen in modeling. Model-based testing is used for a wide range of application types, including user interfaces, protocols, windows core components, frameworks, device drivers, and hardware abstraction layers.

While in general successfully used in practice, the technology of Spec Explorer, as well of the other available tools at Microsoft, raises some challenges which hinder wider adoption. These will be discussed in the remainder of this paper.

3 The Adoption Problems

Authoring Computer folklore says: “every editor is better than a new editor”. Though clearly this statement is sarcastic, one should not underestimate its wisdom. The author of this paper, for example, used to apply the `vi` editor (a great relic of very early Unix times) for his programming over many years, even though on the development platform Visual Studio was available, which provides automatic indentation, incremental compilation, context-sensitive completion, refactoring, and many more nice features.

When initially rolling out one approaches’ favorite modeling notation to end users, the gravity of habits is even stronger: users are asked to use a new language in an authoring environment which usually does *not* provide the convenience features they are acquainted with from modern environments.

Notations have perhaps become less important today than the environments which support them. This at least applies to users which are heavily using these modern development environments – among those are most younger testers and developers. It might apply less to other stakeholders (like the author of this text, which is still using a `vi` emulation mode under Emacs to write this document in LaTeX).

The lesson learned is that if one comes up with a new notation, one should better be sure that either the users of that notation do not care about modern authoring support, or one should match this support. The later is unfortunately not trivial. The effort for decent authoring support for a language is probably an order of magnitude higher than providing its compiler.

Executable Specifications vs Programming Languages The first generation of the Microsoft Research MBT tools was based on the Abstract State Machine Language (AsmL), a high-level executable specification language. Though the basic concepts of this language seem to be simple and intuitive (it is using a “pseudo-code” style notation and avoids any mathematical symbols), apart of some stellar exceptions, for most testers the learning curve was too steep (see [Sto04] for a discussion).

Testers struggled with concepts like universal and existential quantification and set comprehensions. Under the assumption that the problem was not the concept itself but perhaps the unfamiliar way in which it is presented, the next generation, Spec Explorer, offered in addition to AsmL the Spec# notation, which disguised the high-level concepts in C# concrete syntax. Though this approach was more successful, the basic problems remained. Typically, beginners and even intermediate levels in Spec# prefer to write a loop where a comprehension would be much more natural and concise.

This phenomena is not just explained by the inability of users. It is more the *unwillingness* to learn many new concepts at the same time, in particular if they are not obviously coherent. Confronted with a new technology like MBT and the challenges to understand the difference between model and implementation and finding the right abstraction levels, having in *addition* the challenge to learn a new language, is mastered only by a minority.

The conceptual distance between programming languages like C$\#$ and executable specification languages like Spec$\#$ is shrinking steadily. The new forthcoming C$\#$ version 3.0 will contain – in addition to the relatively declarative notational features C$\#$ has already now – support for comprehension notations (as part of the LINQ project [Mic06]).

The lesson learned here is that it appears wiser to not mix evangelizing executable specification languages with the very core of what model-based testing makes up. This should not mean that those notations do not have a place in MBT – they are indeed rather important. It just means that users should not be *forced* to use a new notation and environment in order to write their first models. Let them use existing programming notations and their authoring environments if they like so. The core of a model-based testing approach and tool should be agnostic about this choice; it should be *multi-paradigmatic*.

Scaling up to Model-Based Development One of the promises of MBT is to be an entry door for model-based development. In course of applying MBT at Microsoft, several test teams have attempted to incorporate program managers, domain experts, business analysts, and the like into the modeling process. This has not been very successful so far, though some exceptions exist.

One interesting observation is that executable specification languages like AsmL, which provide a high-level pseudo-code style notation, are more attractive to those stakeholders than programming-oriented notations like Spec$\#$. AsmL had more users authoring system models, compared to just models for test, whereas with the introduction of Spec$\#$ and Spec Explorer, these applications diminished. This is a strong argument to *continue* supporting high-level executable specification languages like AsmL for MBT (just do not make them the only choice).

However, it seems that the main obstacle here is not the language but the modeling style. AsmL, Spec$\#$, or any of the other MBT approaches used at Microsoft are not attractive in the requirements phase since they are *state-based* instead of *scenario-based.* In this way they represent a design by itself – even if on an higher-level of abstraction. These high-level designs are well suited for analysis, but less well for understanding and communicating usage scenarios. Thus to incorporate stakeholders from the requirements league, scenario-based modeling must be supported.

Scenarios are also heavily used in the test organizations themselves. For example, *test plans* are commonly used at Microsoft to describe (in an informal way) what usage scenarios of a feature should be tested. These test plans, as well as the scenarios coming from the requirements phase, are intrinsically *partial*, omitting a lot of details, in particular oracles, parameter assignments, and so on. It is the job of the test engineers to “implement” these test plans.

The challenge for MBT to scale up to model-based development is the support of both the state-based and the scenario-based paradigm in one approach, where it is possible to *combine* (compose) models coming from those different sources. For example, a scenario might provide the control flow, and a state machine the oracle, and the composition of both produces an instantiated test suite. Thereby, scenarios written in textual as well as diagrammatic languages should be supported.

4 The Technology Problems

State Explosion MBT is famous to easily generate a huge amount of tests from even small models. But this turns out to be more a problem in practice than an advantage, commonly referred to as the "state explosion problem", a number one concern mentioned by users of MBT tools at Microsoft.

The state explosion problem has a number of facets. First, the time required to run a test-suite is a significant cost factor. For example, at Microsoft, developers need to run so-called "basic verification tests" (BVT) before they can submit sources to a shared depot. The time required to run the BVT is important for development productivity. If BVT's require hours to finish, developers tend to submit their changes in larger time intervals, which raises problems with the integration of their changes with other developers changes.

This is also a reason why stochastic on-the-fly/online testing is not the solution for the state explosion problem. It is not realistic to run millions of tests "over night" in the standard development process. Indeed, this kind of testing has its proper use in test deployments which run in test labs asynchronously with the development process and in larger time intervals.

Test Selection The notion of *test selection* is generally used in the MBT community to name the process of selecting some representative set of tests from the model – and thus should provide the tools to battle state explosion. Test selection traditionally covers graph traversal techniques which can be applied to models which are boiled down to some finite state machine representation, as well as techniques for generating parameters of tested actions, like pairwise combination, partitioning, and so on. In the context of models which have an unbounded state space, like Spec Explorer models, test selection can also include bounds, filters, state grouping, and other techniques to prune the state space.

While these techniques are mostly automated and well understood, it is a regular complain of MBT users at Microsoft that they have not enough *fine-grained* control over the test selection process. A typical user problem is for example to choose the set of tests from the model where during some initialization phase an arbitrary path is sufficient, in the operation phase paths should be chosen such that all transitions are covered, and in the shutdown phase again an arbitrary path is good enough. MBT tools need to support this kind of fine-grained control over the test selection process.

Some tools support defining so-called *test purposes* which are combined with the model to slice some desired behavior, using special notations for that [FJJV97, TB03]. Instead of introducing a further notation for describing test purposes, it looks desirable to use models to express test purposes and view the test selection problem with test purposes as a model composition problem. Test purposes then fall together with test plans and requirement scenarios, as discussed previously. Even more than for those applications, models used as test purposes must allow to express partial behavior which omits many details.

Model Analysis Another facet of the state explosion problem is the understanding of what the model actually does. Since models represent human abstractions they can be error-prone, missing some intended behaviors because of over-abstraction. Therefore, they require "debugging". Debugging a model for MBT effectively means exploring and analyzing the state space it spans, both by humans and automatically.

The Spec Explorer tool invests a great lot of detail to support human analysis by its viewing capabilities, which allow to visualize the state space directly or using projection techniques. These capabilities are one major cornerstone for the success of the tool, and need to be maintained and extended.

The Spec Explorer tool also supports model-checking with safety and liveness properties. However, this support is not very well developed in comparison to decent model-checking tools, and temporal property checking is not available. Model checking is a key feature that makes modeling more attractive for stakeholders outside of the test organization, and consequently, user requests for it came from this side.

Test Management Test automation does not end with the generation of test cases. In particular, if it comes to testing of distributed systems and/or testing of software on heterogeneous hardware, *test management* is a significant effort of the overall process.

At Microsoft, a variety of test management tools are in use which allow distribution of test jobs on matching hardware and execution of orchestrated tests inclusive of logging for collecting the test results. Other tools support measuring coverage of test suites. The integration of this set of tools with model-based testing tools is only marginally developed, and an improvement here is an often requested feature. For example, users want end-to-end tracking of test case execution with the model source, test versioning, automatic bug filing, generation of repros for failed test runs, and so on.

5 Conclusion

Model-based testing promises a significant contribution in raising software testing to a systematic engineering discipline, and providing an entry door to model-based development. Its application in internal development at Microsoft for half a decade is considered a success, though a break-through of the technology is not in sight. This paper attempted to identify some of the obstacles for wider adoption of MBT at Microsoft, which are typical at least for software development at enterprise level.

The conclusion drawn is that in order to address different concerns both inside the testing organizations as well as in the broader scope of model-based development, a model-based testing tool and approach should be multi-paradigmatic, supporting programmatic and diagrammatic notations, as well as state-based and scenario-based styles. Programmatic notations with decent authoring support should be provided for test engineers, best using mainstream programming languages, whereas diagrammatic, scenario-based notations as well as executable specification language should be provided for test architects and stakeholders outside of test. Moreover, model-checking should be seen as an integral part of model-based testing tools. A tool which follows these design principles is currently under development at Microsoft Research; an overview can be found in an extended version of this paper [Gri06].

The general message of this paper does not come as a surprise: multi-paradigmatic approaches are ubiquitous in model-based development, as for example reflected in UML. However, model-based testing requires that there are full programmatic notations, and not only diagrammatic ones, and puts strong demands on the semantic and tool-technical integration of the various behavioral notations, requiring them to be composable for a common testing goal. This demand is indeed also a long term goal for model-based development in general – yet the model-based testing application provides very concrete requirements, the implementation of which promises immediate payoff.

References

[BGN+03] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a Tool Environment for Model-Based Testing with AsmL. In Petrenko and Ulrich, editors, *Formal Approaches to Software Testing, FATES 2003*, volume 2931 of *LNCS*, pages 264–280. Springer, 2003.

[CGN+05] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, May 2005. to appear in *Formal Methods and Testing*, LNCS, Springer.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM, 2001.

[FJJV97] J.C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1-2):123–146, 1997.

[GKT06] Wolfgang Grieskamp, Nicolas Kicillof, and Nikolai Tillmann. Action Machines: a Framework for Encoding and Composing Partial Behaviors. Technical Report MSR-TR-2006-11, Microsoft Research, February 2006. to appear in *International Journal of Software & Knowledge Engineering*.

[Gri06] Wolfgang Grieskamp. Multi-Pardigmatic Model-Based Testing. Technical Report MSR-TR-2006-111, Microsoft Research, August 2006. To appear as invited contribution of FATES/RV 2006, LNCS.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Mic06] Microsoft. The LINQ Project. `http://msdn.microsoft.com/data/ref/linq`, 2006.

[Rob] Harry Robinson. Finite State Model-Based Testing on a Shoestring. In *STARWEST 99*. available online.

[Sto04] Keith Stobie. Model Based Testing in Practice at Microsoft. In *Proceedings of the Workshop on Model Based Testing (MBT 2004)*, volume 111 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[TB03] Jan Tretmans and Ed Brinksma. TorX: Automated Model Based Testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.