Model-Based Testing

Stuart Reid

Cranfield University
Royal Military College of Science,
Swindon, Wilts SN6 8LA, UK

s.c.reid@cranfield.ac.uk

Abstract

This paper provides an introduction to the topic of model-based testing. The model-base testing process is described, and the choices available at each stage considered. The different types of tool necessary to support the process are explained and example tools listed along with the notations they support. The place for standards in model-based testing is examined, and the new skills needed by testers discussed. Finally, means of determining the suitability of projects for model-based testing are considered before a number of case studies (by both tool developers and users) are described.

Introduction

We all use models to perform testing – otherwise we wouldn't have a clue whether a test passes or fails – these models allow us to know how the software _should_ behave in a given situation. Most people's models are, however, very personal and never see the light of day, only existing for a brief time in the tester's head.

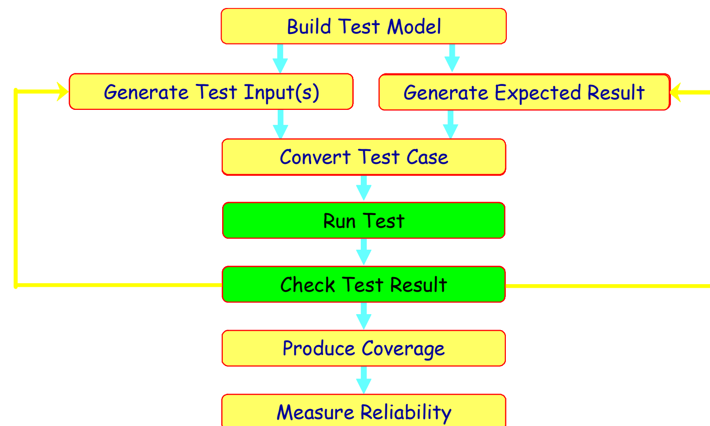

Figure 1: The Model-Based Testing Process

With model-based (or model-driven) testing, the model of the system's behaviour is made explicit and (in an ideal world) used as the basis for the complete automation of the testing (see figure 1). The benefits of this approach are obvious. If we can both automatically generate and run complete test cases (including expected results) then

the number of tests we run is limited only by the availability of a test environment and its processing power. A longer-term benefit is that once a model is built and it is possible to automatically generate test cases from it, then test maintenance becomes simply a matter of 'tweaking' the test model to reflect changes to the system and then letting the automation do the work.

Background

We are getting more and more reliant on software-intensive systems, which, in turn, are getting more and more complex. As the demand for new systems increases, there are regular advances in development technology to try and satisfy this demand. Typical of these advances are the growing capability in the area of automatic code generation and the predicted increases in reuse provided by using the model-driven architecture [14]. With developers building greater numbers of more complex systems comes the need for a corresponding increase in testing. Although testing tool manufacturers would have us believe that the solution is to automate more and more of the test process, practical experience indicates that so far this form of test automation has not been the promised 'silver bullet', especially when the difficulties of maintaining test scripts are taken into account.

It has been suggested that if software testing is to keep apace with development then a leap forward in testing technology is needed, which is where model-based testing comes in. Model-based testing is not simply improving or automating current steps in the test process, but is rather a step-change in technology, requiring major changes in the test process, tools and those staff carrying it out. For instance, testing is moved far earlier in the life cycle, new modelling skills are required by the testers, and test case generation tools are an absolute necessity.

The Test Models

A wide range of model types can be used in model-based testing. The information represented by the test models is used as the basis for the test case generation, so the models obviously need to include the behaviour that the tester wishes to test. For instance, if the model used provides details of the required functionality then test cases exercising this functionality will be produced. In fact, although functionality is the traditional favourite of software developers, models of state behaviour have, so far, been the most popular with model-based testers, due mainly to interest in it from parts of the telecoms industry whose systems perform switching, which is largely state-based. It should be noted that the attributes required of the test model will be slightly different from those required by the developers, especially if the test model includes information to create negative test cases (e.g. scenarios outlining the possible loss of system resources) as the developers tend to restrict their model to the more positive aspects of user behaviour (e.g. functionality the user wants the system to have). Also, while the developers' model must cover the complete set of requirements, the test model does not have to fulfil this criterion. The test model can concentrate on critical areas of the application in enough detail for it to be used as an oracle, which is generally in far more detail than the developers' model.

If the model used is based on a single abstraction, such as functionality or state-based behaviour, it must be remembered that even if model-based testing is subsequently fully performed then only those behaviours represented in the model will have been exercised, and that if other forms of behaviour are also important to the users then other forms of testing will also have to be used. In theory the models used can represent any forms of abstraction, but the modelling of some non-functional requirements, such as usability, is currently very poorly understood, and it may well be some time before model-based testing is able to support this area of testing. In contrast, reliability is considered quite suitable for model-based testing as it is simply based on probability of failure, although any reliability measurement will still be restricted by the abstraction represented by the models used. So, if the model used is limited to state behaviour the reliability measured will correspond to the reliability of the state behaviour of the system, not its overall reliability as failures due to, for instance, data processing faults will not have been considered.

Notations

As stated earlier, so far, many of the models used in model-based testing have been based on state behaviour, which can be represented in a variety of standard notations, such as UML State Diagrams, Message Sequence Charts and Specification and Description Language (SDL). When using state models, it can be useful to supplement the models with extra information about the actions attached to the transitions, which would typically describe data processing behaviour.

Another popular notation used is the requirements table, often made up of a number of input conditions related to a set of outcomes, providing similar information to that of a cause-effect graph. Such notations lend themselves to the generation of test cases that systematically cover logically-derived subsets of typically very large input combinations.

Inevitably most of the models are based on those originally used by developers, but extra information required by the testers often means that hybrid notations, specific to model-based testing are created. For instance, Markov models can be created relatively simply by annotating the transitions on UML State Diagrams with probabilities.

Model Independence

When creating the test model, the degree of independence of this model from that being used by the developers must be decided. At one extreme the test model can be created in complete isolation from the developers, providing the benefit of being able to compare the two models for any differences in their interpretation of the requirements. If this high level of independence is employed then the greatest benefit is achieved by creating the test model at the same time as the development model. This then allows any faults discovered in the development model to be rectified before developers' time is wasted on implementation of faulty specifications.

At the other extreme, the developers' and testers could use a single (presumably detailed) model to drive both development and testing. However, this somewhat

nullifies the benefits of subsequent testing, especially if automatic code generation from the model is used by the developers. More likely is a hybrid approach (see figure 2), where the developer's model is enhanced to add extra information to allow both the generation of inputs and expected results (or another way of checking results). As with the independent approach, by timing the enhancement of the developers' model carefully (to immediately follow its completion, and hopefully precede its use in the next stage of development) then added value from the detailed review of the developers' model performed as part of the enhancement activity can be gained.

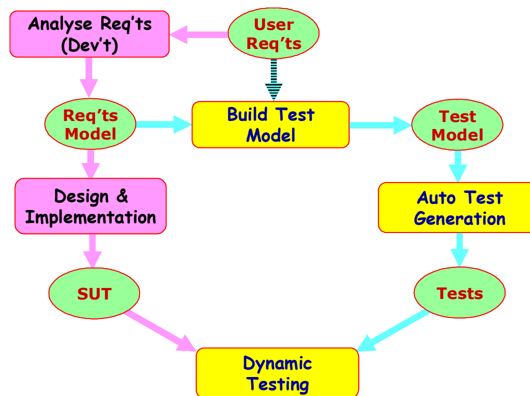

Figure 2: The Hybrid Model Approach

Test Case Generation

Having created a test model, we need strategies and tools to create test cases from this model. The problem of test case generation can be considered in two parts; first, test inputs must be generated and then, second, a means of determining whether the test has passed or failed must be created.

Test Input Generation

When considering the test input generation problem, and given a graphical representation in the test model, it is initially tempting to proceed without further thought and to follow Boris Beizer's advice on what to do when you see a graph and "COVER IT!". However, experience shows that the test models of most real systems are complex enough to require a more considered approach to choosing the coverage that should be attempted.

Test input generation tends to be based on achieving a level of coverage of some particular aspect of the model. The coverage is thus normally closely related to the notation used and the abstraction it supports. The user normally communicates their coverage requirement to the test generation tool (assuming they have a choice) as a 'test objective'. For instance, if the model is state-based, then a simple test objective might be to exercise each of the transitions, or perhaps to visit each of the states. Similarly, a functional model based on a requirements table might suggest a test objective of exercising combinations of inputs set to particular values. There are few agreed standards of model coverage and it appears that the wide diversity of model

notations has led to a corresponding diversity in coverage measures. This is problematic as it can be difficult to compare the results achieved by different model-based testing approaches, which appear to be using similar, but very slightly different, test coverage measures. This situation is not helped by many of the developers of test generation tools, who are unwilling to divulge the details of their test generation algorithms for fear of losing potential competitive advantage.

A typical problem encountered when using state models is known as 'state space explosion', where models with a phenomenally large number of states are created due to the natural complexity of many systems. For instance, on the AGEDIS project, one SDL model of a real system had four main processes each with relatively simple behaviour, but because of the high degree of concurrency, the combined model had more than 500,000 states and 800,000 transitions making it impossible for the AGEDIS toolset to create a test suite [8]. This problem has led to the creation of some complex test coverage criteria and corresponding test generation algorithms, often making it very difficult to comprehend the test coverage achieved.

If a test objective is chosen that is not fully achieved then the resultant testing is of dubious value. Without complete coverage, testing may have been concentrated in just one area and important system functions may not have been exercised at all. It is often possible to specify test objectives that will cause particular aspects of the model to be exercised. In a risk-based testing environment such an approach could be used to ensure all high risk areas of the model are exercised, probably alongside a more general model coverage objective. Where a Markov model has been produced, the probabilities of transition are often used to aid test generation where the test objective is to generate tests that mimic the expected use of the system.

A top-down strategy to test generation can be followed by initially using very high level and therefore simple models (with these it should be easier to keep test generation under control). Use of high level models means that high level faults should be found first, and as the models are refined and more details added then more 'low level' faults will be found.

The Oracle

Probably the most difficult part of the model-based testing approach for people to accept is the concept of an oracle being available as part of the test model. The oracle provides the ability to automatically determine whether tests have passed or failed. Many people initially argue that if an automated oracle was available, then it could have been used as the delivered system, as presumably it could do much the same job. But this often exaggerates the ability of the oracle. If an oracle was available that did everything required of the system then no new system would have been built. The oracle's ability to check whether tests have passed or failed is subtly different from being able to generate the actual expected results (although some do work this way).

The inclusion of the oracle function in the test model is normally the major reason that the test model differs from the developers' model. Much of the extra detail provided in the test model is there to provide the oracle's ability to check whether a test has passed or failed. For state-based models, typical extra information that may

be added for this purpose describes the actions associated with state transitions. The results of these actions generate expected outputs, which are then checked against actual outputs. However, for some state models it is often considered enough of a check that the system followed the correct sequence of states, and the oracle may either monitor the system state using a form of instrumentation where probes are inserted into the system under test or simply assume that if the inputs are accepted in the correct order then the system is operating correctly.

The oracle can be a ‘degraded’ version of the system under test. For instance, it may be a legacy system that is being replaced because it runs too slowly, and it will be too expensive to upgrade its performance. The fact that the oracle runs slower than the ‘real’ system can be of little importance when running tests. Similarly, in projects where simulations are built to define required functionality (or test ideas), it is often possible to run these simulations and use them as oracles. A further degraded form of oracle is where the oracle does not actually produce expected results, but instead perform checks on the actual output to determine that it is ‘within acceptable limits’; this is often known as a plausibility check.

The form of the oracle used will depend on a number of factors that will include the detail available in the model, the ease with which the model can be executed to generate expected results, the risk associated with the system under test, the availability of executable models, the availability of legacy or functionally-similar competitive systems and the ease of instrumenting and monitoring the system under test, among others.

Test Execution

The actual execution of the test cases should be little different for model-based testing than it is for traditional testing approaches using automated testing. As model-based testing is typically performed at the system test level, then a test environment as close to the real world as possible is normally required. If state monitoring is to be performed as part of the testing, then the extra software required to perform this and record the results will have to be included with the system under test.

Given the potentially enormous size of the test suite that can be automatically generated with model-based testing, it is often the case that the complete test suite is not generated, stored and then executed in this sequence due to its prohibitive size. Rather the test cases are executed as they are generated, useful results recorded (often only failure data), and then the next test case generated, and so on. Obviously any test harness run in this, or a similar, manner must be able to handle crashes and automatically reset the test environment to a state suitable for running subsequent tests. In some cases the complex environment of the system under test and the need for intrusive monitoring to support test coverage measurement can mean that the cost of setting up the test harness can be very high.

Tools and Standards

The tools used to generate the models used in model-based testing are normally very similar to those used by developers, and may be identical in some cases. Where

hybrid notations are used then open source development tools can be relatively easy to modify, although in the future it is expected commercial development tool vendors will provide the necessary added functionality for testers.

The tools for test case generation are specific to the model notation used and the test coverage criteria to be achieved. These tools are not currently used in traditional software testing or development and so are specific to model-based testing. Because of their dependence on the input model, they are generally closely-tied to the tools used for model generation.

Test execution is performed in the same way for model-based testing as it is traditional testing, so the same tools can normally be used.

A list of model-based testing tools and the modelling notation they support is shown in figure 3.

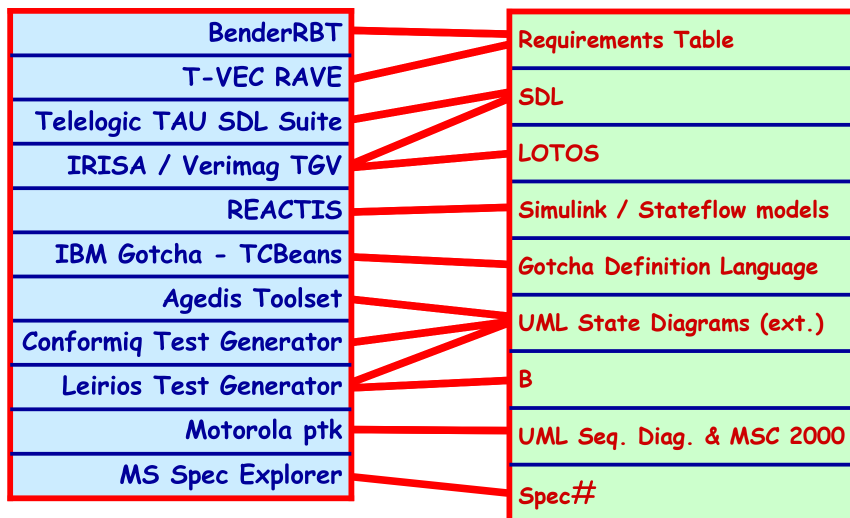

Figure 3: Model-based testing tools and notations

At present there are no standards that specifically support model-based testing; however there are several standards that can be used from other areas of software engineering. For instance, the most likely candidate for the test model is the new UML 2.0 standard from the OMG [12]. Although in its basic form this standard is probably not precise enough for test modelling, it should be possible to enhance this standard's usefulness with a test modelling profile. The output from the test generation tool is a test suite (or a sequence of test cases). The two most obvious candidates for the standard defining test cases are TTCN-3 [11] and a XML-based standard, such as that used in the AGEDIS project [9]. If in the future model-based testing tools communicate using standard data formats, then users will be able to choose between combinations of tools for each of the three main stages rather than being tied to a single tool and supplier, with all the inherent economic and technological limitations that this brings.

The algorithms used for test generation and their corresponding coverage criteria are related by the same coverage measurement. There are some test coverage measures defined in BS 7925-2 [10], and although these were originally defined to support component testing, some of the measures, such as state transition coverage, may be

appropriate for use in model-based testing. There appears to be lack of in-depth knowledge in this area of the model-based testing approach, although Offutt and Abdurazik have suggested a set of coverage criteria based on UML state charts [1]. Some basic coverage measures are widely-understood, but the more complex measures required for sensible coverage of large state models appear to be either proprietary, or at least not well-understood by the testing community. For model-based testing to become accepted on a wider scale the test coverage achieved by the approach must be both widely understood and accepted thus allowing model-based testing to be more easily compared with traditional testing and the best way for this to happen is for these measures to be standardised.

At some point in the near future there will also come a requirement for a standardised model-based testing methodology, to provide a common understanding of the approach.

Tester Skills

The skills required to perform model-based testing are quite different from traditional testing. Gone is the need for test case design and maintenance skills, but this is replaced by the need for skills in the areas of modelling and the setting of test objectives. It seems likely that of the two traditional groups from which testers are drawn, developers are going to find it easier to relate to model-based testing skills than users, whose main experience with models will only be as a reader rather than a writer. Overall the skills are probably more advanced than those currently required for traditional testing, so while re-training of testers is possible, care will need to be taken who is chosen for this.

Suitable Applications

Before anybody moves to using model-based testing, they must be sure that the approach is appropriate for their situation.

The creation of test models is obviously a large up-front cost, but this *should* be recouped by the lower maintenance costs when the system is operational (see figure 4). Of course, if the system is expected to have low maintenance costs anyway, then the test modelling costs will not be recouped by the potential maintenance savings of model-based testing until much later in the maintenance phase. Low maintenance costs might be predicted, for instance, if the system is planned to have only a short operational life or it is expected that there will be particularly few changes to the system required by the users (and its environment).

Obviously the application must be suitable for modelling in a supported notation. As stated earlier, switching applications have been found to be particularly appropriate, as they are well-suited to modelling as a state model and there is good tool support for this. The application must also be considered important enough to warrant the cost of model-based testing. If high quality is not important to the customer then model-based testing will be unlikely to be cost effective.

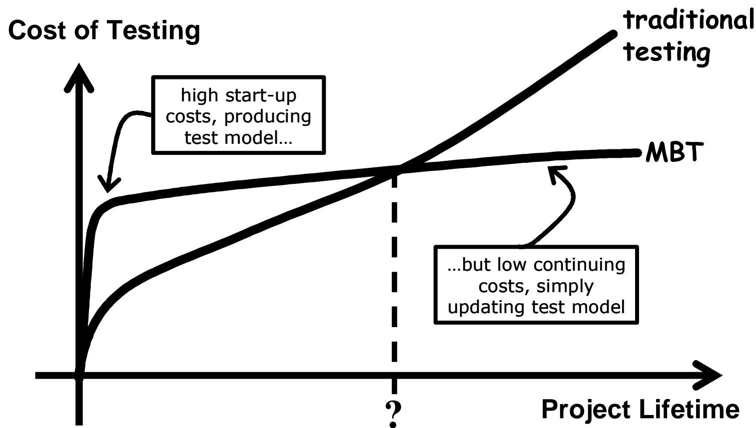

Figure 4: Model-based testing costs

The difficulty of performing traditional testing of concurrent applications could also act as a driver towards using model-based testing, where the mix of many test cases and systematic test case generation provides good test coverage of complex models.

Case Studies

There have been a number of case studies of model-based testing, some performed by tool developers and others by end users. Unsurprisingly those performed by the tool developers are near all very positive. Those performed by users are far more mixed (and may be considered by some to be more useful).

Developer Case Studies

The Test Automation Framework approach has been documented in several papers [2], [3], [4], etc. It uses the SCRtool for modelling and the T-VEC tool for automatic test generation. In one study it was shown to reduce the time and effort required to perform security testing while increasing test coverage of two applications based on Oracle8 [2]. In another study an experiment was performed on the failed Mars Polar Lander software, and, after one false start, the approach was able to identify the probable cause of the loss of the mission [3].

Teradyne present the results of two case studies performed on a telecoms 'call waiting' application, and a work order management and tracking system [5]. For the call waiting application, model-based testing was completed in just 12% of the time required for traditional testing of the same application. Model-based testing was shown to require an additional 10 days' work to create the model for the work order system compared to the traditional approach, but subsequent changes to the system only needed 15 minutes work using model-based testing as opposed to 5 days work using the traditional approach. Also, tests were generated at the rate of one every three seconds using model-based testing, while traditional testing took 20 minutes to generate a single test. Overall model-based testing was considered to have been more efficient, and to have increased the final product quality.

Results from using their AETG software system on four projects were presented by Bellcore [6]. They concluded that a surprisingly small amount of training (about two

hours) was required by testers to be able to use their system, and that numerous defects not exposed by traditional testing had been detected. Their experiments, however, were not 'full' model-based testing as they did not include the use of an oracle and so the check whether tests passed or failed had to be performed manually.

User Case Studies

A case study on the use of model-based testing was performed at Hewlett Packard, looking at printer driver installer testing [7]. An extended finite state machine was used as the modelling notation and the TestMaster tool used to generate test cases. The conclusion from Hewlett Packard was that the model-based testing was successful and test maintenance would be cheaper with this approach. However, the need for a set of unique test skills and the poor integration of the model-based testing tools with their test management tools meant the results were not good enough to persuade Hewlett Packard to take up model-based testing.

The AGEDIS project, funded by the European Commission, saw the development of a complete set of model-based testing tools that were applied by three industrial partners, IBM UK, Intrasoft in Greece and France Telecom [8], [9]. They found that the modelling was well worthwhile as faults were detected early in the life cycle, but also that the skills required to perform the test modelling were difficult to master and that the high cost of generating the model meant it was not worthwhile for small systems. Another problem was that the complexity of the test model for larger systems made automatic test generation of a complete test suite difficult if not impossible given the test generation algorithms they were using.

A joint presentation by Luottokunta and Conformiq bridged the gap between user and developer presentations [13]. The application studied was a switching system for clearing credit card payments. This case study found that tests were generated approximately 600 times faster using the Conformiq Test Generator tool, while they "guesstimated" that there were no additional costs for modelling and management than those required for traditional testing. At the time of the presentation, with most of the system operational, no failures in production had been found. This project appears to be an excellent example of where carefully checking the suitability of an application for model-based testing (Conformiq performed a pre-project proof of concept study) can lead to a very successful outcome.

Conclusion

Model-based testing has already been used successfully on a number of projects and the number of applications for which model-based testing will be suitable will continue to grow. A number of factors will drive this growth. First, more and more developers will use modelling notations such as UML for their development, so providing the basis of the test models needed for model-based testing. Second, more commercial quality tool support will become available. Third and most importantly, is the economic argument. As users come to expect systems of higher quality and project managers recognise the necessity of reducing test maintenance costs, then model-based testing will become increasingly difficult to ignore.

Model-based testing will not, however, be appropriate for all situations. There will still be projects where no explicit model of requirements is available and testing needs to be finished before the end of next week. There will also be situations where the available testers are not professional software engineers, but simply those re-assigned from other parts of the organisation who are currently free.

Model-based testing requires commitment from both management and the testers. Project and test managers must recognise when the up-front costs of model-based testing will mean lower overall costs when test maintenance is considered. Testers must recognise that if they are to be treated on a par with developers they must have the necessary skills, both to build and review models, and to drive the necessary automatic test generation.

References

[1] Offutt, J. & Abdurazik, A., *Generating Tests from UML Specifications*, Second International Conference on the Unified Modeling Language (UML99), Colorado USA, Oct 1999.

[2] Blackburn, M. et al, *Model-based Approach to Security Test Automation*, ISSRE 2002, 13th International Symposium on Software Reliability Engineering, Maryland, USA, Nov 2002.

[3] Blackburn, M. and Knickerbocker, R., *Mars Polar Lander Fault Identification Using Model-based Testing,* Eighth International Conference on Engineering of Complex Computer Systems, Maryland, USA, Dec 2002.

[4] Blackburn, M. et al, *Defect Identification with Model-Based Test Automation*, Society of Automotive Engineers; SAE 2003, Detroit USA, March 2003.

[5] Apfelbaum et al, *Model Based Testing*, Software Quality Week, May 1997.

[6] Dalal, S. R. et al, *Model-Based Testing in Practice*, Proc. of ICSE'99, ACM Press, May 1999.

[7] Struble, S., *Model-Based Testing of Installers in a Development Test Environment*, from http://www.geocities.com/model_based_testing/online_papers.htm.

[8] Craggs et al, *AGEDIS Case Studies: Model-Based Testing in Industry*, Proc. 1st European Conference on Model Driven Software Engineering, Nuremberg, Germany, Dec 2003.

[9] Crichton et al, *Using UML for Automatic Test Generation*, 16th IEEE International Conference on Automated Software Engineering (ASE 2001), San Diego, USA, IEEE Computer Society, Nov 2001.

[10] BS 7925-2-1998, Software Component Testing.

[11] TTCN-3. See http://www.etsi.org/ptcc/ptccttcn3.htm.

[12] UML 2.0. See http://www.uml.org/.

[13] Laine, P. and Laine, A., *Large scale use of model based testing*, ICSTEST UK, London, Oct 2004.

[14] Model Driven Architecture (MDA). See http://www.omg.org/mda/.