Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
Electronic Notes in Theoretical Computer Science 116 (2005) 59–71
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Abstractions for Model-Based Testing[1]

Wolfgang Prenninger and Alexander Pretschner[2]

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany

Abstract

The idea of model-based testing is to compare the I/O behavior of an explicit behavior model with that of a system under test. This requires the model to be valid. If the model is a simplification of the SUT, then it is easier to check the model and use it for subsequent test case generation than to directly check the SUT. In this case, the different levels of abstraction must be bridged. Not surprisingly, experience shows that choosing the right level of abstraction is crucial to the success of model-based testing. We argue that models for specification purposes, models for test generation, and models for full code generation are likely to be different. The paper classifies and discusses different abstractions. It is intended as a step towards guidelines for those who build behavior models to the end of testing.

Keywords: Model-based testing, SUT, verification, specification

1 Introduction

The basic idea of verification is to compare an abstract specification to a concrete implementation. If the two systems are given by means of formalized transition systems, then the aim of formal verification is to establish or refute some simulation relationship, or show that both satisfy certain properties. The specification may also be given as a set of properties. In this case, formal verification amounts to checking whether or not the implementation satisfies these properties. If specification and implementation aren't too complex, then techniques like model checking or deductive theorem proving can be applied.

[1] Supported by the DFG within the priority program *Softspez* (SPP 1064), project *InTime*.
[2] Email: {prenning,pretschn}@in.tum.de, fax +49 89 289 17307

doi:10.1016/j.entcs.2004.02.086

In addition to technical challenges, this gives rise to two observations. First, it is not clear how fine-grained a specification should be. Obviously, it must be detailed enough to yield meaningful results when compared to an implementation. Furthermore, it must reflect the customer requirements, i.e., it must be valid. In the following, we stick to the terminology that validation is concerned with checking an artifact against requirements in the customers' heads whereas verification is concerned with the comparison of two more or less formal artifacts. This is regardless of whether or not the comparison is achieved with mathematical rigor.

Second, techniques like model checking and theorem proving are performed on abstractions. These are given in some form of formalized transition systems, or, more general, *behavior models*.[3] The assumption then is that the piece of software that is to be checked is embedded into a *correctly functioning* environment that meets the tacit assumptions in the model. Now, the environment may be very complex, and formal verification technology cannot ensure whether or not the environment behaves the way it is anticipated in the system to be checked. The point is that verification technology like model checking or deductive theorem proving necessarily operates on abstractions. (Our notion of) testing operates on actual devices in their actual environments, and both activities are necessary.

The fundamental issue boils down to abstraction: checking the abstraction of a system is not enough if the environment is not well-understood, and abstraction must be employed with care in order to provide useful specifications. Note that we use the term "abstraction" in two senses. One denotes the simplified artifact, or model. The other use is concerned with the activity of establishing this simplification. We address these issues in the context of model-based testing.

Our understanding of model-based testing is that we use an abstract behavior model to test a concrete implementation. Abstract models are simplifications, and they are hence easier to grasp. It seems therefore likely that validating such an abstract model is easier than validating the concrete implementation itself. Once the model is considered valid, we use it for automatic test case generation. This means that certain—possibly more than one could manually create—traces of the model are automatically generated. The necessary selection criterion is given by a test case specification, and it is concerned with functional, structural, and random tests (Fig. 1, left; cf. [11]). As a consequence of the different abstraction levels, these test cases cannot be applied directly to the implementation. The input part of the traces is concretized and applied to the implementation. The implementation's output is abstracted

[3] In some cases, these abstractions can be automatically extracted from code.

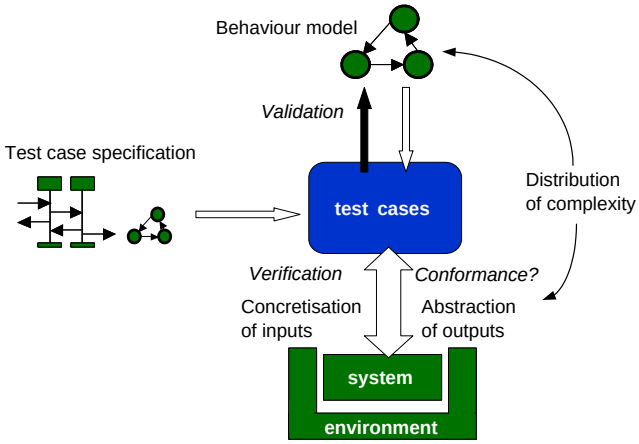

Fig. 1. Model-based test process

and then compared to the output of the model as given by the trace (Fig. 1, bottom). Complexity is hence distributed between the abstract model and the component that takes care of concretizing and abstracting signals (Fig. 1, right).

Some case studies that we cite and discuss in Sec. 3 conform to the above approach. In this paper, we address the question of *which abstractions* (in the sense of structuring and simplifying systems) may be chosen to the end of testing. The idea is to present a first step towards an engineering discipline of model-based testing. While we believe that *abstractions* for testing are similar to the *abstractions* that may be used for code generation, we'll argue that behavior *models* for testing are different from behavior *models* for code generation and specification.

Contribution and organization

The contribution of this paper is a catalog and discussion of applicable abstractions for model-based testing. We also relate models for testing purposes to other models in the development process. Sec. 2 sheds light on the nature of abstraction in the context of model-based development and model-based testing. Sec. 3 reviews abstractions for testing that are used in the literature. Sec. 4 discusses limitations of different abstractions, and Sec. 5 concludes. Related work is cited in the respective context.

2 Abstractions

Arguably, abstraction is the most important tool computer engineers possess. Its aim is the simplification of a complex problem or system. Simplification takes place w.r.t. certain goal, be it analysis, generation, or basis for communication. Again, we'll call the result of an abstraction process a *model*. In this sense, Stachowiak [13, p.131-132] identifies three characteristics of *models*: mapping—models are models of something, simplification—models do not catch all attributes of the original they represent, and pragmatics—models aren't related to the respective original in a unique sense but fulfill certain goals. Models are images or archetypes. Before turning to some concrete abstractions in the context of testing in the next section, we discuss the role of abstractions in a more general manner. Throughout this paper, we won't be concerned with concrete modeling languages.

Information that can be inserted automatically

A system description at the abstract model level contains considerably less information than at the detailed realization level. Today's most successful abstraction techniques are based on compilers or libraries that support the automatic translation of the model to a realization by relying on some form of macro mechanism. For instance,

- procedures abstract from concrete stack frames in programming languages,
- the Swing-API in Java abstracts from the plethora of instructions which were formerly necessary to program GUIs,
- the ISO-OSI model abstracts from communication details—an instruction on a higher layer corresponds to a complex interaction at the lower layers,
- J2EE and Corba are further examples for abstraction from communication details that are inserted by the respective compilers.

Domain specificity

The abstraction gap that is resolvable by compilers and linkers is more or less specific to a domain. While the concept of procedures is common to all programming languages, the Swing API is restricted to the domain of GUI programming, the ISO-OSI stack applies to communication, and the MDA is arguably concerned with business information systems rather than embedded real-time systems. It is certainly one of the most challenging and demanding tasks to develop abstraction techniques which are more or less domain specific and can be automatically translated to the concrete level using compilers, linkers, and the like.

Actually missing information

There are abstractions in models that cannot be resolved automatically by a compiler but are nevertheless useful. There is an inherent complexity in systems which cannot be simplified and abstracted away by means of languages/compilers only—a variation of the popular remark by Brooks that complexity is an essential rather than an accidental property [3].

It is obvious that no full code generation is possible from structural abstractions (architectures), and this also applies to many behavior abstractions that have been built to a well-defined end. For instance, the model of a chip card described by Philipps and Pretschner [11] concentrates on the protocol flow and abstracts from the content of files and the impact of operations on them. This information is simply not included in the model to keep the model manageable. Without further information, a static compiler cannot instantiate an abstract file to a concrete one and its dynamic evolution over time. Clearly, this does not mean that domain-specific languages cannot exist where "file system handling" is one predefined concept and where a compiler inserts all necessary information. A further example for actually missing information is that some models abstract from concrete timing behavior. Static translation by a compiler generally cannot instantiate the most diverse timing behavior of a realization at the concrete level.

Purposes

To summarize, abstractions occur in two forms. They can be simplifications where missing information can be inserted automatically, and they can be simplifications where information is deliberately missing in order to keep the model simple. Simplifications tend to be domain-specific, and the ultimate goal of model-based development seems to lie in finding these abstractions that are applicable to all systems of a given domain.

Abstracting from certain details makes sense only with a given purpose in mind. Different abstractions are used for different purposes. Abstraction takes place to the end of (a) obtaining or enhancing knowledge of a system—which entails abstractions of both a model and its environment, (b) specification of a system, (c) encapsulated access to parts of a system (e.g., via library calls or binding with external components or services), (d) communication between developers, (e) generation of code, and (f) testing systems.

Model-based testing

Model-based testing makes use of both kinds of abstractions, those that involve an actual loss of information and those that don't. If a (behavior) model is used to the end of testing, then it must be valid. That is to say, it

must be a faithful image of the requirements. *Now, if the model contains sufficient information for (production) code generation, then validating the model is likely to be as resource-consuming as directly validating an implementation that has been built without a model.* The reason is that under these conditions, modeling languages actually are programming languages. Models become beneficiary when they are amenable to intellectual mastery or automatic analysis, and this is usually the case if they are less complex than the systems—if they are actual abstractions.

If one wants to test only certain parts of the system, then one can omit others from the model. For instance, this is the case for the above mentioned file system of a chip card. However, not all simplifications will be defined by omitting entire parts of the functionality of a system. In the chip card example, cryptographic operations have not been implemented in the model for they'd be too complex for automatic analysis. Instead, they were abstracted in a way that an external driver component could insert the missing information. The model is hence simple, but it is too simple to be directly usable for testing. In this case, missing information is hence not inserted by a compiler or linker, but by the driver component.

Test models and specification models

Systems under development tend to be described by specification documents that are comprehensive but informal, incomplete and sometimes ambiguous. Theoretically, one would like to have a—possibly executable—and complete specification model that comprehensively, precisely and completely specifies the system. This specification model could be the basis for test case generation to the end of verifying conformance between specification and implementation.

Now, real world specifications become very large so that it would be too expensive to build and maintain specification models which contain all specification aspects—these models would already constitute prototypical implementations. To deal with this dilemma, one builds test models in addition to informal and incomplete specifications. Test models reduce the comprehensive specification to crucial parts which are implemented abstractly, completely, unambiguously. Exactly *which* parts or aspects are crucial for testing is a result of engineering experience.

By abstractly specifying the crucial parts, e.g., those that are error-prone by experience, in rather small test models, one gains the advantages of small and manageable models. Models that can be used for full production code generation, on the other hand, can be seen as implementations, and it is a dubious endeavor to extract code and test cases from the same model.

3 Application

In this Section, we present different abstraction principles that are applied in the literature on model-based testing. We reviewed case studies on model-based testing of processors [5,12,7], smart cards [11,4], protocols [9,1], Java and POSIX [6], by taking into account related work [2,10,8]. We identified five principles: *functional, data, communication, and temporal* abstractions. These principles cannot always be distinguished sharply, and so they are often applied in combination. The following overview may serve as a reference for modelers who face the challenging task of finding adequate abstractions that reflect crucial parts of the SUT's model which, in turn, serves as reference for verifying the SUT's behavior. Being used for testing, all of the following abstractions involve a loss of information and can therefore test only those parts that are specified.

3.1 Functional Abstraction

The purpose of functional (or behavior) abstraction is to concentrate on the "main" functionality of the SUT which has to be verified. This leads to an omission of cumbersome details in the model that are not in the focus of the verification task. [4] By doing so the model often implements only parts of the complete behavior determined in specification documents, i.e., the model does not completely define the intended behavior of the SUT but models significant aspects only. In other words the model specifies the behavior of the SUT under a constrained environment. In addition, functional abstraction supports the model-based testing process: if the SUT's functionality can be divided into independent parts, one can build a separate model for each part in order to verify each functionality separately.

Examples for functional abstraction

The case study described in [11] concentrates on testing the protocols between a smart card and its environment, a terminal. Therefore the model abstracts from the complex realization of all cryptographic functions implemented by a smart card. These functions and their responses are represented only symbolically by yielding data of type `encryptedData` when a command `encrypt(data)` is issued. No cryptographic computations are performed in the model. Instead, these computations are performed at the level of the driver component (see above).

[4] This does *not* necessarily mean that special cases are omitted—if these have to be tested, they have to be modeled.

The approach described in [6] uses separate models for testing different functionalities of the POSIX standard. The first model was developed for testing the byte range locking interface `fcntl`. This interface provides control over open files in order to deal with processes which are accessing the files. The model restricts the POSIX standard by allowing the extension of a file only once. The paper mentions a second model which was developed for testing the POSIX `fork()` operation.

Other examples for partial modeling of the behavior of the SUT are that the model does not determine its behavior in certain states for some input values, or the model abstracts completely from exception handling.

3.2 Data Abstraction

The idea of data abstraction is to map concrete data types to logical or abstract data types in order to achieve a compact representation or a reduction of data complexity at the abstract level. A frequently cited example for data abstraction is to represent binary numbers by integers at the abstract level. However, this example changes only the representation of numbers but does not cause any information loss between the levels of abstraction. A common data abstraction technique with information loss is to represent only equivalence classes of concrete data values in the model. Examples that involve information loss are described below. As mentioned above the key of data abstraction is to construct a mapping between concrete and abstract data types and their elements. Since the abstract data types are used to specify the behavior of the model one test goal is that the operations performed by the SUT on concrete values are correctly implemented with respect to the (abstract) operations on abstract values in the model.

Examples for data abstraction

In [5], the SUT is the Store Data Unit (SDU) of a Digital Signal Processor (DSP). The behavior of the DSP depends heavily on the fill level of an SDU's queues. Therefore the status of a queue has been represented in the model by the abstract data type `empty`, `valid`, `quasifull` or `full`. Then testing discovered a performance bug in the SUT because it turned out that the SDU fills its queues only to quasifull status and does not exploit the full capacity of the queues.

The radical abstraction from files for smart card testing [11] was described above. Another example for data abstraction can be found in [12,11]. There, the data types of operands are abstracted to equivalence classes used in the model. For test case instantiation, these symbolic operands are substituted

by concrete values which are randomly selected or determined by means of a configuration file.

Data abstraction is often motivated by functional abstraction. Hudak et al. describe a network system [8]. At the concrete level a network package consists of destination address, source address, message body, checksum etc. The functional abstraction is determined by the goal of verifying the routing mechanism. Consequently, the contents of a network package is reduced to destination address and checksum at the abstract level, by ignoring the other contents of a package.

3.3 Communication Abstraction

The most prominent application of the communication abstraction principle is the ISO-OSI reference model. Here a complex interaction at a concrete level is abstracted to one operation or message at the more abstract level. At the abstract level, this operation is treated atomically. This is even though in general the corresponding operation at a concrete level can be interleaved with other operations. Using this abstraction principle, one can aggregate handshaking interactions or sequences of causally dependent operations to one operation at an abstract level. For test case instantiation these abstract operations can simply be substituted by the corresponding interaction. For building models, communication abstraction is often combined with functional abstraction.

Examples for communication abstraction

In hardware verification [2] and processor testing [5], a concrete aggregation of pin values, several consecutive signals of different buses, or recurring sequences of processor instructions are abstracted to one symbolic operation in the model.

In protocol testing [9], causally dependent operations concerning the same transaction are collapsed into one atomic operation in the model although the concrete representation can be interrupted by other operations.

Sometimes communication abstraction is combined with data abstraction: in [11] the concrete byte string commands of a smart card (represented by sequences of hex numbers) are abstracted to symbolic and human readable messages in the smart card model.

3.4 Temporal Abstraction

The idea of temporal abstraction is that only the ordering of events is relevant for modeling, i.e., the precise timing of events at the concrete level is

deemed irrelevant. We consider abstractions in which the ordering of certain events is irrelevant—as used with partial order reductions—as communication abstractions.

One kind of temporal abstraction is that the model and the SUT use different granularities of discrete time. Then the granularity of time at the abstract level is coarser than it is at the concrete level. Ideally, a mapping between abstract time steps and the corresponding intervals of lower level time steps is given. This can be a challenging task because in many cases one abstract step does not always correspond to a constant or predictable time interval at the concrete level. This form of temporal abstraction is often used in hardware verification and testing [5,12,2,10] by relating one clock cycle in the model to many clock cycles at the implementation level. Temporal abstraction can be effectively combined with communication or/and functional abstraction.

Another form of temporal abstraction is that the model abstracts from physical time. For example, the concrete implementation may depend on using a timer of 250 ms duration. This timer is abstracted in the model by introducing two symbolic events—one for starting the timer, and one for indicating expiration of the timer. By doing so, the physical duration of the timer is abstracted away even if the duration of certain timers changes over runtime at the concrete level.

One might well argue that this kind of abstraction is the special case of a more general abstraction, namely abstraction from quality-of-service.

4 Limitations

In this Section, we illuminate the limitations and consequences that come up with using abstractions in models for testing a SUT's behavior. It is crucial that the modeler is aware of the limitations, and therefore decides for the right trade off between abstraction and precision when building appropriate models for testing the critical aspects of the SUT. As stated in Section 2, there is an inherent complexity in real systems. If some of them are abstracted in the model there is no direct way to detect faults in the SUT concerning these abstractions. In the following we mention a few limitations and consequences.

Often enough, models suffer from a more or less distinct and implicit "happy world assumption" which stems from functional abstraction. A typical example is that models assume that parameters of input messages or input operations are within allowed bounds or have the permitted length or arity. With the aid of such models, it is not possible to test the SUT's behavior if it receives messages with illegal parameters. For example, in the smart card do-

main, an operation including its operands is represented by a byte string at the concrete level. In the model, an operation and its operands are conveniently symbolized by a string (a name). With this kind of model, the behavior of the smart cannot be tested directly if it receives byte string which are for example one byte too short or too long, respectively. It is up to the test engineer to decide whether to cope with such illegal input at the level of the model, or at the level of the driver component.

Intense data abstraction can lead to information loss that cannot be coped with for test case generation. For example we recall the smart card model abstracting from file contents [11] already discussed in Sec. 2. With this model only static properties like file length but not the dynamic evolution of file contents could be verified.

It is hard to detect feature interaction when functional abstraction is applied to build separate models for testing distinct functionalities. For instance, in [6], separate models are used to test different operations of the POSIX standard. These models help to verify the correct functioning of these operations in a stand-alone manner, but leave open the detection of unmeant behavior caused by feature interaction of these operations.

Finally, problems can arise if temporal abstraction is intensively used in the model. Obviously, in the domain of distributed real time systems a rigorous use of temporal abstraction can prohibit the detection of faults which stem from the intricate interleaved timing behavior of the separate components. As a counterexample, in [5], temporal abstraction is explicitly not used to test a processor. In order to trace generated test cases and to check the expected performance a clock cycle in the model corresponds exactly to a clock cycle in the real processor design.

Nevertheless, w.r.t. these limitations, we believe that model based testing is one of the most promising methodologies which will scale up for verification of complex systems in the near future and additionally provide a well structured process for increasing the quality of hard- and software systems. Note that some see testing as model-based by definition: testing is always performed with a more or less explicitly defined intended behavior, i.e., a model.

5 Conclusion

Many activities of software engineering involve abstractions. Today, abstractions are predominantly used as language constructs (e.g., procedures), transparently moved into the runtime environment (garbage collectors), used in the form of interfaces to libraries, components, or communication infrastructure, and as architectures. Abstractions related to the behavior of a system

begin to play a more prominent role when explicit *models* are used. Loss of information can sometimes be compensated with suitable code generators or linkers. For many purposes, consciously discarding information that cannot be inserted automatically is also necessary.

For rather comprehensive testing, models are often too vague for they specify only partial behaviors. If they are more precise, they can be used to the end of testing. If these test models are sufficiently concrete to generate (production) code from them, then one must ask why validating the model and generating test cases from it is preferable to directly validating the SUT. As a consequence, the level of abstraction of test models lies somewhere between specifications and models for full code generation (i.e., a faithful representation of the implementation itself).

On the one hand, the specification describes the system under development in an abstract and comprehensive way. This is usually done in an informal and incomplete manner. The informal and incomplete specification is needed to build test models in the first place: test models are then refinements of the specifications. On the other hand, test models abstractly implement only parts of the more comprehensive specification, but these parts are implemented completely and unambiguously (functional abstractions, cf. Sec. 3). Test models can then be perceived as a supplement to specification documents which completely specifies the crucial parts of the system for verification. In principle, specifications and test models could be built at the same level of abstraction, completeness, and precision, but then one must solve the question whether a supplier or an OEM builds the model. This is, for instance, the situation in the automotive industries.

We have provided a list of abstractions that are used to the end of model-based testing in the literature. They can be summarized as related to function, data, communication, and time (or, more general, quality-of-service). The purpose of this catalog is to equip model-based testers with some guidance when building their models. We did not address the hard question of which traces (test cases) to select, i.e., the question of quality of a test suite. Building adequate models and identifying relevant properties appear to be the key challenges with model-based testing, not test case generation technology. Whether or not the use of dedicated explicit test models pays off economically appears likely, but empirical evidence is still lacking.

References

[1] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196.

Kluwer Academic Publishers, 1999.

[2] D. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. Ip, W. Paulsen, J. Pierce, J. Rose, D. Shea, and K. Whiting. The transaction-based verification methodology. Technical report, Cadence Design Systems, Inc., August 2000.

[3] F. Brooks. No Silver Bullet. In *Proc. 10th IFIP World Computing Conference*, pages 1069–1076, 1986.

[4] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated Test and Oracle Generation for Smart-Card Applications. In *Proc. E-smart*, pages 58–70, 2001.

[5] J. Dushina, M. Benjamin, and D. Geist. Semi-Formal Test Generation with Genevieve. In *Proc. DAC*, 2001.

[6] E. Farchi, A. Hartman, and S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[7] L. Fournier, A. Koyfman, and M. Levinger. Developing an Architecture Validation Suite—Application to the PowerPC Architecture. In *Proc. 36th ACM Design Automation Conf.*, pages 189–194, 1999.

[8] John Hudak, Santiago Comella-Dorda, David P. Gluch, Grace Lewis, and Chuck Weinstock. Model-based verification: Abstraction guidelines. Technical Report CMU/SEI-2002-TN-011, Carnegie Mellon University, October 2002.

[9] H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In A. Petrenko and N. Yevtushenko, editors, *IFIP TC6 11th International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1998.

[10] T. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 129–157, Boston, 1988. Kluwer Academic Publishers.

[11] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. In *In Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003. to appear.

[12] J. Shen and J. Abraham. An RTL Abstraction Technique for Processor Micorarchitecture Validation and Test Generation. *J. Electronic Testing: Theory&Application*, 16(1-2):67–81, February 1999.

[13] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.