10 Methodological Issues in Model-Based Testing

Alexander Pretschner[1] and Jan Philipps[2]

[1] Information Security
Department of Computer Science
ETH Zürich
Haldeneggsteig 4, 8092 Zürich, Switzerland
Alexander.Pretschner@inf.ethz.ch
[2] Validas AG
gate
Lichtenbergstr. 8., 85748 Garching, Germany
philipps@validas.de

10.1 Introduction

Testing denotes a set of activities that aim at showing that actual and intended behaviors of a system differ, or at increasing confidence that they do not differ. Often enough, the intended behavior is defined by means of rather informal and incomplete requirement specifications. Test engineers use these specification documents to gain an approximate understanding of the intended behavior. That is to say, they build a mental model of the system. This mental model is then used to derive test cases for the implementation, or system under test (SUT): input and expected output. Obviously, this approach is implicit, unstructured, not motivated in its details and not reproducible.

While some argue that because of these implicit mental models all testing is necessarily model-based [Bin99], the idea of model-based testing is to use *explicit behavior models* to encode the intended behavior. Traces of these models are interpreted as test cases for the implementation: input and expected output. The input part is fed into an implementation (the system under test, or SUT), and the implementation's output is compared to that of the model, as reflected in the output part of the test case.

Fig. 10.1 sketches the general approach to model-based testing. Model-based testing uses abstract *models* to generate traces—test cases for an implementation—according to a *test case specification*. This test case specification is a selection criterion on the set of the traces of the model—in the case of reactive systems, a finite set of finite traces has to be selected from a usually infinite set of infinite traces. Because deriving, running, and evaluating tests are costly activities, one would like this set to be as small as possible.

The generated traces can also be manually checked in order to ascertain that the model represents the system requirements: similar to simulation, this is an activity of *validation*, concerned with checking whether or not an artifact—the model in this case—conforms to the actual user requirements. Finally, the model's traces—i.e., the test cases—are used to increase confidence that the

M. Broy et al. (Eds.): Model-Based Testing of Reactive Systems, LNCS 3472, pp. 281-291, 2005.

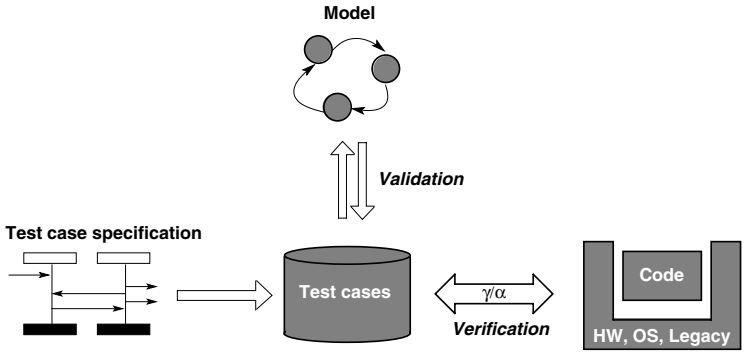

Fig. 10.1. Model-Based Testing

implementation corresponds to the model, or to prove that it does not. Testing is hence an activity of *verification*: the implementation is checked for correctness w.r.t. the model which can arguably be interpreted as a behavior specification, and which represents the formalized user requirements.

This approach immediately raises an array of difficult questions. The issues of test case specifications and generation technology are treated in Chapters 11 and 12 of this book, and are consequently not the subject of this chapter. Instead, we focus on the following two key questions.

(1) Obviously, in the above approach, the model has to faithfully represent the user requirements, i.e., the intended behavior: it has to be valid. Why would one choose to build a costly model, validate it, derive tests and run them on a SUT rather than directly validate the SUT?
(2) Because system and software construction occur—as any engineering activity—under high time and cost constraints, can we build a single model to generate both test cases and production code?

The first question is answered by requiring models to be more abstract, or "simpler", than the SUT. Because they are more abstract, they are easier to understand, validate, maintain, and more likely to be amenable to test case generation. The above approach to model-based testing is then modified as follows. The input part of a model's trace—the test case—is concretized (γ in the figure) before it is fed into the implementation. Conversely, the output of the SUT is abstracted (α in the figure) before it is compared to the output of the model. Note that this approach incurs a cost: aspects of the SUT that were abstracted away can obviously not directly be tested on the grounds of the abstract model.

The second question will be answered by discussing different scenarios of model-based testing that regard different interleavings and flavors of building models and code. Roughly, it will turn out that some sort of redundancy is indispensable: choosing to derive both test cases and implementations from one single model requires one to precisely know what this means in terms of quality assurance: in this way, code generators and assumptions on the environment can be tested.

One might argue that if the model has to be valid anyway, then we could generate code from it without any need for further testing. Unfortunately, this is no viable option in general. Since the SUT consists not only of a piece of code that is to be verified but also of an environment consisting of hardware, operating system, and legacy software components, it will always be necessary to dynamically execute the SUT. This is because the model contains assumptions on the environment, and these may or may not be justified.

Overview

The remainder of this chapter is organized as follows. Sec. 10.2 elaborates on the different levels of abstraction of models and implementations. In Sec. 10.3, we discuss several scenarios of model-based testing and shed light on how to interleave the development of models and of code. Sec. 10.4 concludes.

10.2 Abstraction

Stachowiak identifies the following three fundamental characteristics of models [Sta73].

- Models are *mappings* from a concrete (the "original") into a more abstract (the "model") world;
- Models serve a specific *purpose*;
- Models are *simplifications*, in that they do not reflect all attributes of the the concrete world.

In this section, we take a look at the third point. There are two basic approaches to simplification: omission and encapsulation of details.

10.2.1 Omission of Details

When details are actually discarded, in the sense that no macro expansion mechanism can insert the missing information, then the resulting model is likely to be easier to understand. This is the basic idea behind development methodologies like stepwise refinement, where the level of abstraction is steadily decreased. [1] Specifications at higher levels of abstraction convey the fundamental ideas. As we have alluded to above, they cannot directly be used for testing, simply because they contain too little information. This is why we need driver components that, where necessary, insert missing information into the test cases.

The problem then obviously is which information to discard. That is true for all modeling tasks and, until building software becomes a true engineering discipline, remains a black art which is why we do not discuss it further here.

[1] A similar scheme is also found in incremental approaches like Cleanroom [PTLP99] where the difference between two increments consists of one further component that is 100% finished.

In the literature, there are many examples of abstractions—and the necessary insertion of information by means of driver components—for model-based testing [PP04]. These are also discussed in Chap. 15.

This variant of simplification reflects one perspective on model-based development activities. Models are seen as a means to actually get rid of details deemed irrelevant. As mentioned above, no macro expansion mechanism can automatically insert them, simply because the information is given nowhere. Missing information can, in the context of stepwise refinement, for instance, be inserted by a human when this is considered necessary.

10.2.2 Encapsulation of Details

Details of a system (or of parts a system) can be also be referenced. Complexity is reduced by regarding the references, and not the content they stand for. This second kind of abstraction lets modeling languages appear as the natural extension of programming languages (note that we are talking about behavior models only). The underlying idea is to find ways to *encapsulate* details by means of libraries or language constructs.

Encapsulating the assembly of stack frames into function calls is an example, encapsulating a certain kind of error handling into exceptions is another. The Swing library provides abstractions for GUI constructs, and successes of CORBA and the J2EE architectures are, among many other things, due to the underlying encapsulation of access to communication infrastructures. The MDA takes these ideas even further. Leaving the domain of programming languages, this phenomenon can also be seen in the ISO/OSI communication stack where one layer relies on the services of a lower layer. The different layers of operating systems are a further prominent example.

What is common about these approaches is that basically, macro expansion is carried out at compile or run time. In the respective contexts, the involved information loss is considered to be irrelevant. These macros are not only useful, but they also restrict programmers' possibilities: stack frame assembly, for instance, can in general not be altered. Similarly, some modeling languages disallow arbitrary communications between components (via variables) when explicit connectors are specified. The good news is that while expressiveness—in a given domain—is restricted, the languages become, at least in theory, amenable to automatic analysis simply because of this restriction.

Of course, the two points of view on model-based development activities are not orthogonal. There is no problem with using a *modeling language* in order to very abstractly specify arbitrary systems. In the context of testing, the decision of which point of view to adopt is of utter importance. When models of the pure latter kind are taken into consideration, then they are likely to be specified at the same level of abstraction as the system that is to be tested. We then run into the above mentioned problem of having to validate a model that is as complex as the system under test.

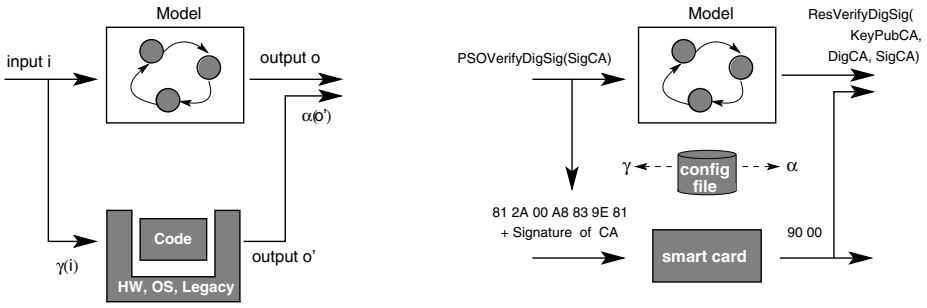

Fig. 10.2. Abstractions and concretizations. The general setting is depicted at the left; an example where α and γ are defined via a configuration file is given at the right.

10.2.3 Concretization and Abstraction

As argued above, with the possible exception of stress testing, it is methodologically indispensable that in terms of model-based testing, models are simplifications of the system under test. Consequently, test cases derived from the model can in general not be directly fed into the system. To adapt the test cases to the system, it is necessary to re-insert the information by means of *driver components*:[2] the input part i of a trace of the model—a test case—is concretized to the level of the implementation, $\gamma(i)$. In general, there will be many choices to select $\gamma(i)$, simply because the model is an abstraction. This choice is left to the test engineer, or a driver component that he or she has to write.

$\gamma(i)$ is then fed into the implementation which reacts by outputting some o'. By construction, o' is not at the same level of abstraction as the output of the model, o. Unfortunately, we cannot in general use γ to compare $\gamma(o)$ to o'. The reason is that as in the case of concretizing input, there are many candidates for $\gamma(o)$, and for comparing the system to the model, a random choice is not an option here.

The classical solution to this problem is to use an abstraction function, α, instead. Since α is an abstraction function, it itself involves a loss of information. Provided we have chosen γ and α adequately, we can now apply α to the system's output and compare the resulting value $\alpha(o')$ to the model's output. If $\alpha(o')$ equals o, then the test case passes, otherwise it fails. In the driver component, this is usually implemented as follows: it is checked whether or not the implementation's output is a member of the set of possible implementation outputs that correspond to the model's output (of course, in non-deterministic settings it might possible to assign a verdict only after applying an entire sequence of stimuli to the SUT).

The idea is depicted in Fig. 10.2, left. Note that the general idea with pairs of abstraction/concretization mappings is crucial to many formally founded ap-

[2] Of course, this concretization may also be performed in a component different from the driver.

proaches to systems engineering that work with different levels of abstraction [BS01b].

Example. As an example, consider Fig. 10.2, right. It is part of a case study in the field of smart card testing [PPS+03]. The example shows a case where the verification of digital signatures should be checked. In order to keep the size of the model manageable, the respective crypto algorithms are not implemented in the model—testing the crypto algorithms in themselves was considered a different task. Instead, the model outputs an abstract value with a set of parameters when it is asked to verify a signature. By means of a configuration file—which is part of the driver component—, the (abstract) command and its parameters are concretized and applied to the actual smart card. It responds `90 00` which basically indicates that everything is alright. This value is then augmented with additional information from the configuration file, e.g., certificates and keys, and abstracted. Finally, it is compared to the output of the model.

Further examples for different abstractions are given in Chap. 15.

10.3 Scenarios of Model-Based Testing

Model-based testing is not the only use of models in software engineering. More common is the constructive use of behavior models for code generation. In this section we discuss four scenarios that concern the interplay of models used for test case generation and code generation. The first scenario concerns the process of having one model for both code and test case generation. The second and third scenarios concern the process of building a model after the system it is supposed to represent; here we distinguish between manual and automatic modeling. The last scenario discusses the situation where two distinct models are built.

10.3.1 Common Model

In this scenario, a common model is used for both code generation and test case generation (Fig. 10.3).

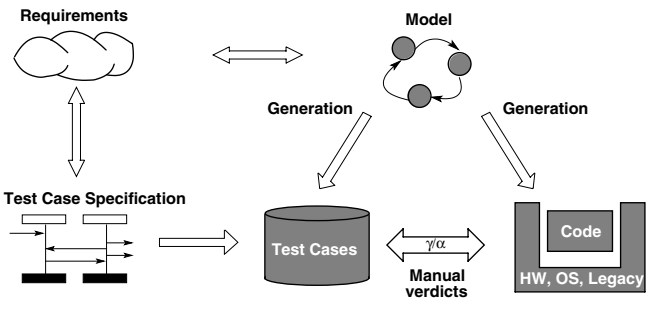

Fig. 10.3. One model is used for both code and test case generation

Testing always involves some kind of redundancy: the intended and the actual behaviors. When a single model for both code generation and test case generation chosen, this redundancy is lacking. In a sense, the code (or model) would be tested against itself. This is why no automatic verdicts are possible.

On the other hand, what can be automatically tested are the code generator and environment assumptions that are explicitly given, or implicitly encoded in the model. This can be regarded as problematic or not. In case the code generator works correctly and the model is valid, which is what we have presupposed, tests of the adequacy of environment assumptions are the only task necessary to ensure a proper functioning of the actual (sub-)system. This is where formal verification technology and testing seem to smoothly blend: formal verification of the model is done to make sure the model does what it is supposed to. Possibly inadequate environment assumptions can be identified when (selected) traces of the model are compared to traces of the system. Note that this adds a slightly different flavor to our current understanding of model-based testing. Rather than testing a system, we are now checking the adequacy of environment assumptions. This is likely to be influential w.r.t. the choice of test cases.

Depending on which parts of a model are used for which purpose, this scenario usually restricts the possible abstractions to those that involve a loss of information that can be coped with by means of macro expansion (Sec. 10.2).

10.3.2 Automatic Model Extraction

Our second scenario is concerned with extracting models from an existing system (Fig. 10.4). The process of building the system is conventional: somehow, a specification is built, and then the system is hand coded. Once the system is built, one creates a model manually or automatically, and this model is then used for test case generation.

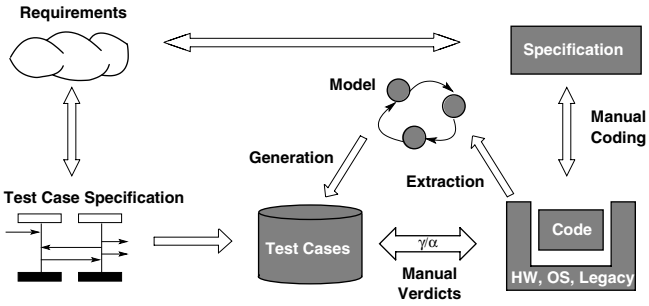

Fig. 10.4. A model is automatically extracted from code

Automatically extracting abstractions from code or more concrete models is a rather active branch of computer science [Hol01, GS97, SA99] which we will not discuss here. The abstractions should be created in a way such that at least some—and identifiable—statements about them should carry over to

the more concrete artifact. In the context of testing, it is important to notice that we run into the same problem of not having any redundancy as above. The consequence is that automatic verdicts make statements only about assumptions in the automatic process of abstraction.

Abstractions are bound to a given purpose [Sta73]. Automatic abstraction must hence be performed with a given goal in mind. It is likely that for test case generation, fully automatic abstraction is not possible but that test engineers must provide the abstraction mechanism with domain and application specific knowledge.

10.3.3 Manual Modeling

A further possibility consists of manually building the model for test case generation, while the system is again built on top of a different specification (Fig. 10.5). Depending on how close the interaction between the responsibles for specification and model is, there will in general be the redundancy that is required for automatically assigning verdicts.

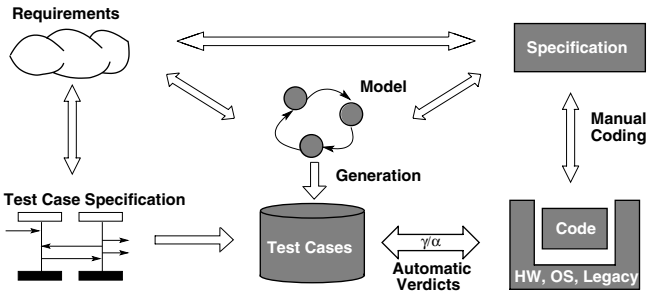

Fig. 10.5. A model is built only for testing purposes

This approach also reflects the situation where building the specification and implementing a system are not necessarily performed by the same organization. For instance, this is often the case in the automotive industry where OEMs assemble devices from different suppliers. Obviously, the OEMs are interested in making sure that the supplied systems conform to the specification.

As an aside, combinations of this scenario and that of the last subsection typically arise when test case generation technology is to be assessed (a recent survey contains some examples [PP04]). Doing so, however, is problematic in that testing is only performed when the system has, in large parts, already been built.

10.3.4 Separate Models

Finally, a last scenario is noteworthy that involves having two redundant and distinct models, one for test case generation, and one for code generation (Fig. 10.6).

This approach allows one to have automatic verdicts. The model for development may be as abstract as desired when the requirement for automatic code generation is dropped.

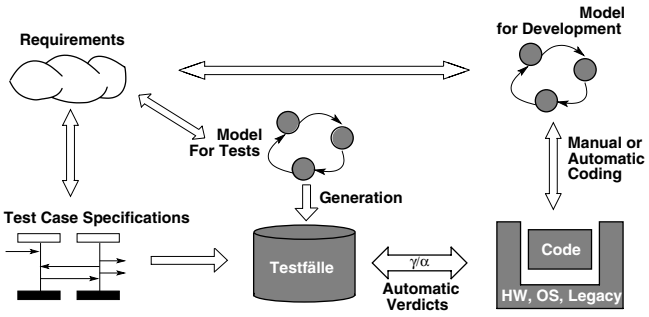

Fig. 10.6. Two models

10.3.5 Interleaving

Except for the last scenario, the above scenarios share the commonality that there is no real interaction between the development processes of the models and that of the code. In iterative development processes with ever changing requirements, this seems unrealistic. With suitable definitions of what an increment is, it is of course possible to interleave the development of two models, or to interleave the development of a model and some code. Of course, this is likely to involve some overhead. We will not discuss this issue any further here since that has been done elsewhere [PLP03] with considerations of the role of regression testing and of compositional testing [Pre03].

10.3.6 Summary

Automatic code generation from models boils down to perceiving models as possibly executable artifacts written in a very high-level programming language. This goes well beyond the use of models for analytical purposes only where, again, it is widely accepted that while it might be too expensive, modeling in itself usually reveals many errors. Currently, the embedded systems industry expresses a high degree of interest in these concepts.

We have shown that one must be careful in ensuring redundancy when models are used for testing and code generation. Models for the further can involve both flavors of simplification that we identified in Sec. 10.2, namely the one where information is encapsulated, and the one where information is deliberately dropped. Models for the latter can obviously only involve encapsulation of

details.[3] We consider a thorough discussion of when the use of models for code generation is likely to pay off utterly important but beyond the scope of this paper. Briefly, we see a separation of concerns, multiple views, and restriction as key success factors of modeling languages [SPHP02]. The following captures the essence of the four scenarios and provides a prudent assessment.

- Our first scenario considered one model as the basis for code and tests. This is problematic w.r.t. redundancy issues and a restriction to abstractions that boil down to macros. Code generators and environment assumptions can be checked.
- The second scenario discussed the automatic or manual extraction of abstractions (beyond its technical feasibility). Because there is no redundancy either, the consequences are similar to those of the first scenario.
- The third scenario discussed the use of dedicated models for test case generation only. Because there is redundancy w.r.t. a manually implemented systems and because of the possibility of applying simplifications in the sense of actually losing information, this scenario appears promising. This is without any considerations of whether or not it is economic to use such models. We will come back to this question in the conclusion in Sec. 10.4.
- Finally, the fourth scenario considered the use of two independent models, one for test case generation, and one for development. The latter model may or may not be used for the automatic generation of code. This scenario seems to be optimal in that it combines the—not yet empirically verified—advantages of model-based testing and model-based development. Clearly, this approach is the most expensive one.

10.4 Conclusion

In this brief overview chapter, we have discussed the role of models in the context of testing. Some emphasis was put on a discussion on the methodological need for different abstraction levels of models and implementations. The basic argument is that the effort to manually validate an SUT—checking whether or not it corresponds to the usually informal requirements—must find itself below the effort necessary to build the model, validate the model, and derive test cases from it. Abstract models are easier to understand than very concrete artifacts. On the other hand, abstraction incurs a cost: aspects that were abstracted can usually not be tested. We discussed the role of driver components that, to a certain extent, can re-introduce the missing information.

A further focus of this article is on different scenarios of model-based testing. We have discussed the role of redundancy and the problematics of generating both tests and production code from one single model.

[3] When they focus on certain parts of a system only, then this clearly is a loss of information. However, code can obviously only be generated for those parts that have been modeled.

The fundamental concern of model-based testing seems to be whether or not it is more cost-effective than other approaches—traditional testing, reviews, inspections, and also constructive approaches to quality assurance. While first evaluations of model-based testing are available [PPW+05], there clearly is a need for studies that examine the economics of model-based testing.