

# Can I Have Some Model-Based GUI Tests Please? – Providing a Model-Based Testing Service through a Web Interface

Mika Katara  
Institute of Software Systems  
Tampere University of  
Technology  
P.O. Box 553  
FI-33101 TAMPERE  
FINLAND  
mika.katara@tut.fi

Antti Kervinen  
Institute of Software Systems  
Tampere University of  
Technology  
P.O. Box 553  
FI-33101 TAMPERE  
FINLAND  
antti.kervinen@tut.fi

Mika Maunumaa  
Institute of Software Systems  
Tampere University of  
Technology  
P.O. Box 553  
FI-33101 TAMPERE  
FINLAND  
mika.maunumaa@tut.fi

Tuula Pääkkönen  
Nokia Technology Platforms  
P.O. Box 1000  
FI-33721 TAMPERE  
FINLAND

Antti Jääskeläinen  
Institute of Software Systems  
Tampere University of  
Technology  
P.O. Box 553  
FI-33101 TAMPERE  
FINLAND  
antti.m.jaaskelainen@tut.fi

## ABSTRACT

Model-based testing (MBT) seems technically superior to conventional test automation systems. However, this technology features some difficulties that can hamper its deployment in industrial contexts. In our earlier work, we have developed a domain-specific MBT solution for GUI testing of Symbian S60 smart phones and their applications. We believe that such a tailor-made solution can be easier to deploy than ones that are more generic. In this paper, we present a web interface that can be used to hide the inherent complexity of the test generation algorithms and provide an easy-to-use MBT service based on the well-known keyword concept. With the help of such interface, we can obtain a better separation of concerns between the test modeling tasks that often require special expertise, and test execution that can be performed by test engineers.

## 1. INTRODUCTION

A ubiquitous problem in software development organizations is how to cut down on the money, time, and effort spent on testing without compromising the quality. A frequent solution is to automate the execution of predefined test cases using software tools. Unfortunately, especially in GUI testing, test automation often does not find the bugs it should and the tools provide a return on the investment only in

regression type of testing. One of the main reasons for this is that the predefined test cases are linear and static in nature – they do not include the necessary variation to cover defected areas of the code and they (almost) never change.

Using model-based testing (MBT) practices to generate tests can introduce more variance to the tests or even generate an infinite number of different tests. Moreover, maintenance of the testware becomes easier when only the models have to be maintained and new updated tests can be generated automatically. Furthermore, developing the test models reveals more bugs than the actual test execution based on those models. This allows early detection of bugs in the software development lifecycle.

Concerning industrial deployment, it has been reported, for instance, that several Microsoft product groups use an MBT tool on a daily basis [2]. However, it seems that large-scale industrial adoption of the methodology is yet to be seen. If MBT is technically superior, why has it not overcome conventional ways of automating tests? Based on some earlier studies [10, 4] as well as our initial experience, there are some non-technical obstacles to large-scale deployment. These include a lack of necessary skills and easy-to-use tools. Moreover, since the roles of the testing personnel are affected by this paradigm change, the test organization needs to be adapted as well [7].

We tackle here the first of these issues, i.e. matching the skills of the testers with easy-to-use tools. A problem with the first generation MBT tools was that they were too general in trying to address too many testing contexts at the same time. We believe that the possibilities of success in MBT deployment will improve with a more *domain-specific solution* that is adapted to a specific context. In our case,

the context is GUI testing of Symbian S60 [11, 12] smart phones. There are over 50 million devices from several vendors shipped with this operating system and GUI platform. Moreover, there are a large number of third party software developers making applications on top of Symbian S60.

Our approach is based on a simple web interface that can be used for providing a model-based testing service. The interface supports setting up testing sessions. In the session, the server sends a sequence of *keywords* to the client that executes them on the SUT. For each received keyword, the client returns the server a Boolean return value: either the execution of the keyword succeeded or not. This on-line approach enables the server to generate tests based on the responses of the client.

The theoretical background of our approach has been introduced previously in [9, 8, 7, 6]. In this paper, the MBT service interface is presented in detail. In addition, an overview of the associated open source tools is provided. The remainder of the paper is structured as follows: In Section 2 we present the background of this paper, i.e. the idea of domain-specific and model-based testing. In Section 3 interfaces of MBT service are described. Use of the web interface, i.e. the interface for test engineers, is demonstrated in Section 4. Finally, Section 5 concludes the paper with some final thoughts.

## 2. DOMAIN-SPECIFIC MBT

Academic research on model-based testing (MBT) has been conducted for almost two decades. In our view, the most fundamental difference between MBT and non-MBT automation is that in the latter case the tests are scripted in some programming or scripting language. In the former case, on the other hand, the tests are generated based on some formal “model” of the SUT that describes the system from the perspective of testing at a high level of abstraction. The definition of a “model” varies greatly depending on the approach. In our approach, a model is a simple state machine that contains states and labeled transitions between them. This formalism enables us to generate tests that introduce variation in the tested *behavior*, for instance, by executing different actions in many different orders allowed by the SUT. In some other MBT approaches, the goal might be to generate all possible data values for some type of parameters. Thus, there are many different types of MBT solutions that do not necessarily have much in common. The algorithms for generating tests from the models may be significantly different depending on the formalism and the testing context.

However, a common goal in many MBT schemes is to increase *coverage* by generating and executing high volumes of tests. Once the MBT regime has been set up and running, the generation of *new* tests based on the models is as easy as running the same old tests again and again.

In spite of these benefits, the industrial adoption of this technology has been slow. Robinson [10] states that the most common problems in deployment are managerial, making easy-to-use tools, and reorganizing the work with the tools. Hartman [4] reports problems with the complexity of the provided solution and counter-intuitive modeling. Our early

experiences support these findings. Moreover, test modeling should be made as easy as possible and it needs to be acknowledged that modeling needs a special kind of expertise. In addition, MBT needs to be adapted to the existing testing process.

We think that a problem with the first generation MBT tools was that they were too general. These tools tried to address excessively many testing contexts at the same time, for instance by generating tests based on UML models that could describe almost any type of system under test (SUT). We believe that the chances of success in MBT deployment will improve with a more *domain-specific solution* that is adapted to a specific context. In our case, the context is GUI testing of Symbian S60 [11, 12] smart phones. Symbian is the most widely spread operating system for smart phones. S60 is a GUI platform build on top the operating system. There are a large number of third party software developers making applications on top of Symbian S60. One driving force in any automation solution for this product family setting is the ability to reuse as many tests as possible when a new product of the family is created.

Symbian S60 domain entails the following problems, among others, from the testing point of view:

- How do you make sure the application under test works with pre-installed applications such as calendar, email, and camera?
- How do you test the interactions between the different applications running on the phone? How do you make sure that the phone does not crash if an end-user installs a third-party application? What happens if an MP3 is being played while the music file is erased by another application?
- How do you verify that your software works with different keyboards and screen resolutions?

The domain concepts of Symbian S60 testing can be described using *keywords* and *action words* [1, 3]. Action words describe the tasks of the user, such as opening the camera application or dialing a specified number. Keywords, on the other hand, correspond to the key presses on the actual phone. Opening a camera application can be done using a short-cut key or a menu, for instance. Alternatively, a keyword can verify that a given string is found from the screen. In each case, an action word needs to be implemented by a sequence of keywords.

Keywords and similar concepts are commonly used in GUI testing tools. We believe that using these concepts in conjunction with MBT can help to deploy the approach in industrial settings. Since testers are already familiar with the keyword concept we just need to hide the inherent complexity of the test generation algorithms and provide as simple user interface as possible.

## 3. INTERFACES FOR MBT SERVICE

In this section, the interface scheme is presented in detail.

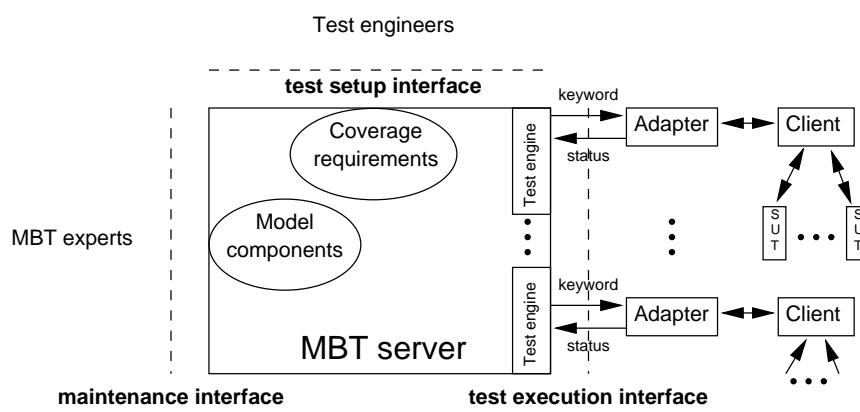


Figure 1: MBT testing server, adapters, clients, and SUTs.

### 3.1 Server and clients

To facilitate the deployment of MBT as much as possible, we have come up with a service concept that provides testers an easy interface to the test generation tools. This concept is based on a test server that is accessed through three interfaces. Firstly, there is an interface through which MBT experts update the test model components on the server. Secondly, and most importantly in this article, there is a web interface through which test engineers can set up tests. Finally, there is an interface for sending keywords to adapters that execute the corresponding events on actual devices. Figure 1 illustrates the scheme.

While the testing server could be installed as a local application in the client machine, there are some practical reasons for dedicating a separate PC for that purpose. The most important reason is that our test generation algorithms can produce better results given more processor time and memory. Fortunately, computing power is very cheap nowadays but it may still pay off to have a dedicated machine. Moreover, the server provides a shared platform for MBT experts to update the software and the model library and test engineers to set up tests. Furthermore, the server does not need to know all the details of the SUT, for instance the form factor or other design issues that may be confidential at the time of testing. It should be enough to know what previously tested member of the product family this new member resembles the most and what the differences are concerning the modeled behavior.

### 3.2 Server to MBT experts: a maintenance interface

MBT experts have two tasks. They update the MBT software, such as the generation algorithms, and they maintain a library of model components on the server. In our domain-specific modeling language, each of the model components describes the behavior of a Symbian S60 application or its subsystem. The description is either at the high level, i.e. at the level of action words, or at a lower level, where a set of action words is refined to sequences of keywords.

Test models, i.e. models used in the test runs, will be built from these components. A set of chosen high-level compo-

nents with the corresponding low-level components developed for the tested device type form the basis for the test model. With a small set of automatic transformations and two generated components, we end up with an executable test model that allows testing interleaving executions of the applications and thereby also their interactions.

### 3.3 Server to test engineer: a test setup interface

There are a number of parameters that need to be given in order to start a test run. The most important are:

1. SUT types: which phone models will be used in the test run. This affects the automatic selection of test model components.
2. Test model: which applications will be used in the test run. Based on this choice the test model components are selected and composed together to form a single test model that will be used in the test run.
3. Test mode: the test can be executed in *smoke test*, *bug hunt* and *use-case testing* mode. In each mode, a coverage criterion should also be given. The criterion defines when the test run can be stopped.
4. Number of clients: how many clients can be used to execute the test. Using more than one client can often improve the time in which the test is finished. For example, complicated coverage criterion can often be divided into smaller criteria that can be fulfilled in concurrent test executions.

In addition to the previous parameters, one can also specify the test generation algorithm, connection parameters and logging system.

To support different types of testing in the various phases of the testing process, the server supports the three testing modes mentioned above. In the smoke testing mode the server generates tests in a breadth-first search fashion until the coverage criterion has been fulfilled, for instance, 30 minutes have passed or 1000 keywords have been executed. In

the use case mode, the tester inputs a use case (in the form of a sequence of action words) to the server that then generates tests that cover that use case. The main motivation for this mode is to be compatible with the existing testing processes: tests are usually based on requirements and the test results can be reported based on the requirements coverage.

In the bug-hunting mode, the server generates a much longer sequence of keywords that try to interleave the behavior of the different applications as much as possible in order to find hard-to-find bugs related to mutual exclusion, memory leaks, etc. In addition, it is also possible to leverage use cases to guide the test generation to those areas of the modeled behavior that might be buggy.

When the test setup is ready, the corresponding test model is automatically built from components of the model library. After that, the given coverage criterion is automatically split so that there is a chunk for every client to cover. Finally, one *test engine* process per every client is launched to listen to a TCP/IP connection. A test engine will serve a client until its part of the coverage criterion has been covered or it is interrupted. Now the MBT server is ready for the real test run, where the clients and the server communicate through a test execution interface.

### 3.4 Server to adapter: a test execution interface

To start a test run, the test engineer starts the devices that should be tested as well as the clients and adapters. The adapters are configured so that they connect to the test engines waiting on the server. Test execution on the client starts immediately when its adapter has been connected to the test engine.

During the execution, a test engine repeats a loop where it first sends a keyword to an adapter. The adapter, with the help of the client it is controlling, converts the keywords to an input event or an observation on the SUT. For instance, there are different keywords for pushing a button on the phone keypad and verifying that a given string is found on the screen. After that, the adapter returns the status of the keyword execution, i.e. a Boolean value denoting success or failure, to the test engine. In a normal case, for instance, when the status of the keyword execution is allowed by the test model, the server loops and sends a new keyword to the adapter.

Otherwise, unexpected behavior of the SUT is detected, maybe due to a bug in the SUT, and the server starts a shutdown or recovering sequence. It informs the adapter that it has found an anomaly. The adapter may then save screenshots, a memory dump or other information useful for debugging. It also acknowledges to the server that it has finished operations. Finally, the test engine may either close the connection, or it may try to recover from the error by sending some keywords again, for instance to reboot the SUT.

Regardless of the mode, during a test session a log of executed keywords is recorded for debugging purposes. When a failure is noticed, the log can be used for repeating the same sequence of keywords in order to reproduce the failure.

GUI testing can sometimes be slow, even with the most sophisticated tools. In order to cope with this, our solution will support concurrent testing of several phones using one server. Testing a new Symbian S60 application could be done so that one client is used for testing the application in isolation from other applications, while other clients are testing some application interactions.

## 4. EXAMPLE

The following example illustrates how a model-based test session is set up and run using an MBT server. The point of view is the test engineer's who orders the service and manages the test execution. We assume that MBT experts have created high-level test model components for each base application in S60 platform, including Calendar and Camera, for instance. In addition, the model component library in the MBT server contains also low-level implementations of actions in the high-level components for two phone models: *A* and *B*.

The test engineer decides to test Calendar and Camera applications in model *A* phones. The engineer has three phones that can be used in the test run. To set up the test session, the engineer opens a web page hosted in the MBT server (test setup interface in Figure 1). He chooses to test model *A* phones, and selects the test model to be build from Calendar and Camera components.

MBT server asks for a coverage criterion for the test run. To help answering, the server also shows a list of interesting actions in the chosen test model components. An interesting action is a high-level description of what a high-level test model does in a certain point of its execution. Such actions are indicated by MBT experts when they create the model components.

From Calendar, the engineer chooses interesting actions of testing a day view, a week view and adding calendar entries. Similarly, taking a photo, recording a video, changing image settings and adjusting display contrast are chosen to be tested in Camera. A valid coverage criterion is an interesting action or a pair of valid coverage criteria combined with *and*, *or* or *then* operators. The operators denote that both, either one, or the latter after the former criterion must be covered, respectively. Because the engineer wants that every chosen action is tested despite of the order, he forms the criterion by combining the interesting actions with *and* operators. Test mode is set to bug hunt to introduce extra variation in the test generation.

The last parameter for the test run is the number of clients, which the engineer sets to three. This means that there will be three concurrently running tests, each of which aim to cover their part of the given coverage criterion. The criterion at hand can be easily split in three: each test run has a disjoint set of actions to cover. However, if there are *thens* instead of *ands* in the criterion, it cannot be split. In that case, the same criterion is used in every test run. Because of a random component in test generation algorithms, parallel test runs are very likely to take different paths. Thus, one test run may fulfill the coverage criterion faster than another. Therefore, parallel test runs may result in a speed gain even if the criterion could not be split.

Having set the test parameters, the test engineer sends the parameters to the server. The server responds with three TCP/IP port numbers and starts three test engines, which listen to the ports. Next, the engineer sets up three test clients. Each client controls one phone in this example. Finally, the clients connect to the test engines in the MBT server through the given ports. Test runs start: clients request keywords, test engines respond, clients execute the keywords, send results and ask for more keywords.

## 5. DISCUSSION

We are currently conducting industrial case studies to assess the applicability of the MBT server. Thus, the MBT service described in this paper is not yet available for commercial use. There is no cost benefit analysis available either. However, the software running in the server and test models for basic Symbian S60 applications have been released under the MIT open source license. In addition, a prototype adapter for using Mercury QTP [5] as a client will be available. Developing an adapter for some other connectivity solution between phones and a PC is straightforward, at least in principle, since the adapter's behavior is simple: execute the keyword obtained from the server on the SUT and return "true" or "false" to the server based on the success of the execution. Future work also includes Eclipse [13] based tools for test model design.

In our experience, apart from developing adapters for different clients, a big challenge in MBT deployment is the creation of test models, which calls for domain expertise and modeling skills. While such skills can be hard to find, we think that the majority of testers can use the tools with the very simple interface we have described. The experts who master the art of test model creation are then responsible for creating new models and updating the old ones on the server side.

We believe that the approach will facilitate the deployment of MBT in the Symbian S60 context. Moreover, since keywords and actions words are common concepts in automated GUI testing, it should be possible to adapt the scheme to other GUI testing domains as well.

## Acknowledgements

This paper reports ongoing results of the TEMA research project funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq Software, F-Secure, Plenware Group and Mercury Interactive. For details of the project, see <http://practise.cs.tut.fi/project.php?project=tema>.

## 6. REFERENCES

- [1] H. Buwalda. Action figures. *STQE Magazine*, March/April 2003, pages 42–47, 2003.
- [2] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with Spec Explorer. In *Proceedings of Formal Methods 2005*, number 3582 in Lecture Notes in Computer Science, pages 542–547. Springer, 2005.
- [3] M. Fewster and D. Graham. *Software Test Automation: Effective use of test execution tools*.

Addison-Wesley, 1999.

- [4] A. Hartman. AGEDIS project final report. Available at <http://www.agedis.de/documents/FinalPublicReport%28D1.6%29.PDF>, 2004. Cited May 2007.
- [5] HP. Mercury QuickTest Pro homepage. <http://www.mercury.com>. Cited May 2007.
- [6] M. Katara and A. Kervinen. Making model-based testing more agile: a use case driven approach. In *Proceedings of the Haifa Verification Conference 2006*, number 4383 in Lecture Notes in Computer Science, pages 219–234. Springer, 2007.
- [7] M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Satama. Towards deploying model-based testing with a domain-specific modeling approach. In *Proceedings of TAIC PART – Testing: Academic & Industrial Conference*, pages 81–89, Windsor, UK, Aug. 2006. IEEE Computer Society.
- [8] A. Kervinen, M. Maunumaa, and M. Katara. Controlling testing using three-tier model architecture. In *Proceedings of the Second Workshop on Model Based Testing (MBT 2006)*, volume 164(4) of *Electronic Notes in Theoretical Computer Science*, pages 53–66, Vienna, Austria, 2006. Elsevier.
- [9] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. Model-based testing through a GUI. In *Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005)*, number 3997 in Lecture Notes in Computer Science, pages 16–31. Springer, 2006.
- [10] H. Robinson. Obstacles and opportunities for model-based testing in an industrial software environment. In *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 118–127, Nuremberg, Germany, Dec. 2003. Available at <http://www.model-based-testing.org/ObstaclesAndOpportunities.pdf>. Cited May 2007.
- [11] S60. <http://www.s60.com>. Cited May 2007.
- [12] Symbian. <http://www.symbian.com/>. Cited May 2007.
- [13] The Eclipse Foundation. Eclipse homepage. <http://www.eclipse.org/>. Cited May 2007.