Model-Based Testing – Next Generation Functional Software Testing

MBT RENEWS THE WHOLE PROCESS OF FUNCTIONAL SOFTWARE TESTING: FROM BUSINESS REQUIREMENTS TO THE TEST REPOSITORY, WITH MANUAL OR AUTOMATED TEST EXECUTION.

by Bruno Legeard and Mark Utting

Model-Based Testing (MBT) is an increasingly widely-used technique for automating the generation and execution of tests. There are several reasons for the growing interest in MBT:

- The MBT approach and the associated commercial and open source tools are now mature enough to be applied in many application areas, and empirical evidence is showing that it can give a good Return On Investment (ROI);
- The complexity of software applications continues to increase, and the user's aversion to software defects is greater than ever, so our functional testing has to become more and more effective at detecting bugs;
- The cost and time of testing is already a major portion of many projects (sometimes exceeding the costs of development), so there is a strong push to investigate methods like MBT that can decrease the overall cost of test by designing tests automatically, as well as executing them automatically.

MBT renews the whole process of functional software testing: from business requirements to the test repository, with manual or automated test execution. It supports the phases of designing and generating tests, documenting the test repository, producing and maintaining the bi-directional traceability matrix between tests and requirements, and accelerating test automation.

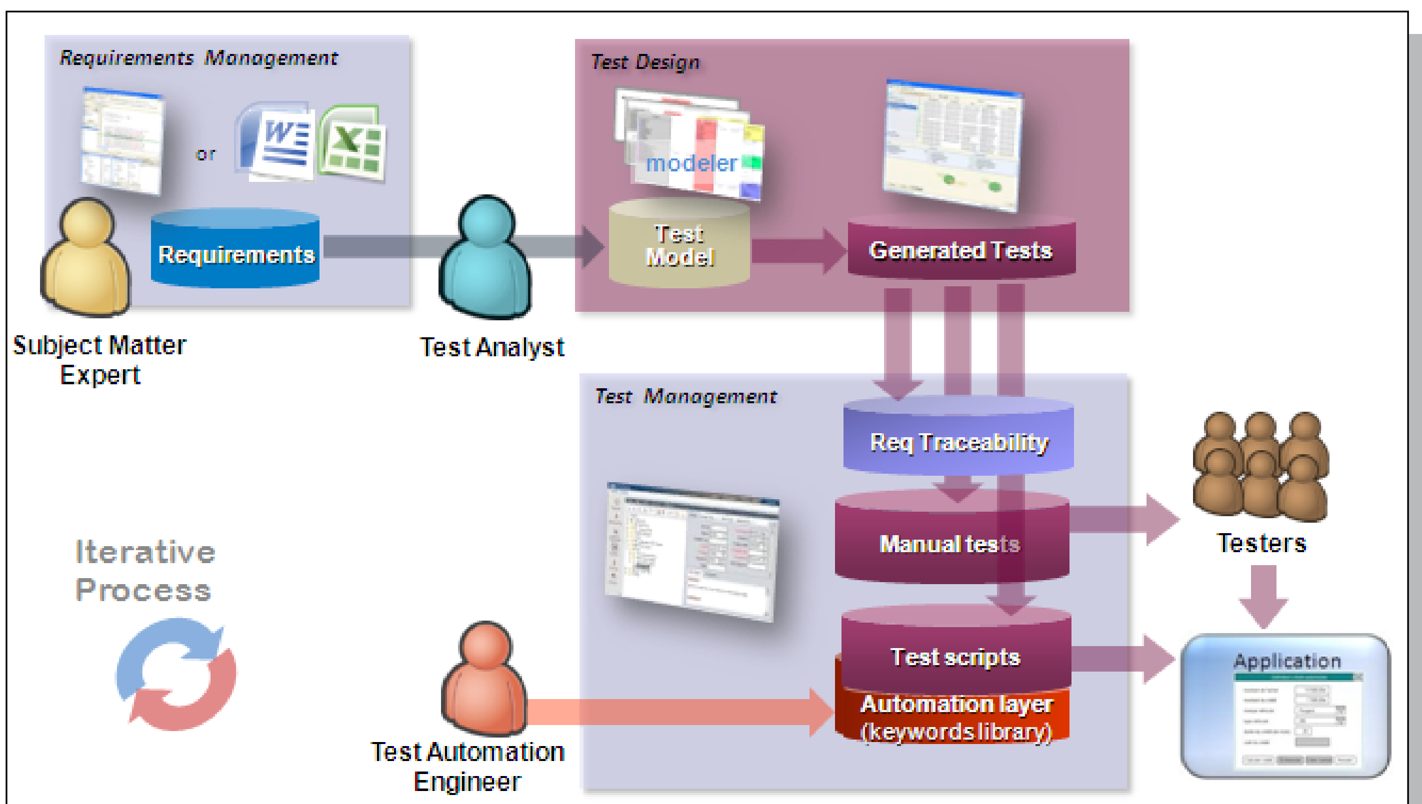

Figure 1: The MBT Process.

This paper addresses these points by giving a realistic overview of model-based testing and its expected benefits. We discuss **what** model-based testing is, **how** you have to organize your process and your team to use MBT, and give some examples of MBT approaches and tools.

What is MBT?

Model-based testing refers to the processes and techniques for the automatic derivation of abstract test cases from abstract formal models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases.

Therefore, the key points of model-based testing are the modeling principles for test generation, the test generation strategies and techniques, and the concretization of abstract tests into executable tests. A typical deployment of MBT in industry goes through four stages (Figure 1).

1. **Design a Test Model.** The model, generally called the test model, represents the expected behavior of the System Under Test (SUT). Standard modeling languages, such as Unified Modeling Language (UML) or Simulink, are used to formalize the control points and observation points of the system, the expected dynamic behavior of the system, the business entities associated with the test, and some data for the initial test configuration. Model elements such as transitions or decisions are linked to the requirements, to ensure bi-directional traceability between the requirements and the model, and later to the generated test cases. Models must be precise and complete enough to allow automated derivation of tests from them.

2. **Select some Test Generation Criteria.** Usually an infinite number of possible tests could be generated from a model. The test analyst chooses Test Generation Criteria to select the highest priority tests or to ensure good coverage of the system behaviors. One common kind of test generation criteria is based on structural model coverage, using well known test design strategies such as equivalence partitioning, cause-effect testing, pair-wise testing, process cycle coverage, or boundary value analysis[1]. Another useful kind of test generation criteria ensures that the generated test cases cover all the requirements, perhaps with more tests for requirements that have a higher level of risk. In this way, model-based testing can be used to implement a requirement and risk-based testing approach. For example, for a non-critical application, the test analyst may choose to generate just one test for each of the nominal behaviors in the model and each of the main error cases; but for one of the more critical requirements, they could apply more demanding coverage criteria such as all loop-free paths, to ensure that the businesses processes associated with that part of the test model are thoroughly tested.

3. **Generate the tests.** This is an automated process that generates the required number of high-level (abstract) test cases from the test model. Each generated abstract test case is typically a sequence of high-level SUT actions, with input parameters and expected output values for each action. These generated test sequences are similar to the high-level test sequences that are designed manually in keyword-based or action-word testing [2]. They are easily understood by humans, but too high-level to be directly executed on the SUT. A further concretization phase automatically translates each abstract test case into a concrete (executable) test [3], using user-defined mappings from abstract data values to concrete SUT values, and mappings from abstract operations into SUT Graphical User Interface (GUI) actions or Application Programming Interface (API) calls. For example, if the test execution is via the GUI of the SUT, then the action words are linked to the graphical object map, using a test robot, such as HP QuickTest Pro, IBM Rational Functional Tester or Selenium. If the test execution of the SUT is API-based, then each action word must be implemented on this API via some glue code. The expected results of each abstract test case are translated into oracle code that will check the SUT outputs and decide on a test pass/fail verdict. Tests generated from the test model may be structured into multiple test suites and published into standard test management tools, such as HP Test Director, IBM Rational Quality Manager, or the open-source TestLink tool. Maintenance of the test repository is done by updating the test model, then automatically regenerating the test suites;

4. **Execute the Tests.** Generated concrete tests are typically executed within a standard automated test execution environment, such as HP QuickTest Pro or IBM Rational Functional Tester. Alternatively, it is possible to execute tests manually – i.e. a tester runs each generated test on the SUT, records the test execution results, and compares them against the generated expected outputs. Either way, when the tests are executed on the SUT, we find that some tests pass and some tests fail. The failing tests indicate a discrepancy between the SUT and model, which needs to be investigated to decide whether the failure is caused by a bug in the SUT, or by an error in the model or the requirements. Empirically, MBT finds SUT errors, but is also highly effective at exposing requirements errors [4].

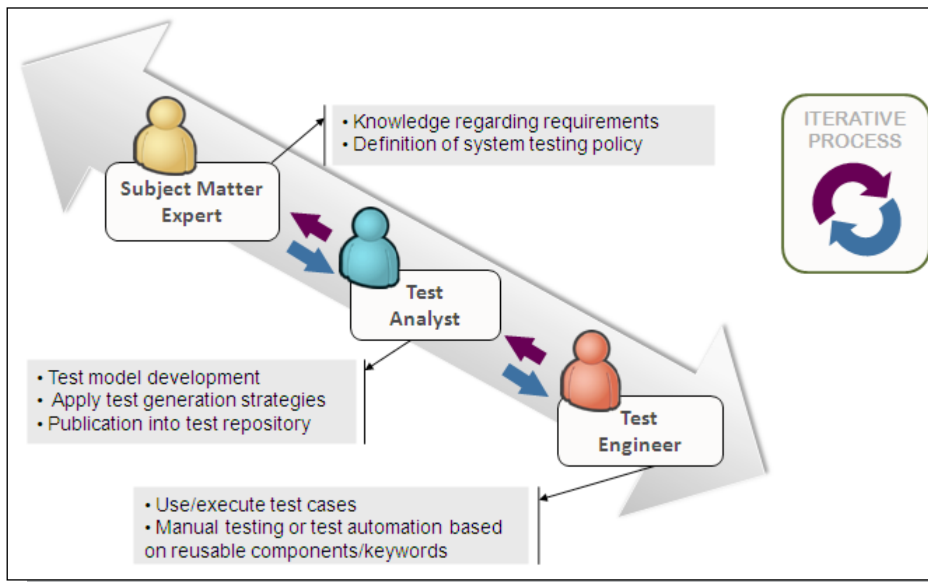

Figure 2: Main Roles in the MBT.

Roles in the MBT process

The MBT process involves three main kinds of people (Figure 2 on the following page):

1. The **Subject Matter Expert** is the reference person for the SUT requirements and business needs. He/she dialogues with the test analyst to clarify the specifications and testing needs.
2. The **Test Analyst** interacts with the customers and subject matter experts regarding the requirements to be covered, and then develops the test model. He/she then uses the automated test generation tool to generate tests and produce a repository of test suites that will satisfy the project test objectives.
3. The **Test Engineer** is responsible for connecting the generated tests to the system under test so that the tests can be executed automatically. The input for the test engineer is the test repository generated automatically by the Test Analyst from the test model.

The test analyst is responsible for the quality of the test repository in terms of coverage of the requirements and fault detection capability, so the quality of his/her interaction with the subject matter expert is crucial. In the other direction, the test analyst interacts with the test automation engineer to facilitate test automation (implementation of key-words). This interaction process is highly iterative.

The Scope of Model-Based Testing

MBT is mainly used for functional black-box testing. This is a kind of back-to-back testing approach, where the SUT is tested against the test model. Differences in behavior are reported as test failures. The model formalizes the functional requirements, representing the expected behavior at a given level of abstraction.

Models can also be used for encoding non-functional requirements such as performance or ergonomics, but this is currently a subject of research in the MBT area rather than common practice. Security requirements can often be tested using standard MBT techniques for functional behavior.

Regarding the testing level, the current mainstream focus of MBT practice is system testing and acceptance testing, rather than unit or module testing. Integration testing is considered at the level of integration of subsystems. In the case of a large system of systems, MBT may address test generation of detailed test suites for each sub-system, and manage end-to-end testing for the whole system.

An Example of MBT

In this section, we present a MBT approach based on a representative commercial tool: ***Microsoft Spec Explorer.*** We illustrate the main characteristics of the approach by giving an example of modeling an application and generating tests from the model.

Microsoft Spec Explorer [5] is a new Microsoft tool that is planned to be released with Visual Studio 2010. It has been used internally within Microsoft for several years, by several hundred testers. It is now becoming a product for external use. To illustrate how it works, we model a simple container object that can have strings added or removed from it. For extreme simplicity, we focus on testing the container with just a single string.

The first modeling step is to decide which actions of the SUT (the container API) we want to model. In Spec Explorer, this is documented in a configuration file in a C-like notation called Cord (Figure 3 on page 12). We model just two actions: *Add1* corresponds to adding the string to the container, and *Remove1* corresponds to removing the string from the container.

To model the behavior of these two actions, we write a

```
config Main
{
// The two implementation actions that will be modeled and tested
action abstract static void StringSetModel.Add1();
action abstract static bool StringSetModel.Remove1();
...
}
```

Figure 3: Configuration file (Config.cord) that defines the actions we want to model and test.

small C# class that defines each of the actions as a method (Figure 4). The class also has a private Boolean flag called *str1*, to remember whether the string is currently in the container or not. The actions simply set or clear this flag. The *Remove1* action also returns a Boolean result that tells the caller whether the string was actually removed from the collection, or whether it was already missing from the collection. Since this model is written in a familiar language (C#), it is not difficult for

```
// This C# program models a set that contains a single string.
public static class StringSetModel
{
  private static bool str1;

  [Action("Add1")]
  static void Add1()
  {
    str1 = true; // str1 is now in the set.
  }

  [Action("Remove1/result")]
  static bool Remove1()
  {
    bool wasPresent = str1;
    str1 = false; // str1 is now out of the set.
    return wasPresent;
  }
}
```

Figure 4: Model.cs, which defines a simple model of adding and removing one string.

someone with a programming background to develop quite sophisticated test models.

After designing this test model, which describes the expected behavior of the collection SUT, we can explore the model visually, to see if we've got the model right. Spec Explorer includes an 'Exploration Manager' tool that can explore all the possible actions and internal states of a model, and when we run this on our model we get the diagram shown in Figure 5.

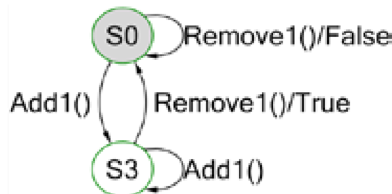

Figure 5: A finite state diagram of all the states and transitions of the model.

This looks good - the grey S0 state is the initial state where the string is not in the collection, which is why the *Remove1* action returns *False.* Spec Explorer has figured out that removing the string from the S0 state has no effect (leaves the state unchanged), and that adding the string twice is the same as adding it once (we stay in state S3), etc. For larger models, the Cord language has several ways of limiting exploration to a subset of the model, so that it is possible to visualize one aspect of its behavior at a time.

The most interesting and important part of using Spec Explorer is generating tests from the model. There are a variety of test generation strategies possible, such as generating one long test that will test all the states and transitions of the model, generating multiple shorter tests, or defining specific scenarios that focus the testing on just one aspect of the model.

Figure 6 (on page 14) shows a *'machine'* declaration in the Config.cord file that says we want to generate several short tests. The left side of Figure 6 gives a graphical view of the tests generated from that machine. For example, the longer test sequence checks that *Remove1* returns *False* on the empty container, then adds the string and checks that removing the string returns true, but removing it a second time returns false (because it has already been removed). As well as this graphical view of the generated tests, Spec Explorer also generates a test suite that implements the same tests as C# code. This can be connected to a real implementation of the SUT, and then executed by Visual Studio to test that SUT and report any failures.

So with Spec Explorer, the human tester designs a high-level C# class that models the expected behavior of the SUT, and also writes several Cord commands to specify how that model should be explored and what kinds of tests should be generated. Then Spec Explorer automates the detailed test case design, generates executable C# tests, and displays those tests visually.

Continued on pg 14

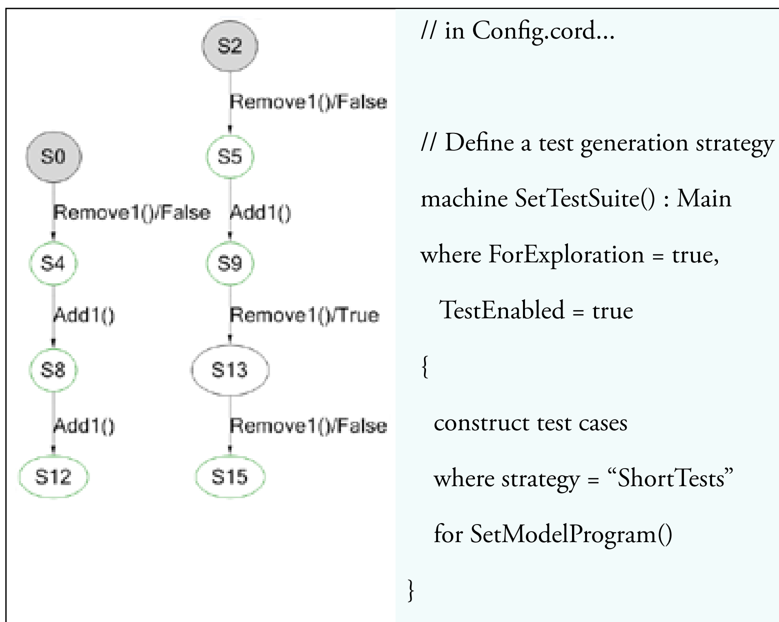

Figure 6: The two tests generated by the 'ShortTests' strategy.

IT applications with Smartesting Test Designer

Test Designer, from Smartesting, is a commercially available model-based testing tool dedicated to IT applications, secure electronic transactions and packaged applications such as SAP or Oracle E-Business Suite. Test cases are generated from a behavior model of the SUT, using requirement coverage and custom scenarios as test selection criteria. Test Designer models are written in a subset of standard UML (class and object diagrams as well as state machine diagrams [6], with OCL annotations [7]). Test Designer supports both a transition-based modeling style (i.e. UML State Machines) and a Pre/Post style (i.e. OCL).

Model elements such as transitions and OCL decisions can be linked to the informal requirements that they cover. Test coverage can then be based on requirements coverage. A range of structural model coverage criteria are also supported, such as transition, decision and effect coverage. The test engineer may also define business scenarios as custom test case specifications that use the UML operations. Test Designer supports both manual and automated test execution, using an offline approach. The generated test cases can be output to test management systems like HP Quality Center or IBM Rational Quality Manager, with bidirectional traceability and full change management for evolving requirements.

Now, we illustrate how Test Designer can be used to model and test the actiTIME application.

actiTIME Overview

actiTIME is a time management program developed by Actimind. Details about its features, and free downloads, can be found on the website **www.actitime.com.** Ordinary users have access to their time track for input, review and corrections. They can also manage projects and tasks, do some reporting and of course they can manage their account (see (1) in Figure 7).

In our sample model we focus on the user time-tracking features of actiTIME version 1.5; after logging into the system the user can specify how much time he spent on a specific task.

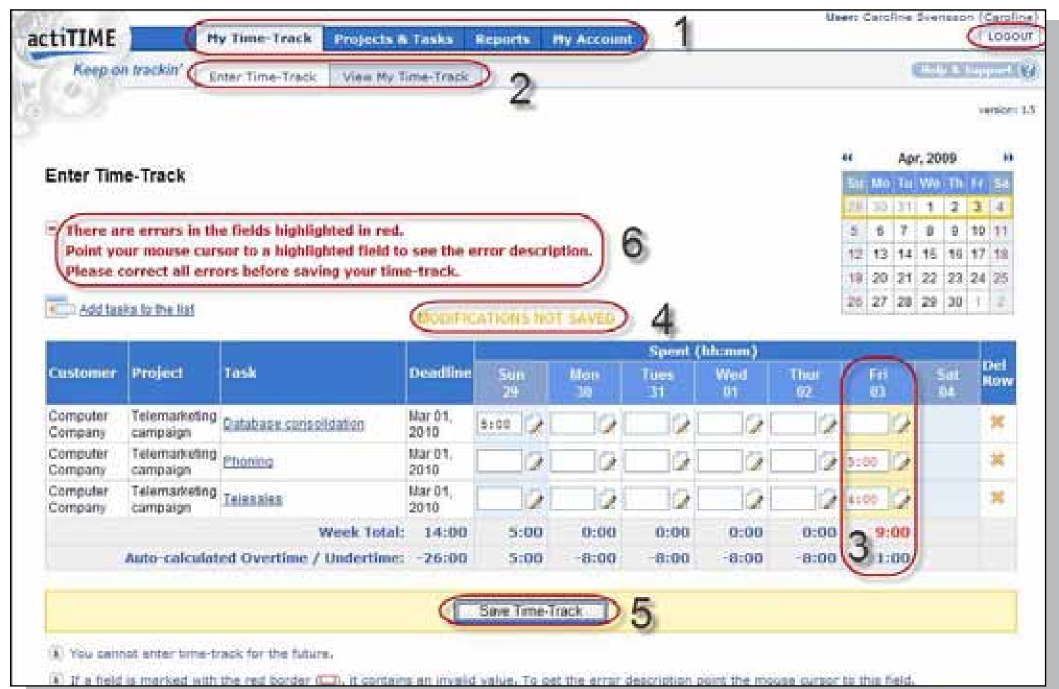

Figure 7: actiTIME User Interface.

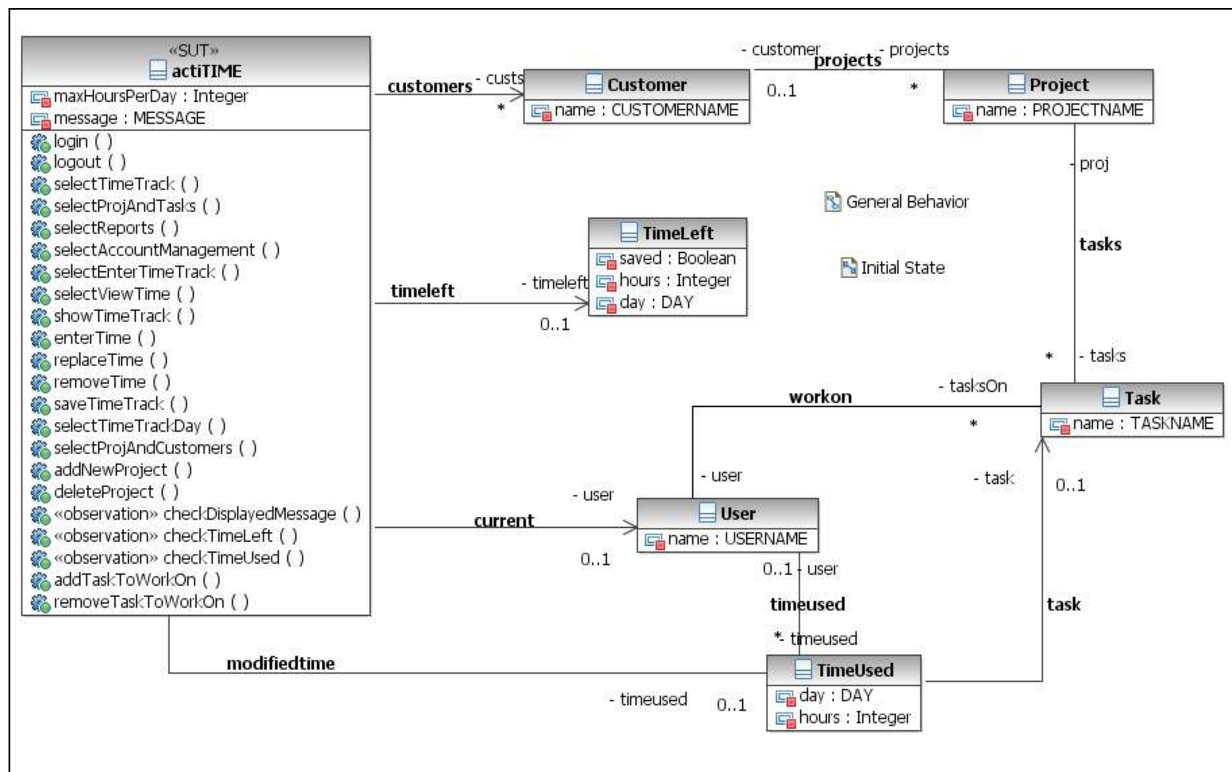

Figure 8: Class Diagram for actiTIME Test Model.

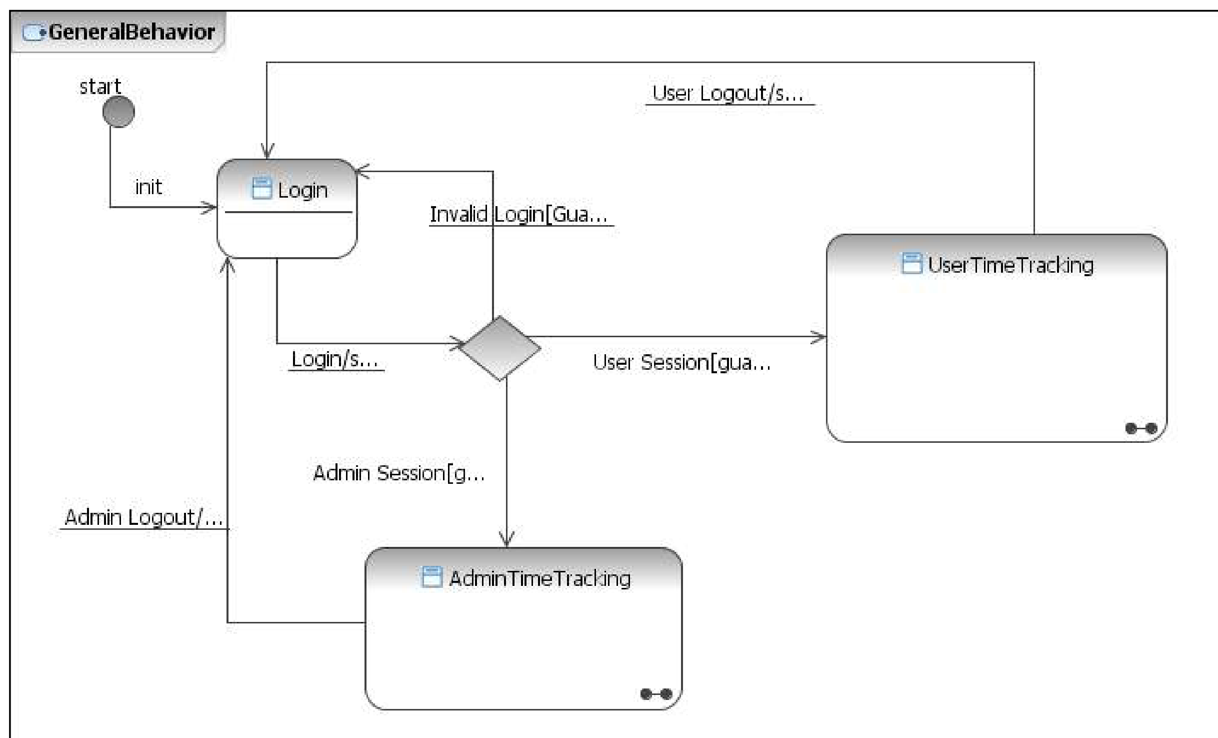

Figure 9: High Level State Machine for actiTIME (partial).

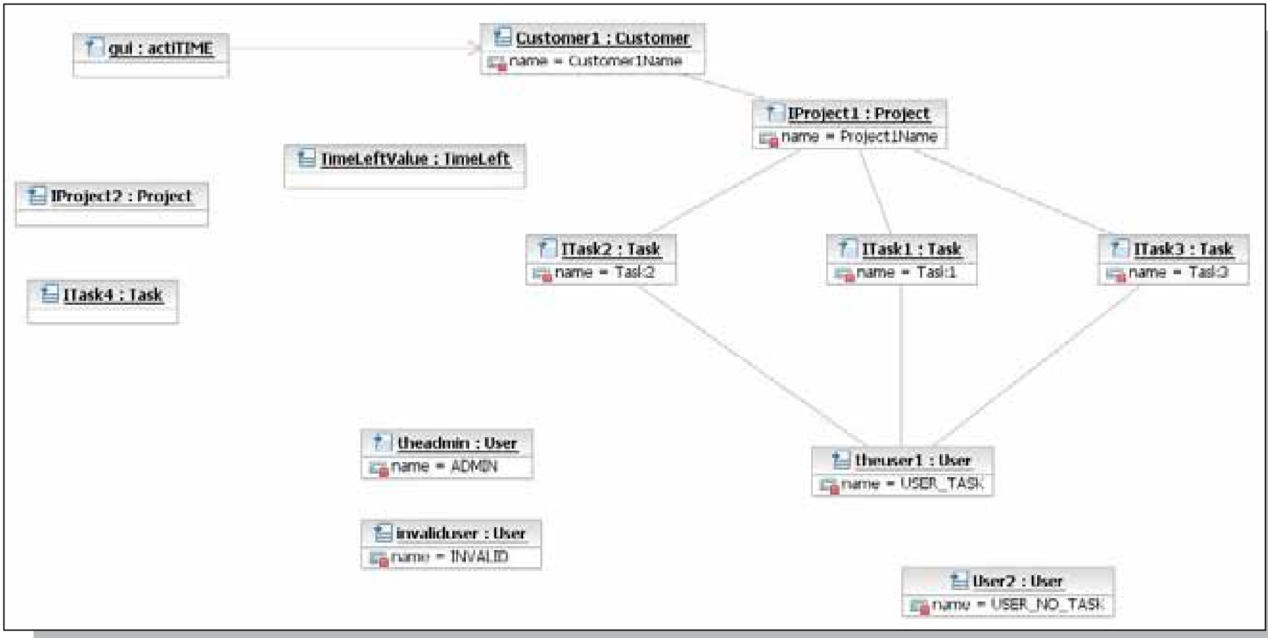

Figure 10: Object Diagram for actiTIME.

actiTIME Test Model

The test model represents the expected behavior of the application, covering the requirements of Table 1. It is based on three UML diagrams (see Figure 8, Figure 9 and Figure 10):

- the class diagram represents the business entities and the user actions to be tested;
- the layered state machine represents the dynamic expected behavior;
- the instance diagram gives some test data and initial configuration of the application.

Test Generation with Test Designer

Figure 11 shows the GUI of Test Designer for the project actiTIME. A list of the generated test cases (structured by test Suites) is displayed on the left, and the details of one test case are displayed on the right. The details of the requirements and test aims that are covered by a particular test step are shown in the right-hand bottom corner.

After test generation, the generated tests are published into a test repository. These tests are ready for manual test execution. Each test is fully documented in the Design Steps Panel.

For test automation, complete script code is generated and

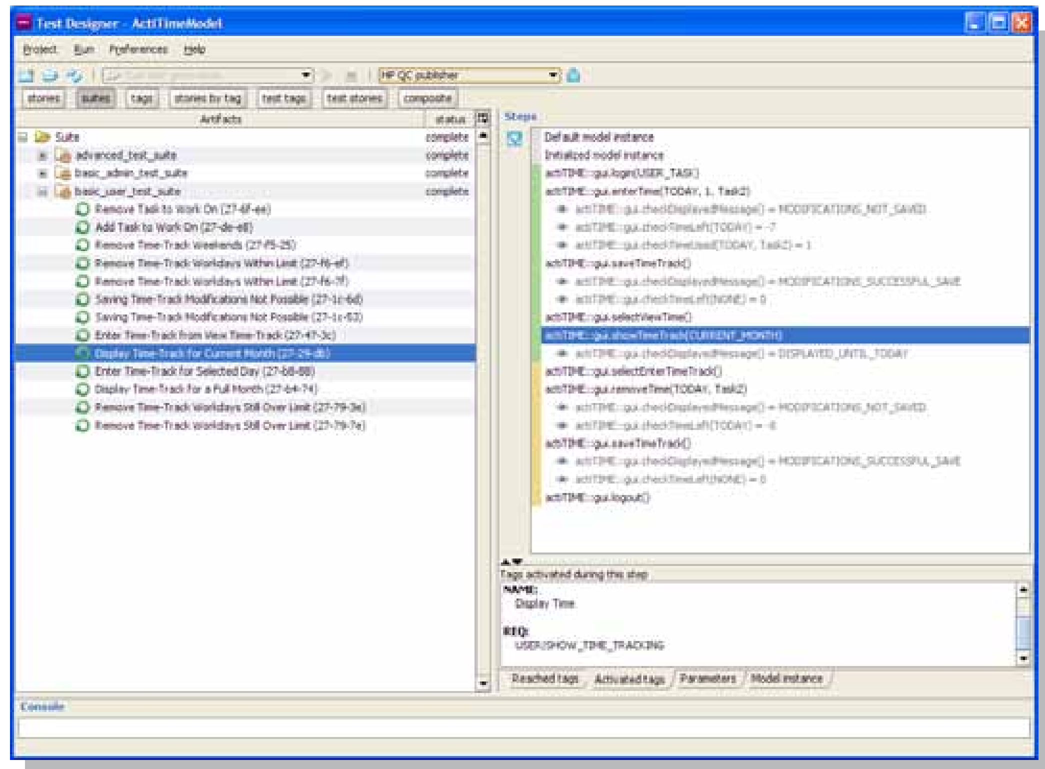

Figure 11: Smartesting Test Designer user interface. Project actiTIME.

maintained for each test case. The remaining (optional) task for the test automation engineer is to implement each key-word used in UML test model so that it is defined as a sequence of lower-level SUT actions. If this is done, the generated test scripts can be executed automatically on the SUT. An alternative approach is to leave the key-words undefined, in which case a human tester must execute the scripts manually.

To sum-up, Test Designer is a typical MBT solution for IT applications, using a subset of UML as input language (class diagrams, state diagrams, instance diagrams, and OCL specification language), providing test generation and publication features both for manual and automated testing.

Key factors for success when deploying MBT

The key factors for effective use of MBT are the choice of MBT methods, the team organization, the qualification of the people involved, and a mature tool-chain.

1. **MBT Methods, requirements and risks:** MBT should be implemented on top of current best practices in functional software testing. The SUT requirements must be clearly defined, so that the test model can be designed from those requirements. The product risks should be well understood, so that they can be used to drive the MBT test generation.
2. **Organization of the test team:** MBT can be a vector for process and productivity improvements. Roles (for example, the test analyst who designs the test model and the test automation engineer who implements the adaptation layer) are reinforced.
3. **Team Qualification - test team professionalism:** The qualification of the test team is an important pre-requisite. Test analysts, test automation engineers, and testers should be professional and have appropriate training in MBT techniques, processes, and tools.
4. **The MBT tool chain:** To maximize the effectiveness of MBT, it is important use an integrated tool chain, including a MBT test generator that integrates with your test management environment and test automation tools.

Expected benefits of MBT

Model-based Testing is an innovative and high-value approach compared to more conventional functional testing approaches. The main expected benefits of MBT may be summarized as follows:

- Contribution to the quality of functional requirements:
  - Modeling for test generation is a powerful means for the detection of "holes" in the specification (undefined or ambiguous behavior).
- Contribution to test generation and testing coverage:
  - Automated generation of test cases;
  - Systematic coverage of functional behavior;
  - Automated generation and maintenance of the requirement coverage matrix;
  - Continuity of methodology (from requirements analysis to test generation).
- Contribution to test automation:
  - Definition of action words (UML model operations) used in different scripts;
  - Test script generation;
  - Generation of skeleton code for a library of automation functions;
  - Independence from the test execution robot.

Conclusion

The idea of model-based testing is to use an explicit abstract model of a SUT and its environment to automatically derive tests for the SUT: the behavior of the model of the SUT is interpreted as the intended behavior of the SUT. The technology of automated model-based test case generation has matured to the point where large-scale deployments of this technology are becoming commonplace. The prerequisites for success, such as qualification of the test team, integrated tool chain availability and methods, are now identified, and a wide range of commercial and open-source tools are available.

Although MBT will not solve all testing problems, it is an important and useful technique, which brings significant progress over the state of the practice for functional software testing effectiveness, and can increase productivity and improve functional coverage.

References

[1] M. Utting, B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kauffman, 2007.

[2] Hung Q. Nguyen, Michael Hackett, and Brent K. Whitlock, *Global Software Test Automation: A Discussion of Software Testing for Executives*, Happy About, 2006.

[3] Elfriede Dustin, Thom Garrett, and Bernie Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*, Addison-Wesley Professional, 2009).

[4] Keith Stobie, Model Based Testing in Practice at Microsoft, *Electronic Notes in Theoretical Computer Science, Volume 111,* 1 January 2005, pages 5-12, *Proceedings of the Workshop on Model Based Testing (MBT 2004).*

[4] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson, *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, in *Formal Methods and Testing*, Springer Verlag, 2008.

[5] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*, Addison-Wesley Professional, 2003.

[6] *Jos Warmer* and *Anneke Kleppe*, *The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition)*, Addison-Wesley Professional, 2003.

About the Authors

In 2007, Mark Utting and Bruno Legeard authored the first industry-oriented book on model-based testing, “Practical Model-Based Testing: A Tools Approach”.

Bruno Legeard is Chief Technology Officer of Smartesting, a company dedicated to model-based testing technologies and Professor of Software Engineering at the University of Franche-Comté (France). He started working on model-based testing in the mid 1990’s and has extensive experience in applying model-based testing to large information systems, e-transaction applications and embedded software.

Mark Utting works for Netvalue.net.nz, using agile techniques to develop Next Generation Genomics Software, and is also an Associate Professor in Computer Science at The University of Waikato in New Zealand. He is the author of more than 50 publications on model-based testing, formal methods for object-oriented and real-time software, and language design for parallelism.

Author Contact Information

Email: Bruno Legeard: legeard@smartesting.com

Email: Mark Utting: marku@cs.waikato.ac.nz