Model-Based Testing Service on the Web

Antti Jääskeläinen[1], Mika Katara[1], Antti Kervinen[1],
Henri Heiskanen[1], Mika Maunumaa[1], and Tuula Pääkkönen[2]

[1] Tampere University of Technology
Department of Software Systems
P.O.Box 553
FI-33101 Tampere, FINLAND
{antti.m.jaaskelainen,firstname.lastname}@tut.fi
[2] Nokia Devices
P.O.Box 68
FI-33721 Tampere, FINLAND

Abstract. Model-based testing (MBT) seems to be technically superior to conventional test automation. However, MBT features some difficulties that can hamper its deployment in industrial contexts. We are developing a domain-specific MBT solution for graphical user interface (GUI) testing of Symbian S60 smartphone 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 service concept and an associated web interface that hide the inherent complexity of the test generation algorithms and large test models. The interface enables an easy-to-use MBT service based on the well-known keyword concept. With this solution, a better separation of concerns can be obtained between the test modeling tasks that often require special expertise, and test execution that can be performed by testers. We believe that this can significantly speed up the industrial transfer of model-based testing technologies, at least in this context.

1 Introduction

A widespread 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 test automation tools. Unfortunately, especially in graphical user interface (GUI) testing, test automation often does not find the bugs that 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. Moreover, since GUI is often very volatile, it takes time to update the test suites to test the new version of the system under test (SUT). Hence, costly but flexible manual testing is still often chosen as the primary method to ensure the quality, at least in the context of mass consumer products, where GUIs are extremely important.

Model-based testing (MBT) practices [1] that generate tests automatically can introduce more variance to the tests, or even generate an infinite number of different tests. Moreover, maintenance of the testware should become easier when only the models

have to be maintained and new updated tests can be generated automatically. Furthermore, developing the test models may reveal more bugs than the actual test execution based on those models. Since model development can be started long before the SUT is mature enough for automatic test execution, detection of bugs early in the product lifecycle is supported.

Concerning industrial deployment of MBT, it has been reported, for instance, that several Microsoft product groups use an MBT tool (called Spec Explorer) on a daily basis [2]. However, it seems that large-scale industrial adoption of the methodology is yet to be seen. If MBT is technologically superior, why has it not overcome conventional ways of automating tests? Based on some earlier studies [3, 4] as well as our initial experience, it seems that there are some non-technological obstacles to large-scale deployment. These include the lack of easy-to-use tools and necessary skills. Moreover, since the roles of the testing personnel are affected by this paradigm change, the test organization needs to be adapted as well [5].

In this paper, we tackle the first of these issues, i.e. matching the skills of the testers with easy-to-use tools. We think that one 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 the GUI testing of Symbian smartphone applications. There have been cumulatively over 150 million Symbian smartphones shipped [6]. We concentrate on the devices with the S60 GUI framework [7], which is the most commonly found application platform in the current phone models. In addition to device manufacturers, there are a large number of third party software developers making applications on top of Symbian S60. Compared to a more generic approach, based on UML and profiles, for instance [8], our tools should effect a higher level of usability and automation in this particular context.

The background of our approach has been introduced previously in [5, 9–11]. In this paper, based on earlier work [12, 13], the MBT service interface is presented in detail. Our approach is based on a simple web GUI that can be used for providing a model-based testing service. The interface supports setting up MBT sessions. In a session, the server sends a sequence of *keywords* to the client, which executes them on the SUT. For each received keyword, the client returns to 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, in a way somewhat similar to the Spec Explorer tool [2].

Our scheme should facilitate industrial deployment by minimizing the tasks of the testers. In addition to the service interface, this paper presents an overview of the associated open source tools. The remainder of the paper is structured as follows: In Section 2, we present the background of this paper, i.e., domain-specific MBT for S60 GUI testing. Sections 3 and 4 describe the modeling formalism and the associated tool set. In Section 5, the service concept is introduced in detail including the interfaces that we have defined. Finally, Section 6 concludes the paper with a final discussion including ideas for future work.

2 Domain-specific MBT

Research on model-based testing (MBT) has been conducted widely in both industry and academia. From the practical perspective, the 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 a formal model of the SUT. The model describes the system from the perspective of testing at a high level of abstraction. However, the definition of a "model" varies greatly, depending on the approach [1]. In our approach, a model is a parallel composition of Labeled State Transition Systems (LSTSs). 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 execute high volumes of different 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. Obviously, old tests can still be repeated for debugging purposes if necessary.

In spite of these benefits, the industrial adoption of this technology has been slow. Robinson [3] states that the most common problems in deployment are the managerial difficulties, the making of easy-to-use tools, and the reorganization of 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, it must be acknowledged that modeling needs a special kind of expertise that may not be available in a testing organization. However, such expertise might be available as a service, especially when operating in a specialized domain such as testing smartphone applications.

We think that a problem with the first generation MBT tools was that they were too general. These tools tried too much to address many testing contexts at the same time, for instance by generating tests based on UML models that could describe almost any type of SUT. We believe that the chances of success in MBT deployment will improve with more domain-specific solutions that are adapted to specific contexts. In our case, the context is the GUI testing of Symbian S60 [6, 7] smartphone applications. Symbian is the most widely spread operating system for smartphones and S60 is a GUI platform built on the top of it. 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. Thus, we have built our test model library to support the reuse of test models.

In addition, in terms of industrial adoption, MBT needs to be adapted to the existing testing processes that are shifting towards more agile practices [14] from the traditional ones based on the V-model [15] and its variations. In agile contexts, on the one hand, developers are already relying on test automation to support refactoring and generally understand its benefits as compared to manual testing. On the other hand, it

seems especially important to provide easy-to-use tools and services that do not place an additional burden, such as that of test modeling, on the project personnel. We have identified a minimum of three modes [11] to be supported in agile processes: *smoke testing* should be performed in each continuous integration cycle; user stories can be tested in a *use-case testing* mode; and there should be a *bug hunting* mode, whose only purpose is to support finding defects efficiently in long test runs.

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

- How to make sure the application under test works with pre-installed applications such as calendar, email, and camera?
- How to test the interactions between the different applications running on the phone? How to make sure that the phone does not crash if a user installs a third-party application? What happens if, for instance, some application attempts to delete an MP3 file that is being played by another application?
- How to test 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* [16, 17]. Action words describe the tasks of the user, such as opening the camera application, dialing a specified number, or inserting the number of the recipient to a message. Keywords, on the other hand, correspond to physical interaction with the device such as the key presses and observations. Each action word needs to be implemented by at least one sequence of keywords. For example, starting a camera application can be performed using a short-cut key or a menu, for instance, and verifying that a given string is found from the screen. The verification enables checking that the state of the model and state of the SUT match each other during the test run.

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 solution and provide as simple a user interface as possible. The existing test execution tools that already implement keywords should be adaptable to receive a sequence of keywords from a server. The role of the server is to encapsulate the test model library and the associated test generation heuristics. Based on a single keyword execution on the SUT, the client tool returns to the server a Boolean value based on success or failure of the execution. The server then selects the next keyword to be sent to the client based on this return value.

3 Modeling Formalism

In this section, the fundamentals of our modeling formalism are presented for the interested reader. As already mentioned, we use Labeled State Transition Systems (LSTSs) as our modeling formalism. This is an extension of the Labeled Transition System (LTS) format with labels added to states as well as to transitions. The formal definition is presented below. It should be noted that while each transition is associated with exactly one action, any number of attributes may be in effect in a state.

Definition 1 (LSTS). *A* labeled state transition system*, abbreviated LSTS, is defined as a sextuple* $(S, \Sigma, \Delta, \hat{s}, \Pi, val)$ *where S is the set of* states*,* Σ *is the set of* actions *(transition labels),* $\Delta \subseteq S \times \Sigma \times S$ *is the set of* transitions*,* $\hat{s} \in S$ *is the* initial state*,* Π *is the set of* attributes *(state labels) and* $val : S \longrightarrow 2^{\Pi}$ *is the* attribute evaluation function*, whose value* $val(s)$ *is the set of attributes in effect in state s.*

In our approach, the models are divided into four categories according to their uses: *action machines*, *refinement machines*, *launch machines* and *initialization machines*. Action machines are used to model the SUTs on the action word level. Thus, they are the main focus of the modeling work. Keyword implementations for action words are defined in refinement machines. Together, these machines form most of the model architecture; the remaining two types are focused on supportive tasks. Launch machines define keyword sequences required to start up an action machine, such as switching to a specific application. Initialization machines, on the other hand, define sequences for setting the SUT into the initial state assumed by action machines and are executed before the actual test run. They can also be used to return the SUT back to a known state after the test. Both of these functions have simple default actions. Hence, explicitly defined launch and initialization machines are rarely needed.

Concerning the keywords, many of them require one or more parameters to define their function. Sometimes these are fixed to the GUI, such as a parameter that defines which key to press, but sometimes they represent real-world data: a date or a phone number, for example. Embedding such information directly into the models is problematic, because they would be limited to a fixed set of data values and possibly tied to a specific test configuration. Another problem with the use of data is that storing it in state machines requires duplicate states for each possible value of data, which quickly results in a state space explosion [18]. To solve these problems, we have developed two methods of varying the data in models: localization data and data statements.

The basic function of *localization data* is to hold the text strings of the GUI in different languages, so that the models need not be tied to any specific language variant of the SUT. The data is incorporated into the model by placing a special identifier in a keyword. When the keyword is executed, the identifier is replaced with the corresponding element from the localization tables. More complicated use of data can be accomplished by placing *data statements* (Python [19] code) in actions. These statements may be used in any actions, not just keywords. Data provided by external *data tables* can be used in these data statements.

In order to be used in a test run, the models must be combined in *parallel composition*. The models involved in this process are action machines, refinement machines, launch machines (both explicitly defined and automatically generated), and a special model called the *task switcher*. The latter is generated to manage some of the synchronizations between the models. In the composition, the models are examined and rules generated for them according to the domain-specific semantics to determine what actions can be executed in a given state. As usual, the composition can be used to create one large test model that combines all the various components, or it can be performed on the fly during the test run. We have found the latter method to be preferable, since combining a large number of models can easily result in a serious state explosion prob-

lem. The definition of the parallel composition, extended from [20] for LSTSs, is the following:

Definition 2 (Parallel composition $\|_R$). *$\|_R (L_1,\ldots,L_n)$ is the* parallel composition *of LSTSs $L_1,\ldots,L_n$, $L_i = (S_i,\Sigma_i,\Delta_i,\hat{s}_i,\Pi_i,val_i)$, according to* rules *$R$; $\forall i,j; 1 \leq i < j \leq n : \Pi_i \cap \Pi_j = \emptyset$. Let Σ_R be a set of resulting actions and $\surd$ a "pass" symbol such that $\forall i; 1 \leq i \leq n : \surd \notin \Sigma_i$. The rule set $R \subseteq (\Sigma_1 \cup \{\surd\}) \times \cdots \times (\Sigma_n \cup \{\surd\}) \times \Sigma_R$. Now $\|_R (L_1,\ldots,L_n) = (S,\Sigma,\Delta,\hat{s},\Pi,val)$, where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \{a \in \Sigma_R \mid \exists a_1,\ldots,a_n : (a_1,\ldots,a_n,a) \in R\}$
- $((s_1,\ldots,s_n),a,(s'_1,\ldots,s'_n)) \in \Delta$ *if and only if there is* $(a_1,\ldots,a_n,a) \in R$ *such that for every* i $(1 \leq i \leq n)$ *either*
  - $(s_i,a_i,s'_i) \in \Delta_i$ *or*
  - $a_i = \surd$ *and* $s_i = s'_i$
- $\hat{s} = (\hat{s}_1,\ldots,\hat{s}_n)$
- $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$
- $val((s_1,\ldots,s_n)) = \{\pi \in \Pi \mid \exists i; 1 \leq i \leq n : \pi \in val_i(s_i)\}$

The composition is based on a rule set which explicitly defines the synchronizations between the actions. An action of the composed LSTS can be executed only if the corresponding actions can be executed in each component LSTS, or if the component LSTS is indifferent to the execution of the action. In some, extreme cases an action may require the cooperation of all the component LSTSs, or a single component LSTS may execute an action alone. In practice, however, most actions in our models are executed singly or synchronized between two components, though larger synchronizations also exist.

An important concept in the models is the division of states into *running* and *sleeping states*. In more detail, running states contain the actual functionality of the models, whereas sleeping states are used to synchronize the models with each other. The domain-specific semantics ensure that exactly one model is in a running state at any time, as is the case with Symbian applications. As testing begins, the running model is always the task switcher. Running and sleeping states are defined implicitly according to the transitions in the models.

4 Overview of the Tools

In this section, we provide an overview of the toolset supporting our approach. The toolset is currently under construction. The tool architecture is illustrated in Figure 1. The toolset can be divided into four parts plus a database. The first is the model design part, which is used for creating the component models and data tables. The second is the test control part, where tests are launched and observed. The third is the test generation part that is responsible for assembling the tests and controlling their execution. The fourth is the keyword execution part, whose task is to communicate with the SUT through its GUI.

Concerning the model design part of the toolset, the tools are used to create the test models and prepare them for execution. There are two primary design tools: Model

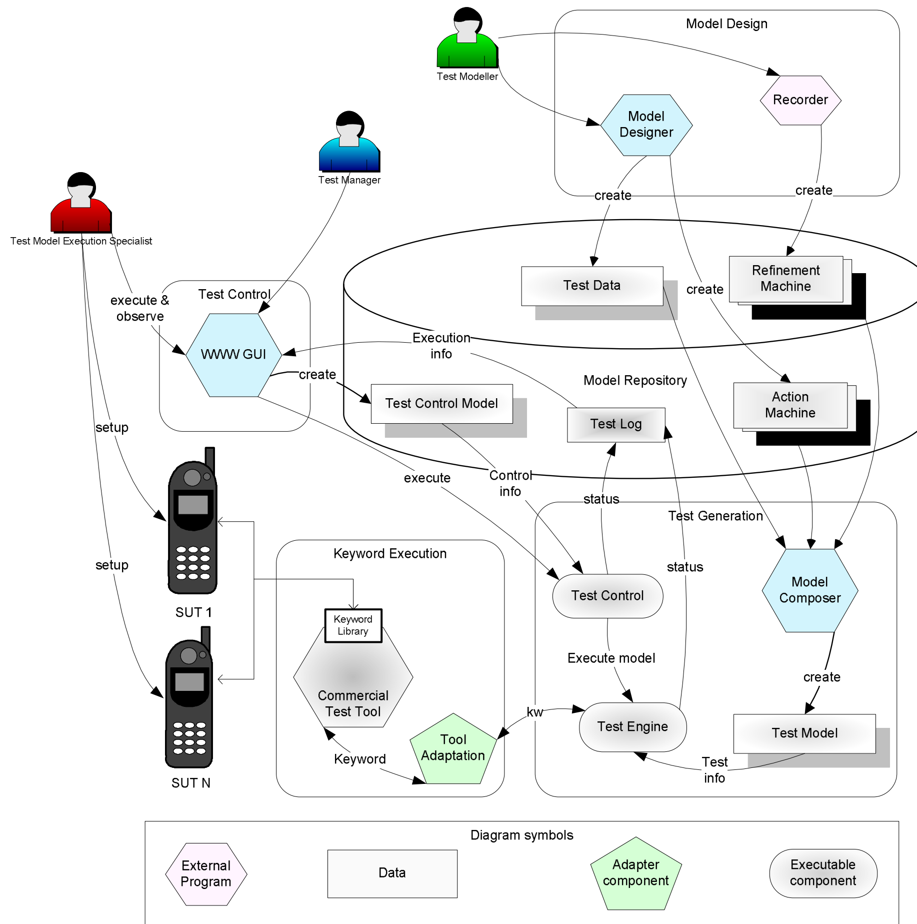

Fig. 1. Test tool architecture.

Designer [13] and Recorder [21]. The latter is an event capturing tool designed to create keyword sequences out of GUI actions; these sequences can then be formed into refinement machines. Model Designer, on the other hand, is the main tool for creating action machines and data tables. It is also responsible for assembling the models into a working set ready for testing; even refinement machines created with Recorder pass through Model Designer. The elements of this working set are placed into the model repository.

After the models with their associated information have been prepared with the design tools, the focus moves to the test control part. This part contains a web GUI which is used to launch the test sessions. Once a test session has been set up, the Test Control tool in the test generation part of the toolset takes over. First, it checks the *coverage requirement* (a formal test objective) that it received and determines what

model components are required for the test run. These are given to Model Composer, which combines them into a single model on the fly. The model is managed by Test Engine, which determines what to do next, based on the parameters it receives from Test Control. Both Test Control and Test Engine report the progress of the test run into a test log, which may be used for observing, debugging, or repeating the test.

As keywords are executed in the model, Test Engine relays them to the keyword execution part. The purpose of this part is to handle their execution in the SUT. The SUT responds with the success status (true or false) of the keyword, which is then relayed back to Test Engine. The first link in the communication between Test Engine and the SUT is handled by a specific adapter tool, which translates the keywords into a form understood by the receiver and manages the gradual execution of some more complex keywords. The next part in the chain is the test tool which directly interacts with the SUT. The nature of this tool depends on the SUTs in question and is not provided alongside the toolset. The users of the toolset must provide their own test tool and use the simple interface offered by the adapter. In our case, we have used commercial components, namely Mercury Functional Testing for Wireless (MFTW) and Mercury QuickTest Professional (QTP) [22].

We have designed the architecture to support the plugging-in of different test generation heuristics. Currently, we have implemented three heuristics which allow us to experiment with the tools: a purely random heuristics that can be used in bug hunting mode, and two heuristics based on game-theory [11] to be used in the use case testing mode: a single thread and a two thread version. The difference between the two is that the latter continues to search an optimal path to a state fulfilling the coverage requirement, while the other thread waits for a return value from the client executing a keyword.

It is anticipated that in deploying our approach the testing personnel should consist of the following roles (see Figure 1): test manager, test modeler, and test model execution specialist. The test manager defines the entry and exit criteria for the test model execution, and defines which metrics are gathered. The test manager should also focus on communicating the testing technology aspects. This includes explaining how model-based testing compares to conventional testing methods and advocating reasons for and against using it for management and testing personnel. In these respects, model-based testing is similar to any new process initiation.

The main goal of the test modeler is to update and maintain the test model library using the Model Designer and Recorder tools based on product specifications if such exist. The test modeler can also be responsible for designing the execution of the model and setting up the environment accordingly.

The test model execution specialist orders the test sessions from the web GUI according to the chosen test strategy. He/she also observes the test execution to ensure that the models are used according the agreed principles and test data. Another focus of this role is in reporting the results and faults onward. The purpose is to document the test model usage and testware in a way that enables its reuse.

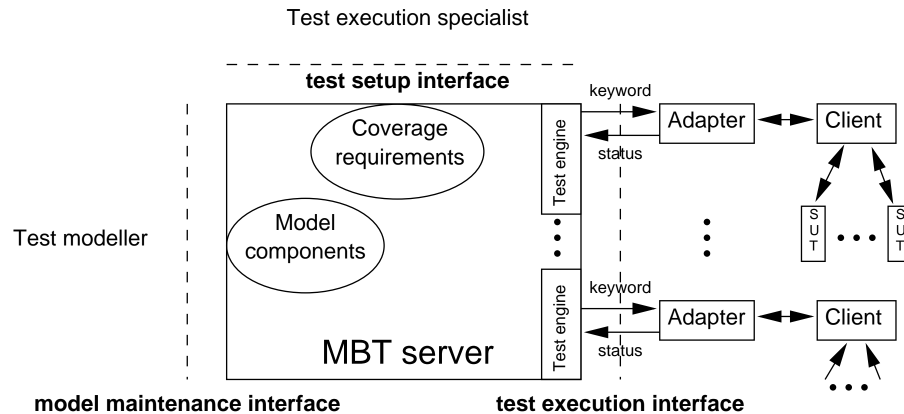

Fig. 2. MBT testing server, adapters, clients, and SUTs.

5 Providing a Symbian S60 Test Service

In this section, the service scheme is presented in detail. The following subsections describe the interfaces provided by our server.

5.1 Server and Clients

The architecture of the toolset described earlier enables a client-server scheme where the keyword execution and test generation parts are separated. To facilitate the deployment of model-based GUI testing in the context of Symbian S60 applications, we have set up a prototype version of the server that implements the test generation part. It provides testers an easy interface to the MBT tools.

The server is accessed through three interfaces. First, there is an interface through which test modelers update the test model components on the server. Second, there is a web interface through which test execution specialists can set up tests. Finally, there is an interface for sending keywords to adapters which execute the corresponding events on actual devices. Figure 2 illustrates the scheme.

Although the MBT 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 some of our test generation algorithms, i.e. the ones based on game heuristics, can produce better results given more processor time and memory. Fortunately, computing power is very cheap nowadays but it still pays off to have a dedicated machine. Moreover, the server provides a shared platform for test modelers to update the model library and test execution specialists to set up tests. Furthermore, all the users of the server do not need to know the details of the SUT, for instance the physical form or other design issues that may be confidential at the time of testing. For the purposes of test modeling, 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.

5.2 Test Setup Interface

There are a number of parameters that need to be given in order to start a test run. The most important ones 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, but it can also be used to guide the test generation as in the case of use-case testing mode.
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, a complicated coverage criterion can often be divided into smaller criteria that can be fulfilled in concurrent test executions.
5. 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, which then generates tests to cover that use case using the game heuristics. As already discussed, the main motivation for this mode is compatibility with the existing testing processes: the tests are usually based on requirements and the test results can be reported based on the coverage of the requirements. In the bug-hunting mode, in addition to purely random generation, the server could generate a much longer sequence of keywords that tries to interleave the behavior of the different applications as much as possible in order to detect hard-to-find bugs related to mutual exclusion, memory leaks, etc.

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 could be split so that there is a chunk for every client to cover. Finally, one *test engine* process per every client could be 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, during which the clients and the server communicate through the test execution interface.

5.3 Test Execution Interface

To start a test run, the test execution specialist starts the devices to be used as targets in the tests 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 test execution tool it is controlling, converts the keyword into an input event or an observation on the SUT. As already discussed, there are different keywords for pushing a button on the phone keypad and verifying that a given string is found on the screen, for instance. 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, 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 sends an acknowledgement of having finished operations to the server. Finally, the test engine may either close the connection, or 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, we should extend our solution to support the concurrent testing of several target 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.

5.4 Using the Web GUI

The testers interact with the server using a web interface. The interface has been implemented in AJAX [23] and it consists of several different views. In the following, we will introduce the basic usage of the interface step by step.

When the tester wants to start a test session, he or she first logs into the system. After that, the system offers two alternatives: either to start a session by repeating a log from some previous session or simply from scratch. In the latter case, a model configuration must next be selected. Such a configuration can consist of models of certain applications whose interactions should be tested, for instance. Next, a view called the coverage requirement editor is opened (see Figure 3). In this view, the tester can construct a new coverage requirement from actions of the model components included in this configuration. Since the number of different actions can be large, there is a possibility to limit the shown actions to those marked “interesting” by the test modelers. The coverage requirement is composed of actions and operators *THEN*, *AND*, and *OR*, as well as parentheses. As an example, consider a requirement for sending a multimedia message (MMS) from one SUT to another with an attachment:

```
action Messaging1-Main:NewMMS THEN
action Messaging1-MMS:InsertObject THEN
action Messaging1-MMS:Select THEN
action Messaging1-Sender:Send THEN
```

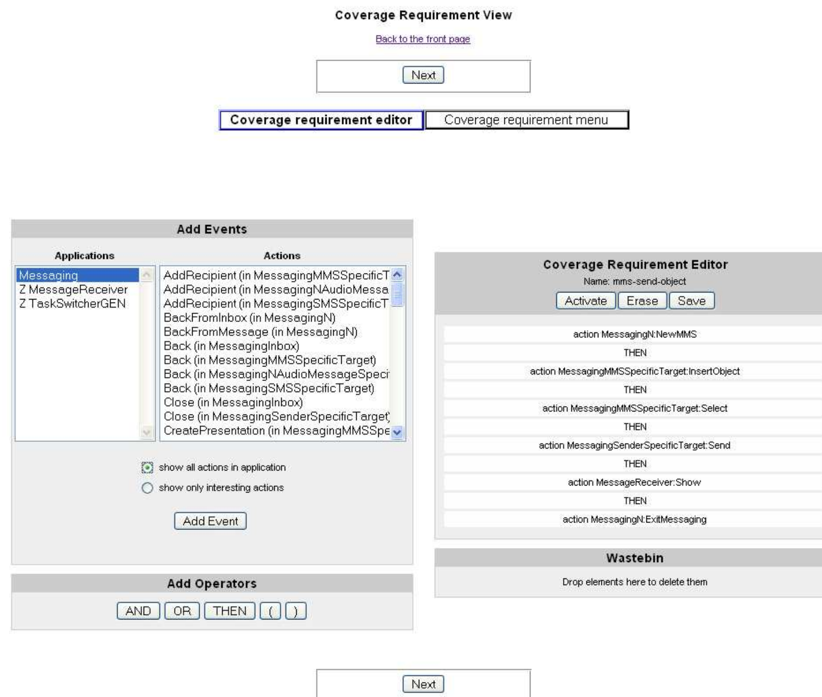

Fig. 3. Coverage requirement editor.

```
action Messaging2-Receiver:Show THEN (
  action Messaging1-Main:ExitMessaging AND
  action Messaging2-Main:ExitMessaging
)
```

In the example, Messaging1 is the SUT that should send the MMS and Messaging2 the one that should receive it. Once the message has been composed, sent, received and opened, both SUTs should return to the main menu in a non-specified order. The right hand side of Figure 3 shows the corresponding coverage requirement in the case of one SUT. In the one phone configuration, the sender and the receiver are the same device, while in the two phone configuration they are different. Replacing operator AND with OR would simply mean that either one of the phones should return to the main menu. If the requirement under construction if not well-formed, the requirement turns red and an error message is displayed. The coverage language is presented in more detail in [11].

Since constructing long coverage requirements can take some effort and time, there is a view where they can be saved and loaded (see Figure 4). Moreover, there is an option to upload and download coverage requirements if the tester wants to use another editor.

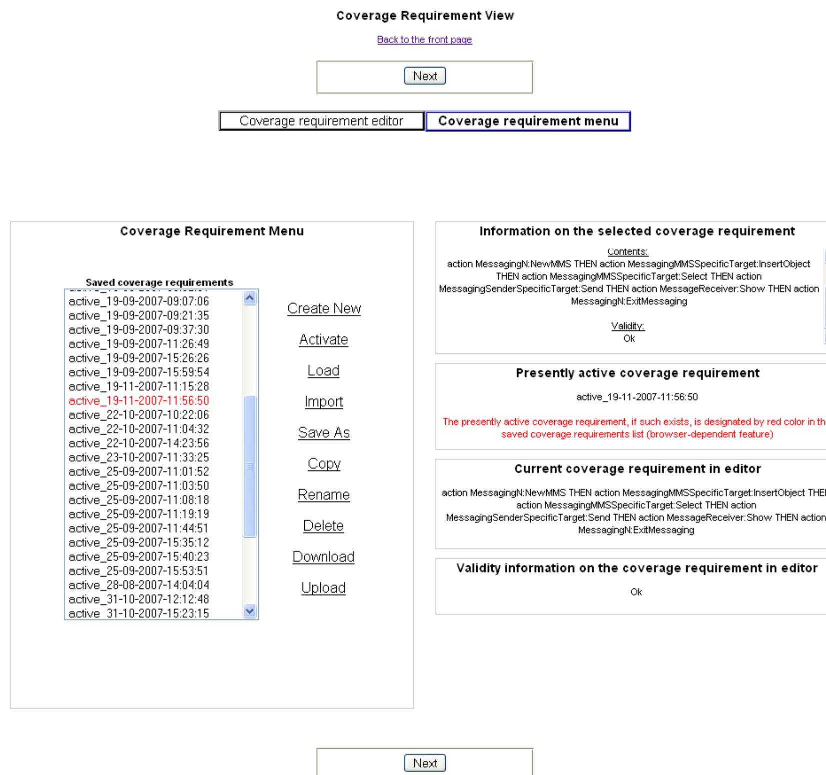

Fig. 4. Coverage requirement menu.

In the next view, the tester can set the parameters for the test session. First of all, there are different heuristics corresponding to the different testing modes. Moreover, there are some other parameters to be selected based on the heuristics used. For instance, using the game heuristics in the requirement coverage mode requires the depth of the search tree. There are naturally default values available, but based on the model complexity, better results, i.e. reaching the coverage requirement faster, can be achieved by carefully selecting the parameters. In addition to these, the tester can specify the seed for the random number generator.

Another important selection to be made in this view is the data and localization tables to be used in the test runs. For this purpose, the tester is presented with a list of predefined files in the server.

Finally, the tester can choose to start the test run in the next view. There is also a selection on how detailed a log is displayed during the test run. In any case, the tester can always choose to view all the logged information. The log is automatically saved so that the test run can be repeated for debugging purposes, for instance. When the test execution specialist presses the “Start” button, the server starts waiting for a

Fig. 5. Test setup with two SUTs.

connection from a client where the SUTs have been connected using Bluetooth or a USB connection. An example test setup with two targets is shown in Figure 5. On the right hand side the test log in the web GUI is shown. The client machine on the left hand side has two targets connected using a Bluetooth connection.

After the test session is finished, the web interface turns either green or red, based on success or failure. In the latter case, the tester may want to download the log for reporting or debugging. In the former case, the tester can report that the requirement in question has now been tested. The interested reader can view a video of the test session described in the above example at `http://www.cs.tut.fi/~teams`.

6 Discussion

In this paper we have described a model-based GUI testing service for Symbian S60 smartphone applications. The approach is based on a test server that is currently in the prototype stage. We are implementing the tools we have described and are releasing new versions under the MIT Open Source Licence. A download request can be made through the URL mentioned above.

In our solution, the server encapsulates the domain-specific test models and the associated test generation heuristics. The testers, or test execution specialists, order

tests from the server, and the test adapter clients connect to the phone targets under test. The main benefit of this approach compared to more generic approaches is that it should be easier to deploy in industrial environments; in practice, the tasks of the tester are minimized to specifying the coverage requirement as well as some parameters for heuristics, etc. We are developing the web interface to be as usable as possible and plan to conduct usability surveys in the future.

How then could the service model be used? The organization of testing services affects what kind of testing process could be used. This demands a flexible approach for ease of coordination [24]. In industrial practice, it would be important to get reliable service based on the current testing needs. This is in line with the current trends of the software industry [25]. At best, there would be several providers for the service to fulfill the needs of different end-users. Beside technical competence, communication skills are emphasized in order to provide transparency to the details of the solution.

Case studies on using the service concept are on the way. We have already used the web GUI internally for several months. In these experiments, the SUT has been the S60 Messaging application, including features such as short message service (SMS) and multimedia messages (MMS). The former supports sending only textual messages, while the latter supports attaching photos, video and audio clips. So far we have performed testing with configurations of one to two phones. Based on the positive results of this internal use, we are working towards transferring this technology to our industrial partners. One of the partners has already successfully tried out our test server in actual test runs without the web GUI. We anticipate that the web GUI will help us in conducting wider studies in the future.

Acknowledgements

This paper reports the ongoing results of research funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq Software, F-Secure, and Plenware, as well as the Academy of Finland (grant number 121012). For details, see `http://practise.cs.tut.fi/project.php?project=tema`.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
2. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing concurrent object-oriented systems with Spec Explorer. In: Proceedings of Formal Methods 2005. Number 3582 in Lecture Notes in Computer Science. Springer (2005) 542–547
3. Robinson, H.: Obstacles and opportunities for model-based testing in an industrial software environment. In: Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany (2003) 118–127
4. Hartman, A.: AGEDIS project final report. Available at `http://www.agedis.de/documents/FinalPublicReport%28D1.6%29.PDF` (2004) Cited March 2008.
5. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Satama, M.: Towards deploying model-based testing with a domain-specific modeling approach. In: Proceedings of TAIC PART – Testing: Academic & Industrial Conference, Windsor, UK, IEEE Computer Society (2006) 81–89

6. Symbian. (http://www.symbian.com/. Cited March 2008.)
7. S60. (http://www.s60.com. Cited March 2008.)
8. OMG: UML testing profile, v 1.0. (http://www.omg.org/technology/documents/formal/test_profile.htm. Cited March 2008.)
9. Kervinen, A., Maunumaa, M., Pääkkönen, T., Katara, M.: 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, Springer (2006) 16–31
10. Kervinen, A., Maunumaa, M., Katara, M.: 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., Vienna, Austria, Elsevier (2006) 53–66
11. Katara, M., Kervinen, A.: 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. Springer (2007) 219–234
12. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Jääskeläinen, A.: Can I have some model-based GUI tests please? Providing a model-based testing service through a web interface. In: Proceedings of the second annual Conference of the Association for Software Testing (CAST 2007), Bellevue, WA, USA (2007)
13. Jääskeläinen, A.: A domain-specific tool for creation and management of test models. Master's thesis, Tampere University of Technology (2008)
14. Boehm, B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison Wesley (2004)
15. Rook, P.: Controlling software projects. Softw. Eng. J. **1** (1986) 7–16
16. Buwalda, H.: Action figures. STQE Magazine, March/April 2003 (2003) 42–47
17. Fewster, M., Graham, D.: Software Test Automation: Effective use of test execution tools. Addison–Wesley (1999)
18. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, London, UK, Springer-Verlag (1996) 429–528
19. Python: Python Programming Language homepage. (http://python.org/. Cited March 2008.)
20. Karsisto, K.: A new parallel composition operator for verification tools. Doctoral dissertation, Tampere University of Technology (number 420 in publications) (2003)
21. Satama, M.: Event capturing tool for model-based GUI test automation. Master's thesis, Tampere University of Technology (2006) Available at http://practise.cs.tut.fi/project.php?project=tema&page=publications. Cited March 2008.
22. HP: Mercury Functional Testing homepage. (http://www.mercury.com/us/products/quality-center/functional-testing/. Cited March 2008.)
23. Zakas, N.C., McPeak, J., Fawcett, J.: Professional Ajax. 2nd edn. Wiley (2007)
24. Taipale, O., Smolander, K.: Improving software testing by observing practice. In: ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering, New York, NY, USA, ACM Press (2006) 262–271
25. Microsoft: Microsoft unveils vision and road map to simplify SOA, bridge software plus services, and take composite applications mainstream. (2007-11-28) Available at http://www.microsoft.com/presspass/press/2007/oct07/10-30OsloPR.mspx. Cited March 2008.