Position Paper: Model-Based Testing

Mark Utting
The University of Waikato, New Zealand
Email: marku@cs.waikato.ac.nz

1 Introduction

This position paper gives an overview of model-based testing and discusses how it might fit into the proposed grand challenge for a program verifier.

Model-based testing [EFW02, BBN04, LPU04] is a break-through innovation in software testing because it completely automates the validation testing process. Model-based testing tools automatically generate test cases from a model of the software product. The generated tests are executable and include an oracle component which assigns a pass/fail verdict to each test.

Model-based testing helps to ensure a repeatable and scientific basis for product testing, gives good coverage of all the behaviors of the product and allows tests to be linked directly to requirements. Intensive research on model-based testing in the last 5-10 years has demonstrated the feasibility of this approach, shown that it can be cost-effective, and has developed a variety of test generation strategies and model coverage criteria. Some commercial tools have started to emerge, from the USA (T-Vec, Reactive Systems, I-logix), and also from Europe (Conformiq, Leirios Technologies, Telelogic), as well as a wide variety of academic and research tools [BFS05].

The discussion in this paper is limited to functional testing, rather than the more specialist areas of testing real-time software or concurrent software, because model-based testing is less mature in these areas.

The paper gives an overview of the variety of methods and practices of model-based testing, then speculates on how model-based testing might promote or complement the program verifier grand challenge.

2 Overview of Model-Based Testing

Model-based testing is the automation of black-box test design. It usually involves four stages:

1. **building an abstract model of the system under test.** This is similar to the process of formally specifying the system, but the kind of specification/model needed for test generation may be a little different to that needed for other purposes, such as proving correctness, or clarifying requirements.

2. **validating the model (typically via animation).** This is done to detect gross errors in the model. This validation process is incomplete of course, but this is less crucial in this context, than in the usual refinement-to-code context. With model-based testing, if some errors remain in the model, they are very likely to be detected when the generated tests are run against the system under test (see below).

3. **generating abstract tests from the model.** This step is usually automatic, but the test engineer can control various parameters to determine which parts of the system are tested, how many tests are generated, which model coverage criteria are used etc.

4. **refining those abstract tests into concrete executable tests.** This is a classic refinement step, which adds concrete details missing from the abstract model. It is usually performed automatically, after the test engineer specifies a refinement function from the abstract values to some concrete values, and a concrete code template for each abstract operation.

After this, the concrete tests can be executed on the system under test, in order to detect *failures* (where the outputs of the system under test are different to those predicted by the tests).

It is the redundancy between the test model and the implementation that is important. Experience shows that failures that occur when the tests are run are roughly equally likely to be due to errors in the model or errors in the implementation.

So the process of model-based testing provides useful feedback and error detection for the requirements and the model, as well as the system under test.

The remainder of this section gives a brief overview of the variety of model-based testing practices, under five headings.

Nature of the Model: The model used for test generation can be a functional model of just the system under test, or of the environment of the system (capturing the ways in which the system will be used), or (more usually) a model of both the system and its environment. Models of the system are useful for predicting the outputs of the system, which allows test oracles to be generated, while models of the expected environment are useful for focussing the test generation on an expected usage of the system.

Model Notation: Almost all formal specification notations can and have been used as the basis for model-based testing. Pre/post notations such as Z,

B, JML and Spec# are widely used for model-based testing, but so are transition-based notations such as Statecharts and UML state machines.

Control of Test Generation: It is necessary to be able to control the generation of tests, to determine how many tests are generated and which areas or behaviours of the system they test.

One approach for controlling the generation is to specify (in addition to the model) some patterns or test specifications, and then generate only tests that satisfy these patterns or specifications.

Another approach is to specify a *model coverage criteria* which determines which tests are interesting. Most of the usual code-based coverage criteria (such as statement coverage, decision/condition coverage, MC/DC, full predicate coverage, def-use coverage) have been adapted to work as model coverage criteria.

On-line or Off-line Test Generation: On-line model-based testing generates tests from the model in parallel with executing them. This makes it easy to handle non-determinism in the system under test, since the test generator can see the outputs from the system under test (after its non-deterministic choice) and change the subsequent test generation accordingly.

On the other hand, off-line test generation generates tests independently of executing those tests. This has numerous practical advantages, such as being able to execute the generated tests repeatedly (for regression testing), in different environments etc.

Requirements Traceability: It is highly desirable for model-based testing tools to produce a requirements traceability matrix, which relates each informal requirement to the corresponding tests. This kind of traceability helps with validation of the informal requirements, and can also be used as a coverage criteria (*every requirement is tested*).

Requirements traceability can be obtained by annotating the model with requirements identifiers, then preserving those annotations throughout the test generation process, in order to produce a relation between requirements and test cases.

3 Similarities and Differences

This section discusses several similarities and several differences between using model-based testing to find errors in a program versus using a program verifier.

3.1 Similiarities

3.1.1 Redundant Specification

To verify the behavioural correctness of a program, we must have a specification of the expected behaviour of the program. Similarly, model-based testing requires a specification (model) of the expected behaviour. In both cases, we have two descriptions of the program behaviour: one is the specification and the other is the executable code.

It is important that these two descriptions should be as independent as possible, since the goal of model-based testing, and of program verification, is to find discrepancies between the two descriptions (or prove that there are no discrepancies). That is, there must be a large degree of *redundancy* between the specification/model and the implementation code. For example, it is usually pointless to derive a specification from the code, then use that specification as the basis for verifying or testing the code—one would expect to find no errors, because there is no redundancy between the two descriptions of behaviour—they are consistent by construction. (The only reason for performing such an exercise would be to try and find errors in the testing/proof tools themselves).

This shows that both model-based testing and program verification have a common overhead of writing the specification. The requirement for redundancy implies that the specification and the code must both be the result of some human creativity, since if one was derived automatically from the other there would be no redundancy.

3.1.2 Abstract Specification

We have seen that model-based testing and program verification both require a specification of the expected behaviour of the system. Another similarity between the two approaches is that this specification needs to be *abstract*. That is, we want the specification to be shorter and simpler than the program itself, typically by omitting many of the details of the program. Otherwise, programmers would not be willing to write such lengthy specifications, the cost of writing them and validating them would be prohibitive, and it would be difficult to have confidence in the correctness of the specification. So abstraction is the key to developing practical specifications, and is a key goal of both model-based testing and program verification.

3.1.3 Reasoning Technology

The goal of most model-based testing tools is to fully automate the test generation process. This requires sophisticated reasoning about specifications and sequences of operations. The reasoning technologies that have been used in

model-based testing include: model-checking, symbolic execution, constraint-based simulation, automated theorem proving, and even interactive theorem proving.

These are the same kinds of reasoning technologies that are needed and used within program verifiers. Indeed, the needs of the two approaches are almost identical. One difference is that, because model-based testing does not try to test all behaviours, it is often acceptable to restrict data types to be small and finite, to make reasoning decidable and fast. However, if the focus of a program verifier is on finding errors (like ESC/Java2 [1]), and completeness is not an essential goal, then the same technique can be used.

3.2 Differences

3.2.1 Colour

Model-based testing is a black-box approach, which can be applied to binary programs without source code, to embedded software, or even to hardware devices. In contrast, program verification is a white-box approach, which requires the source code of the program to be verified, and also requires a formal semantics of that source code. This means that program verification is more restricted in the kinds of systems that it can verify, than model-based testing.

3.2.2 Partiality

Model-based testing is quite often used to test just one aspect of a complex system. This means that the specification can specify just that aspect, rather than having to specify the complete system behaviour. For example, in a GPS navigation system for a car, we might use model-based testing to test the tracking of the vehicle's position, and ignore all route planning, route display and user interaction features. A separate model-based testing project might test the route planning algorithms, while ignoring the other features such as position tracking. Such an approach tests each aspect independently, but does not explore any interactions between the aspects (for example, between route planning and position tracking).

The ability to perform model-based testing from a partial specification means that each partial specification can be more smaller and more abstract than one comprehensive specification would be, which makes it easier to get the specification right and simplifies the test generation task. Furthermore, it seems likely that fewer specifications will be needed for model-based testing than for verification of a program, since a verifier usually requires specifications of all modules within the system, whereas model-based testing requires just a specification of the observable behaviour of the top-level system. One can argue that

[1] See `http://secure.ucd.ie/products/opensource/ESCJava2`.

it is good engineering discipline to require specifications of all modules! However, the point here is simply that the *minimum level* of specification needed for model-based specification is likely to be less than that required for verification, which may help to make model-based testing less costly than verification.

A partial specification may be specifying a subsystem of the system under test, with multiple specifications specifying different subsystems, or it may be specifying a very abstract view of the behaviour, with multiple specifications specifying alternative abstractions of the same system. The former case (verifying subsystems independently) is often used for program verification too, but the latter case (verifying a system with respect to multiple specifications) seems to be rarely used for program verification and may deserve further investigation. It is related to program slicing, which has recently been suggested as an abstraction technique for model-checking programs [VTA04].

3.2.3 Confidence

It is common wisdom that testing can never give a 100% guarantee that a program is bug-free, whereas proof can. This is a fundamental and intrinsic difference between testing and proof.

However, to play devil's advocate, I shall misquote Dijkstra: "Program testing can be used to show the presence of bugs, but [almost] never to show their absence!" [Dij70]. I've added the 'almost' to comment that there are some circumstances, like the following, where testing can 'prove' the absence of bugs.

- Some of the algorithms for generating tests from finite state machines [LY96] can guarantee that if all tests pass, then the system under test has identical behaviour to the specification (the FSM), under the (rather strong!) assumption that the implementation has a maximum number of distinct states that is no greater than the number of states in the specification.
- HOL-Test [BW05] is an interactive model-based testing tool that generates tests from a model, but also generates 'uniformity assumptions' which formalise the assumptions being made about the cases that are not tested and their similarity to the tests. If one could prove these uniformity assumptions, then passing all the tests would imply that the implementation is a refinement of the specification.
- Some algorithms for testing concurrent processes can exhaustively test all interleavings of the processes, which can guarantee that there are no bugs due to concurrency interactions between the processes.
- Cleanroom [Mil93] uses rigorous development techniques (informal proof) to obtain high quality software, and does not use testing as a bug-detection technique. Instead, it uses testing to determine the statistical reliability of the software, by performing random testing based on a profile of the

expected usage of the software. This does not guarantee that the software is bug free, but does tell us its mean time between failures.

Of course, we expect that the proposed program verifier will guarantee absolutely no bugs! Well, except for out-of-memory errors, integer overflow errors etc., which are usually considered outside the scope of verification.

The point is that formal verification is relative to a semantics of the programming language that is usually a simplification of the real semantics. Typically, we ignore some difficult issues such as resource limits. In contrast, testing is the only verification technique that executes the program in its real environment, under conditions that are as close as possible to its intended use. So, as Fig. 1 suggests, verification is good at finding *all* errors down to a certain level of abstraction (usually the simplified language semantics, but the long term goal is to push this abstraction level down as far as possible), whereas testing is good at finding *some* errors at all levels of abstraction (down to and including the hardware).

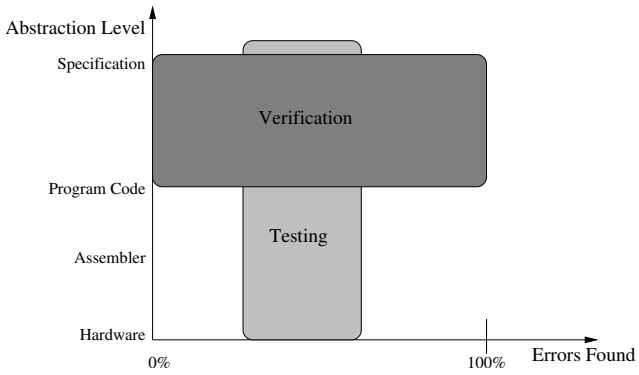

Figure 1: Strengths of Verification and Testing

For these reasons, it is likely that testing and proof will always be somewhat complementary technologies, and the goal will be to find a suitable balance between them.

4 Relationship to the Program Verifier Grand Challenge

Once the grand challenge of a fully automatic program verifier has been achieved, one might argue that there will no longer be any need for model-based testing, or any kind of functional testing, because it will be easy and cheap to *prove* programs correct, rather than test them. And proof is obviously preferably to

testing, since it gives guarantees about *all* behaviours being correct, rather than just detecting some (unknown) proportion of the obvious errors.

However, in the interim, model-based testing obviously has a role to play. Even when the grand challenge has been achieved, model-based testing may still be useful. Its roles could include:

1. Using model-based testing as an introduction to the ideas of formal models and verification. That is, model-based testing is a cost-effective approach to finding errors in non-verified programs.

   The adoption of model-based testing by industry will build experience in formal modelling skills and the use of automated testing/verification tools. These are necessary prerequisites for the use of a full program verifier. Thus, model-based testing can be viewed as an evolutionary step (the missing link!) from the status quo, towards fully automatic program verification by proof.

   Model-based testing changes the current software lifecycle in a small way (it introduces formal modelling, and modifies the test development processes, but leaves the rest of the lifecycle unchanged), whereas full verification seems likely to require more significant methodological changes.

2. Validating the specification notations which will be used by the program verifier. Similar notations are needed for model-based testing and for full verification. So model-based testing may be a useful way to validate the expressive power and usability of proposed specification notations.

3. Using model-based testing as a specification-validation tool. Experience with model-based testing shows that the faults exposed by executing the generated tests are often due to specification or requirements errors, rather than implementation errors [BBN04]. So if we have a program and a proposed specification for that program, model-based testing could be used to detect errors in the specification, before starting verification.

4. Using model-based testing as an approximation of the program verifier. That is, after an engineer has written a specification of the desired system, and also written an implementation of that system (or done a large refinement step towards an implementation), model-based testing could be used to automatically find some of the errors in that implementation/refinement. This is similar usage as the previous role, but aimed at finding errors in the implementation rather than the specification.

   Model-based testing may give faster and more comprehensible detection of errors than a proof approach. After all errors detected via model-based testing have been corrected, then a proof approach could be started. This is based on the oft-quoted observation that 90% of proofs fail because the conjecture that is being proved is false.

5. Using model-based testing as an alternative to full verification. If we assume that the program verifier requires a significant amount of input to achieve good results (for example, very precise specifications of the system and each module within the system), then the cost of full verification may not be cost-effective for some non-critical systems. In this case, model-based testing may be a viable alternative, albeit with reduced guarantees about the program correctness. Model-based testing is likely to require fewer specifications than full verification (eg. only a top-level system model, rather than a model of each module within the implementation, and a partial model is often sufficient to generate useful test suites).

References

[BBN04] M. Blackburn, R. Busser, and A. Nauman. Why model-based test automation is different and what you should know to get started. In *International Conference on Practical Software Quality and Testing*, 2004. See `http://www.psqtconference.com/2004east/program.php`.

[BFS05] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for Test Case Generation. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems [BJK+05]*, Springer LNCS 3472, pages 391–438. Springer-Verlag, 2005.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*. Number 3472 in LNCS. Springer-Verlag, 2005.

[BW05] A. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in LNCS, pages 16–32. Springer-Verlag, Linz, 2005.

[Dij70] Edsger W. Dijkstra. On the reliability of mechanisms. In *Notes On Structured Programming*. EWD249, 1970. See `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`.

[EFW02] I.K. El-Far and J.A. Whittaker. Model-based software testing. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 825–837. Wiley-InterScience, 2002.

[LPU04] B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. *The Journal of Software Testing, Verification and Reliability*, 14(2):81–103, 2004.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — A survey. *Proceedings of the IEEE*, 84(2):1090–1126, 1996.

[Mil93] Harlan D. Mills. Zero defect software: Cleanroom engineering. *Advances in Computers*, 36:1–41, 1993.

[VTA04] Vivekananda M. Vedula, Whitney J. Townsend, and Jacob A. Abraham. Program slicing for atpg-based property checking. In *17th International Conference on VLSI Design, Mumbai, India, January 5-9 2004*, pages 591–596. IEEE Computer Society Press, 2004.