

# Towards Deploying Model-Based Testing with a Domain-Specific Modeling Approach

Mika Katara, Antti Kervinen,  
Mika Maunumaa  
Tampere Univ. of Technology  
Institute of Software Systems  
P.O.Box 553, 33101 Tampere  
FINLAND  
firstname.lastname@tut.fi

Tuula Pääkkönen  
Nokia Technology Platforms  
P.O.Box 68  
FI-33721 Tampere, FINLAND

Mikko Satama  
Tampere Univ. of Technology  
Institute of Software Systems  
P.O.Box 553, 33101 Tampere  
FINLAND  
firstname.lastname@tut.fi

## Abstract

*Model-based testing automating the generation of test cases is technically superior to conventional scripted testing. However, there have been difficulties in deploying the methodology in large scale industrial context. In this paper we present a domain-specific approach to model-based GUI testing that should be easier to adopt than more generic solutions. The method is based on keywords and action words that are considered as best practices in conventional GUI test automation. The basic idea is to record GUI events just like in capture/replay tools, but instead of producing scripts that can be hard to maintain, we produce sequences of keywords. These sequences are further transformed semi-automatically into Labeled Transition Systems where action words are used as transition labels. The action words model user behavior at a high level of abstraction while the keywords correspond to the GUI navigation. We also describe the associated tool set that we are developing and an example of using the approach.*

## 1 Introduction

Deploying formal testing methods in industrial context involves facing obstacles that are not always technical in nature [9]. On the one hand, there are managerial problems such as getting the right people to sponsor the prototyping and piloting with new methods. On the other hand, there are issues involved with the people actually using the methods. These include how to make the tools associated with the methods as easy to use as possible and how to reorganize work if the tools automate a previously manual task, for instance.

In this paper, we concentrate on the latter issues in the

context of model-based system testing. Model-based testing can be seen as a part of model-driven development (MDD) that tries to raise the level of abstraction in software development and generate code (as well as tests) automatically from structural and behavioral models. A promising approach to MDD is to use domain-specific modeling languages that can be designed exclusively for the problem domain at hand [3]. However, choosing between different types of test modeling languages involves many trade-offs. In our case of a product family setting and system testers not necessarily fluent in any generic modeling language, for instance UML, a domain-specific approach is seen as a good option to facilitate deployment.

Our testing targets applications running on Symbian OS [10], which is an operating system for mobile devices such as smart phones. Although there is hardly any safety critical functionality included, the quality of the consumer products needs to be extremely high for obvious reasons. Model-based testing seems like a promising approach for system testing in this domain, since it not only automates the laborious development of test scripts but also carries the promise of finding defects effectively in the fast pace of product family development. To support testing of product families, the test models are constructed so that only minimal changes are needed when the system under test (SUT) is superseded by another product of the same family. Such maintenance activities should be much easier to accomplish using test models than traditional scripts.

From the point of view of the test personnel, even with a domain-specific approach, model-based testing brings new definitions to the commonly known testing roles. The roles of the test personnel change and there is more demand to specialist knowledge than before. While the basic purpose of testing stays the same, i.e. the target is to find defects, we should find the defects earlier in the product life-cycle.

In our previous work we have developed a prototype of a model-based test generator that tests Symbian OS applications on-line through a graphical user interface (GUI) of a smart phone [7]. The test automation architecture includes a commercial GUI test automation system that we have extended with model-based capabilities. To support test modeling in a GUI context, we have also defined a so-called 3-tier test model architecture [6]. However, we still need to address the issue on how an average tester could construct a test model capable of finding defects effectively.

We believe the solution lies in the domain-specific approach to modeling. By developing a domain-specific test modeling language for Symbian GUI testing, we anticipate that the introduction of the model-based tools will be eased. Ideally, a test model could be obtained from a design model by sophisticated model transformations. However, such tools are not available today and that would also rely on the availability of the design model. Another approach would be to design test models by hand from scratch, using some visual modeling tool, for instance.

Instead, we envision that we could ease test modeling by capturing GUI events directly. This resembles capture/replay approach advocated by conventional GUI test automation tools. However, in contrast to producing scripts that can be hard to maintain, we produce sequences of so called keywords, which are further developed into higher level action words corresponding to our 3-tier model architecture.

The contributions of this paper are in describing how to define a domain-specific test modeling language using event capturing in the context of the 3-tier test model architecture. Moreover, towards deployment, we discuss the anticipated effects of our approach to test personnel. The rest of the paper is structured as follows: In Section 2 the background of the contributions is discussed. Sections 3 and 4 present the domain-specific test modeling methodology and its anticipated effects to the test organization, respectively. An example of the approach is given in Section 5. Finally, Section 6 draws some conclusions.

## 2 Background

In this section we present the background of our research. First, we discuss the keywords and action words that are concepts originating from conventional GUI test automation. Then the 3-tier model architecture based on keywords and action words is illustrated. Moreover, we describe the context of our test tool development in the Symbian OS environment.

### 2.1 Keywords and action words

Buwalda recommends in the description of *action-based testing* [2] that testers having the necessary knowledge of the problem domain should focus on high-level concepts while designing the tests. In practice, this means modeling business processes and picking interesting sequences of events for discovering defects. Buwalda calls these high-level events *action words*; other authors have also come up with similar approaches. Possibly a different person, fluent in the test automation tools, then implements the action words with *keywords* that form a concrete implementation layer of the test automation system.

An example of a keyword from the Symbian test environment is `kwPressKey`, corresponding to a keystroke. Such a keyword could be used, for instance, in a sequence that models starting the calculator application. The sequence would correspond to a single action word, for instance `awStartCalculator`. Action words represent abstract operations such as “make a phone call” or “open Calculator”. Their implementation could consist of sequences of keywords with related parameters as test data. However, one should note that the difference between keywords and action words is somewhat in the eye of the beholder. The most generic keywords can be considered as action words in the functional sense. The main difference between the two concepts is in the purpose of use.

### 2.2 3-tier test model architecture

Our approach to domain-specific test modeling using keywords and action words is build around a 3-tier model architecture, consisting of *control*, *action* and *refinement machines* each in its own tier (see Figure 1). The machines are LTSs (Labeled Transition Systems) that model the behavior of the user of the SUT. Next, the three tiers are discussed in detail. For more detailed discussion and formal definitions the reader is referred to [6].

#### 2.2.1 Keyword tier

The executable test model is obtained using the two lowest tiers in the model architecture, i.e., action machines and refinement machines. The purpose of the refinement machines is to refine the action words in action machines to sequences of executable events in the GUI of the SUT.

In a product family context reusability is paramount. To support testing of different products of the same product family, we must separate the functionality from the GUI events. This allows us to reuse action machines with SUTs that support the same operations but with different GUI. For example, two camera applications can have exactly the same functionality in two devices one having regular keyboard and the other a touch screen. However, designing an



which is the initial state. When the machine wakes up for the first time, the Camera application is started. It is also verified that the application has started indeed. After that, the test guidance algorithm makes a choice between three possible actions: a photo can be taken, the Camera application quitted, or the Camera application can be put to the background by entering to a sleep state.

There are two communication mechanisms between action machines at the action tier. The first one is for controlling which action machine is currently running. It uses primitives  $Sleep_{TS}$ ,  $Wake_{TS}$ ,  $Sleep_{App}$  and  $Wake_{App}$ . The primitives represent putting to sleep and waking up an application with a task switching application ( $TS$ ) or directly within another application ( $App$ ).

When an action machine goes to a sleep state by executing the  $Sleep_{TS}$ , it synchronously wakes up the task switching action machine. After some steps, the machine executes the  $Wake_{TS}$  action synchronously with another action machine. The other machine then wakes up and the task switching machine enters a sleep state. In some cases it is handy to bypass the task switching. A running action machine  $\mathcal{A}$  can put itself into sleep and synchronously wake up another machine  $\mathcal{B}$  by executing action  $Sleep_{App}<\mathcal{B}>$  (the woken action machine executes  $Wake_{App}<\mathcal{A}>$ ). In the latter case the task switching action machine does not wake up at any point.

The  $Sleep_{TS}$ - $Wake_{TS}$  and  $Sleep_{App}$ - $Wake_{App}$  primitives have been inspired by the two possible ways the user can activate an application in Symbian. The former corresponds to the situation where a task switching application, modeled by the task switching action machine, is used for activating some application running in the background. The latter is used for modeling the situation where the user activates a specific application directly within another application. For instance, it is possible to activate the Gallery application by choosing “Go to Gallery” from the menu of the Camera application.

We use the other communication mechanism for exchanging information on shared resources between action machines. For that we use primitives for requesting ( $Req$ ) and giving permissions ( $Allow$ ).  $Req$  can be executed in running states and  $Allow$  in sleeping states; the primitives cannot wake up a sleeping action machine or put a running one into sleep.

### 2.2.3 Test control tier

The highest level tier, test control, is used for defining which test models to use and what kind of tests to run in which order. For instance, it could be specified that a test control machine should first run three very short smoke tests with three different test models. Each model could be built from a single action machine composed with the corresponding

refinement machines. The action machines could test for example the Camera application, the Messaging application, and the Telephony application.

After a smoke test, a longer test could be run covering all the elements in some use case such as “take a photo, send it and receive a phone call”. This time the model would be composed of all the LTSs as well as the task switcher LTSs. Finally, we could start a possibly never-ending robustness test. The coverage data obtained in the previous test could be used to avoid retesting the same paths again.

Execution of a transition in a test control machine consists of setting up a new test, running it and handling the coverage data. All the information concerning test setup, coverage objectives and test guidance can be encoded in the transition labels called *control words*.

To set up a test, a control word determines which test model should be used in the test run and what kind of initial coverage data should be used. It is possible to start the run with a coverage data based on the execution logs of some previous test runs with the model. To run the test, coverage objectives and guidance heuristics are defined similarly. Coverage objectives are stated in the coverage language defined in [6]. Coverage objective is a Boolean function whose domain is the coverage data, so the coverage requirement for the whole run is obtained by combining individual objectives with logical “and” and “or” operators.

## 2.3 The tool vision

In order to introduce model-based testing practices for testing of Symbian OS applications, we are currently developing tools based on our experiences with a previous prototype implementation [7].

The architecture of the tool set is depicted in Figure 4. The architecture consists of a tool adaptation part and a model execution part. The former provides a high-level interface through which the latter can execute keywords and inspect the results of the executions.

The tool adaptation part is located inside a GUI testing tool for MS Windows, Mercury’s QuickTest Pro (QTP) [8]. QTP is capable of capturing information about window resources (buttons, text fields etc.) and providing access to those through an API. It also enables scripting test procedures using VBScript and recording a test log when requested, among other things.

To access the SUT we use m-Test [5] that provides access to the GUI of the SUT as well as to some internal information, such as the list of running processes. It displays the interactive user interface of a Symbian device (keyboard and display) as an ordinary application window in MS Windows accessible from QTP. A keyword library, which is implemented with VBScript of the QTP, converts each keyword to a sequence of window events for the m-Test.

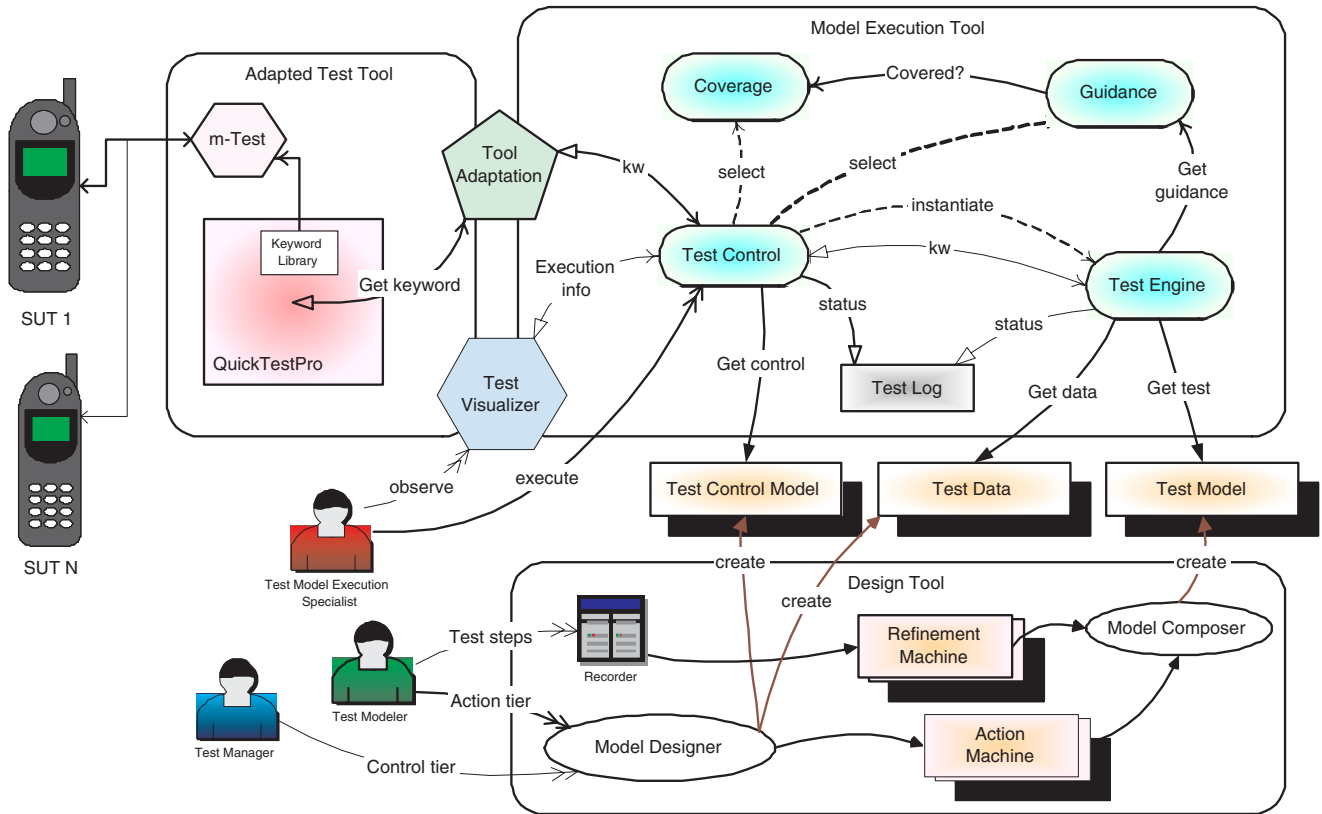


Figure 4. Test tool architecture.

The model execution part is an independent component possibly running on another computer. When a test run is initiated, a *Test Control* object encapsulating a test control machine is instantiated. Moreover, *Test Control* creates test *Guidance* and *Coverage* objects to guide the execution. Executing a transition in the test control machine corresponds to setting up and running a test.

When a new test run is set up, a *Test Model* object is instantiated. In addition, new test *Guidance* and *Coverage* objects are created to guide the run. It is possible to load the coverage data of a previous test run to the *Coverage* object. The data can be used for avoiding retesting the same behavior or for repeating the same test. *Test Control* initializes a test run by passing those three objects to *Test Engine*. During the test run, *Test Engine* asks *Guidance* which action should be executed next. *Guidance* uses *Coverage* to find out how executions of different events would affect the coverage objective.

Information concerning loaded test models, coverage objectives, executed transitions, results of the executions etc. is passed to the *Test Log* component. Its contents are visualized by *Test Visualizer*, which can be used both in on-line and off-line modes to see how the current test is ad-

vancing and to help debugging after a run, respectively.

Finally, *Model Designer* is a visual tool for designing control and action machines. An executable test model is built using test *Model Composer*. Because of the automated synchronization, it is enough that the user selects the action and refinement machines to be composed.

### 3 Developing domain-specific test models

In action-based testing, action words corresponding to the concepts of the problem domain define the domain-specific language for this domain. This section presents our contribution to domain-specific test modeling using the 3-tier model architecture. First, we introduce the bottom-up approach and then the top-down approach. Finally, we discuss the roles of the testing personnel.

#### 3.1 Bottom-up approach

The bottom-up approach can be used for building test models for the product family. In this approach, the test model can be built from scratch using an event capturing tool called *Recorder* (see Figure 4). The tool is capable of

capturing GUI events much in the same way as conventional capture/replay GUI test tools. However, instead of producing hard-to-maintain test scripts, our tool produces a list of keywords corresponding to the sequence of GUI events.

Test modeling begins by starting *Recorder* and recording a test using the GUI provided by m-Test. After each event, the tool outputs the corresponding keyword at the end of the keyword sequence. The user can also input free form text corresponding to names of states in between the keywords. This way we can equate points in the sequence that correspond to identical states of the SUT, allowing us to introduce loops to the test models later. Without branching and looping tests generated from models would be similar to linear and static test scripts limited in their ability to find defects.

During the recording, we need to add verification events. On the one hand, we could add verification keywords after each input event, but this might slow the test execution needlessly. On the other hand, adding a verification keyword only to the end of the sequence might lead to problems in debugging when something fails. So, there is a trade-off involved that needs to be resolved in a case-by-case fashion.

After recording a test, we split the sequence into action words. This is done by the *Model Designer* tool (see Figure 4) that reads in the generated keyword sequence and visualizes it as a LTS. In the LTS, the states with equal names are equated to introduce loops. If there are states that serve as starting and end points of more than one loop and the loops correspond to the same sequences of events, the tool asks the user whether the redundant ones should be deleted.

To assemble an action word model, the tool searches the keyword sequences for subsequences that have been archived. If such a subsequence is found, the tool suggests replacing that with the corresponding action word. To archive a new subsequence, the user selects the corresponding keywords and inputs the name of the corresponding action word. Naturally, the subsequences should not contain branching. This process is continued until there are only action words in the model. Another complementary way to assemble action machines is to recognize the names of the start and end states of keyword sequences. If the names match with the ones in the archive, again, the tool suggests to replace the sequence by the corresponding action word.

The action word model is finished by adding sleeping states where ever the execution is allowed to be interleaved with another action machines. Also Allow and Req transitions for resource sharing can be added at this point. *Recorder* could also be used for this, but we think that is more straightforward and intuitive to do it with the *Model Designer* tool.

### 3.2 Top-down approach

The top-down approach can be used for maintaining the test models. This approach starts from an action word model. In a general case, there might be some action words that do not yet have a corresponding keyword sequence implementing them. Alternatively, an implementation of some action word may need to be changed if, for instance, a new type of keyboard has been introduced to the next product in the family. In both situations *Recorder* can be used to record the keyword sequences. *Model Designer* is used for selecting the action word whose implementation needs to be defined. After selecting such an action word, *Recorder* is started to record the sequence the user inputs with the m-Test GUI. The user can also choose to override an existing keyword sequence by recording a new one.

The test modeling can also be started by drawing an LTS of the action word model from scratch while there are not yet any keyword models defined. In this case, the keyword models are built one-by-one using *Recorder* as described above.

## 4 Adapting an organization

Typical testing organization consists of three to four different role categories. Firstly, there is the Test Manager, who takes care of test planning, i.e., schedules, test strategy and basic follow-up as well as reporting to other stakeholders. The second party is the Test Engineer who takes care of the test specification. In traditional testing, this task includes specifying individual test cases consisting of the test objective, initial state, input sequence and expected outcome [1]. The Test Engineer also takes care of executing the test case and recording the results. The third role category includes various Test Specialist roles, such as for test automation, test environments, performance testing etc. Naturally, the roles described are not mutually exclusive, and the job descriptions can actually have a mix of tasks of various roles. Depending on the goals and the organizational structure, the mix can vary.

The defects found in model-based testing can be more complex than in the case of traditional test, so describing and reporting the error situation requires more details. This holds especially in concurrency related cases. However, model-based testing as a core technology will bring some changes to all the above roles.

When starting to deploy model-based testing, the roles should not be changed at once. The focus should not be in redefining the roles but to find new angles to old ones to utilize the best capabilities in the new context. In any case, not all testing will be model-based so both the conventional and new skills will be needed.

Creation of the new role definitions, job descriptions and skill analysis should happen interactively with the testing personnel enabling job descriptions to be accurate. This will also support test professionals in their competence development, when they can focus on skills that they see are needed from the model-based testing point of view. The more freedom individuals can be given to defining their roles, the less change resistance should appear for the technology in general. Obviously, job descriptions should define not only what is included but also what is excluded.

One of the most prominent skills that model-based testing will require from testing personnel, is the information analysis and structuring. Regardless of whether or not design models can be used as input for test modeling, the Test Engineer needs to understand test models deeply. Test model understanding is crucial, since it corresponds to the mental model of the behavior of the SUT [4] and has elements of the test plan as well as test specification.

In the beginning, the test modeling skills might be a new test specialist area altogether. The models obtained from design and architecture are not directly adaptable to testing as such. Thus, experts are needed to go through current documentation of the product to create proper test models. Understanding the test models generated by others can be the first step. Next, a person should develop skills to modify and create new models.

Besides modeling, the understanding of the underlying product architecture needs to deepen. It might be a challenge to pin-point the cause of the defect as accurately as possible, since it requires some knowledge of the test tools. One needs to be able to grasp from test model execution logs what has been executed and what might have been causing the failure and why.

We anticipate that the testing personnel in deploying our approach consist of the following roles (see Figure 4): Test Manager, Test Modeler, and Test Model Execution Specialist, which can be compared to the Test Engineer role. Test Manager role stays mostly the same, since someone should define the entry and exit criteria to the test model execution, coverage criteria and define which metrics are gathered. In the 3-tier model architecture, these tasks are related to the test control tier. Test Manager also should focus in communicating the testing technology aspects. This includes explaining how model-based testing compares to the conventional testing methods and advocating reasons for and against using it towards management and testing personnel as in any new process initiation.

Test Modeler is basically a new role or a new variation of the Test Specialist role. The main goal of the Test Modeler is to create test models using the *Model Designer* and *Recorder* tools. This can be done based on product specifications and using the bottom-up and top-down approaches in accordance with the chosen test strategy. The Test Mod-

eler can also be responsible of designing the execution of the model and setting up the environment accordingly.

Test Model Execution Specialist follows up the test execution across the test model to ensure that the model is used according to the agreed principles and test data. The main focus of this role is in reporting the results and faults onwards. The purpose is to document the test model usage and testware architecture in a way that enables its reuse.

The different roles need to work together to form a tight chain in order to find the most crucial defects as early as possible. The people behind the roles should rely on each other's input even more, since test modeling renders the tested behavior in more detail than in conventional test automation. Test documentation is also needed, but not necessarily in the current extent. It is needed to explain the test strategy instead of very detailed step-by-step instructions. This should speed up planning and make the plans more reusable as well.

## 5 Example

An imaginary example of using the approach is presented next. The development organization has heard that the testing organization is piloting with our approach. To challenge the new testing methodology, they provide the testing organization with the following use case to be tested: Alice sends Bob an SMS asking him to join for a lunch. Bob replies by sending a multimedia message containing a picture of a bumper taken at a traffic jam and a text "I will call you when I'll get there". After a while, Bob calls to Alice, but she is on the phone with Carol so the line is busy. After a few minutes Alice calls back to Bob and they agree to meet at the cafeteria.

The testers start to work on the challenge immediately. The testing organization is composed of three persons: Jack, the Test Modeler, Margaret, the Test Manager, and William, the Test Model Execution Specialist. First, Jack sets up a test environment with three phones, i.e., one for Alice, one for Bob and one for Carol. The three phones are connected to a Test Machine running m-Test and QTP using Bluetooth wireless link (see Figure 4).

Because this is a pilot project, there are no archived action machines for all the necessary parts of the use case. Assuming that the testing organization has already tested with action words corresponding to taking photos and sending them as MMS (see Figures 2 and 3), they need to develop action machines for sending and receiving SMS as well as making and answering calls. Sending and receiving SMS is very similar at the action machine level to sending and receiving MMS, so Jack uses the *Model Designer* tool in a copy-and-paste fashion to development action machines for SMS. However, this functionality is implemented with somewhat different sequences of keystrokes. Only the key-

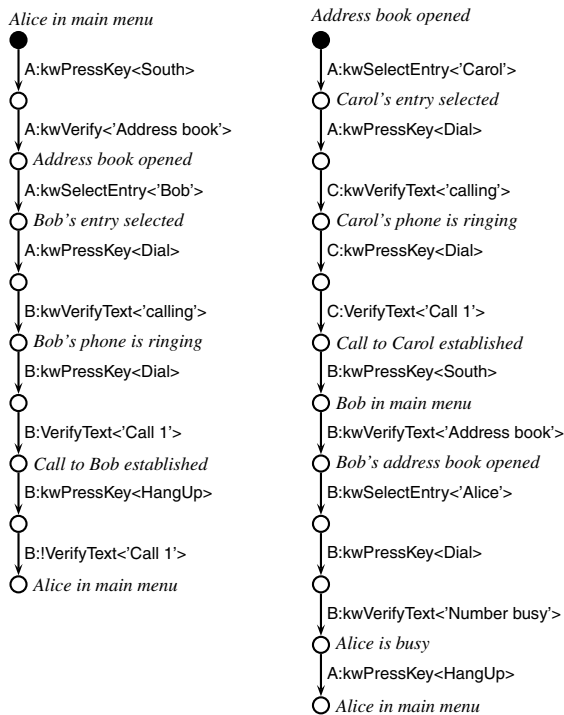


Figure 5. Bottom-up modeling of calls.

word sequence (and the corresponding action word) used for choosing the receiver of the message from the list of contacts is the same. To create the needed refinement machines, he uses the top-down approach with *Recorder* for sending a SMS by inputting the message to be send by Alice using m-Test.

On the other hand, the action machines for making and answering calls are developed using the bottom-up approach. For this purpose Jack again starts the event capturing and makes a call from Alice's phone to Bob's phone using m-Test. After answering the call, he hangs up the Bob's phone (successful scenario). Next, he starts another call from Alice's phone to Carol's phone before making a call from Bob's phone to Alice's phone. This way he records the keyword sequence for the call that is not established because the callee is busy (concurrency case). He chooses to use verification events each time before pressing the call button and at the end of the sequences. For readability and for the next phase in modeling, Jack adds comments to some states, especially when it is reasonable to branch the test execution in a state. Using the same comment for the same states helps in the next phase.

The resulting keyword sequences (see Figure 5) are loaded into the *Model Designer* tool where new action words for making and answering a call are defined. The number to be called is searched from the list of contacts and not inputted using the keyboard. Since similar scheme was

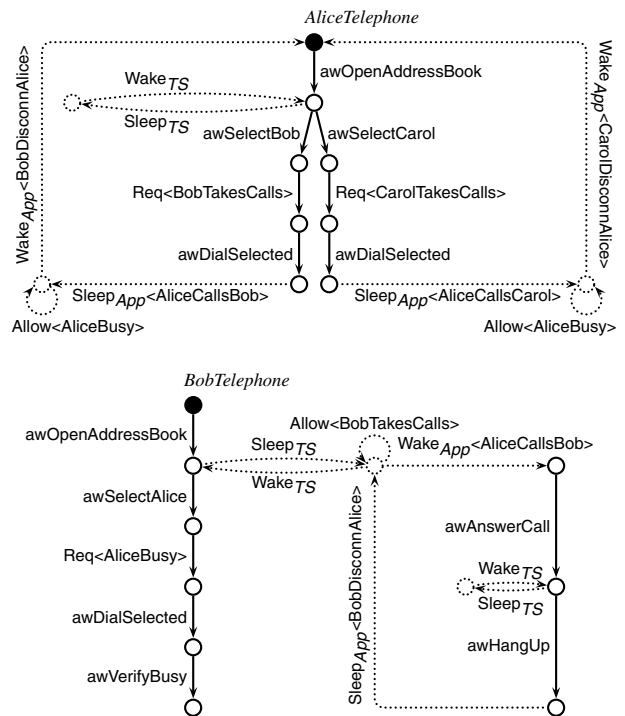


Figure 6. Two Telephone action machines.

used when sending MMS or SMS, the corresponding action word may be reused in a new context of making a call. Alternatively, the number could be picked up from the list of answered (or unanswered) calls or speed dialed. However, Jack decides to record these keyword sequences later.

In the Model Designer, Jack designs the skeletons of Telephone action machines, separately for Alice, Bob and Carol. First, he connects the commented states in Figure 5 with action words. This results roughly in the solid-line parts of the action machines in Figure 6. Second, Jack adds the sleeping states to the action machines (the dotted parts in Figure 6). At this phase he decides the states in which the action machines accept calls. He also adds *Req<...>* transitions to ensure that action machines are called only when they are ready for it. Thirdly, the action word transitions which reverse from existing states to previous ones are added. For example, a transition labeled *awExitAddressBook* is drawn from the target state of the *awSelectCarol* transition to the initial state of the *AliceTelephone* action machine. To keep Figure 6 readable, these transitions are not shown. Finally, a new sleeping state is added for every action machine to be the initial state. As discussed in Section 2.2.2, every action machine is initially in a sleeping state.

After assembling the action machines, Margaret quickly develops a simple control machine that ensures that this particular use case is tested in the test run. Since the de-

velopment organization has been using this use case to develop the applications, no failures are detected in a test run. However, when William executes the test model with certain heuristic that covers much more functionality and other possible interleavings than the simple use case, a failure is noticed after some hours. If Alice is calling Carol while Bob's MMS arrives the call is hang-up for some reason.

The new action words developed for this use case are archived for further use. When William reports the bug to the development organization they realize that the testers might now have a more powerful weapon to look for their defects. Moreover, they can now get more detailed steps to reproduce the failures found by testing, when there are concurrent or other steps that need lots of repetition.

## 6 Discussion

In this paper we have described a domain-specific solution for developing test models for the Symbian OS applications to be tested through a GUI. The approach has been designed for testing software product families in a way that eases the burden of maintenance when the SUT changes to another product of the same family.

Our approach is generic in the sense that it can be used for developing domain-specific languages for model-based GUI testing with action words and keyword also in other environments. This can be done with the help of an event capturing tool (in our case *Recorder*) converting GUI events to sequences of keywords and a modeling tool (*Model Designer*) used for transforming subsequences of keywords into action words. The keyword sequences are recorded using the event capturing tool and post-processed using the modeling tool. The post processing includes defining action words that correspond to concepts of the problem domain rather than the solution domain. The domain-specific language consisting of the action words can then be used for test modeling in that domain.

Concerning implementation of the tool set, we are currently in a process of developing *Recorder*, which should be a rather straightforward task. We also need to develop a plug-in for some visual modeling tool for developing action word models (*Model Designer*). The choice of the tool is seen as a difficult task since it affects the test personnel using our approach. Thus, it is a careful choice involving usability issues among other things. However, this remains as future work as well as conducting industrial case studies to validate the approach.

## References

- [1] British Computer Society. ISEB foundation certificate in software testing material, 2001.
- [2] H. Buwalda. Action figures. *STQE Magazine, March/April 2003*, pages 42–47, 2003.
- [3] Domain-Specific Modelling Forum. DSM case studies and examples. Available at <http://www.dsmforum.org/cases.html>, 2006.
- [4] I. K. El-Far. Enjoying the perks of model-based testing. In *Proceedings of the Software Testing, Analysis, and Review Conference (STARWEST) 2001*, 2001.
- [5] Intuwave. m-Test homepage. At URL <http://www.intuwave.com>.
- [6] 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)*, Vienna, Austria, Mar. 2006. To appear.
- [7] 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)*, July 2005. To appear in LNCS 3997.
- [8] Mercury Interactive. QuickTest Pro homepage. At URL <http://www.mercury.com>.
- [9] 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>.
- [10] Symbian Ltd. Symbian Operating System homepage. At URL <http://www.symbian.com/>, 2006.