

Model-Based Testing Through a GUI

Antti Kervinen¹, Mika Maunumaa¹, Tuula Pääkkönen², and Mika Katara¹

¹ Tampere University of Technology, Institute of Software Systems
P.O.Box 553, FI-33101 Tampere, FINLAND

{firstname.lastname}@tut.fi

² Nokia Technology Platforms
P.O.Box 68, FI-33721 Tampere, FINLAND

Abstract. So far, model-based testing approaches have mostly been used in testing through various kinds of APIs. In practice, however, testing through a GUI is another equally important application area, which introduces new challenges. In this paper, we introduce a new methodology for model-based GUI testing. This includes using Labeled Transition Systems (LTSs) in conjunction with action word and keyword techniques for test modeling. We have also conducted an industrial case study where we tested a mobile device and were able to find previously unreported defects. The test environment included a standard MS Windows GUI testing tool as well as components implementing our approach. Assessment of the results from an industrial point of view suggests directions for future development.

1 Introduction

System testing through a GUI can be considered as one of the most challenging types of testing. It is often done by a separate testing team of domain experts that can validate that the clients' requirements have been fulfilled. However, the domain experts often lack programming skills and require easy-to-use tools to support their work. Compared to application programming interface (API) testing, GUI testing is made more complex by the various user interface issues that need to be dealt with. Such issues include input of user commands and interpretation of the output results, for instance, using text recognition in some cases.

Developers are often reluctant to implement system level APIs only for the purposes of testing. Moreover, general-purpose testing tools need to be adapted to use such APIs. In contrast, a GUI is often available and there are several general-purpose GUI testing tools, which can be easily taken into use. Among the test automation community, however, GUI testing tools are not considered an optimal solution. This is largely due to bad experiences in using so-called capture/replay tools that capture key presses, as well as mouse movement, and replay those in regression tests. The bad experiences are mostly involved with high maintenance costs associated with such a tool [1]. The GUI is often the most volatile part of the system and possible changes to it affect the GUI test automation scripts. In the worst case, the selected capture/replay tool uses bitmap comparisons to verify the results of the test runs. False negative results can then be obtained from minor changes in the look and feel of the system. In practice, such test automation needs maintenance whenever the GUI is changed.

The state of the art in GUI testing is represented by so-called *keyword* and *action word* techniques [2, 3]. They help in maintenance problems by providing a clear separation of concerns between business logic and the GUI navigation needed to implement the logic. Keywords correspond to key presses and menu navigation, such as “click the OK button”, while action words describe user events at a higher level of abstraction. For instance, a single action word can be defined to open a selected file whose name can be given as a parameter. The idea is that domain experts can design the test cases easily using action words even before the system implementation has been started. Test automation engineers then define the keywords that implement the action words using the scripting language provided by the GUI automation tool.

Although some tools use smarter comparison techniques than pure bitmaps, and provide advanced test design concepts, such as keywords and action words, the maintenance costs can still be significant. Moreover, such tools seldom find new bugs and return the investment only when the same test suites are run several times, such as in regression testing. The basic problem is in the static and linear nature of the test cases. Even if only 10% of the test cases would need to be updated whenever the system under test changes, this can mean modifying one hundred test cases from the test suite of one thousand regression tests.

Our goal is to improve the status of GUI testing with model-based techniques. Firstly, by using test models to generate test runs, we will not run into difficulties with maintaining large test suites. Secondly, we have better chances of finding previously undetected defects, since we are able to vary the order of events. Towards these ends, we propose a test automation approach based on Labeled Transition Systems (LTSs) as well as action words and keywords. The idea is to describe a test model as a LTS whose transitions correspond to action words. This should be made as easy as possible for also testers with no programming skills. The maintenance effort should localize to a single model or few component models. The *action machines* we introduce are composed in parallel with *refinement machines* mapping the action words to sequences of keywords. The resulting composite LTS is then read into a general-purpose GUI testing tool that interprets the keywords and walks through the model using some heuristics. The tool also verifies the test results and handles the reporting.

The contributions of this paper are in formalizing the above scheme, introducing novel test model architecture and applying the approach in an industrial case study. Finally, we have assessed the results from an industrial point of view. The rest of the paper is structured as follows. Sections 2 and 3 describe our approach in detail as well as the case study we have conducted. The assessment of the results is given in Section 4. Related work is discussed in Section 5 and conclusions drawn in Section 6.

2 Building a Test Model Architecture

In the following, we will develop a layered test model architecture for testing several concurrently running applications through a GUI. The basis for layering is in keyword and action word techniques, and therefore we will first introduce how to adapt these concepts to model-based testing.

As a running example, we will use testing of Symbian applications. Symbian [4] is an operating system targeted for mobile devices such as smartphones and PDAs. The variety of features available resembles testing of PC applications, but there are also characteristics of embedded systems. For instance, there is no access to the resources of the GUI. In the following, the term system under test (SUT) will be used to refer to a device running Symbian OS.

2.1 Action Words and Keywords

As Buwalda [3] recommends in the description of *action-based testing*, test designers should focus on high-level concepts in test design. This means modeling business processes and picking interesting sequences of events for discovering possible errors. These high-level events are called action words. The test automation engineer then implements the action words with keywords, which act as a concrete implementation layer of test automation.

An example of a keyword from our Symbian test environment is `kwPressKey` modeling a key press. The keyword could be used, for instance, in a sequence that models starting a calculator application. Such a sequence would correspond to a single action word, say `awStartCalculator`. Thus, action words represent abstract operations like “make a phone call”, “open Calculator” etc. Implementation of action words can consist of sequences of keywords with related parameters as test data. However, the difference between keywords and action words is somewhat in the eye of the beholder. The most generic keywords can almost be considered as action words in the sense of functionality; the main difference is in the purpose of use and the level of abstraction.

Our focus is on the state machine side of the action-based testing. We do not consider decision tables, which are recommended as one alternative for handling test combinations [3]. However, there have been some industrial implementations using spreadsheets to describe keyword combinations to run test cases, and they have proven quite useful. Experiences also suggest that the keywords need to be well described and agreed upon jointly, so that the same test cases can be shared throughout an organization.

2.2 Test Model, Action Machines and Refinement Machines

We use the TVT verification toolset [5] to create test models. With the tools, the most natural way to express the behavior of a SUT is an LTS. We use two tools in the toolset: a graphical editor for drawing LTSs, and a parallel composition tool for calculating the behavior of a system where its input LTSs run in parallel. We will compose our *test model* LTS from small, hand-drawn LTSs with the parallel composition tool. The test model specifies a part of the externally observable behavior of the SUT. At most that part will be tested when the model is used in test runs.

In our test model architecture, hand-drawn LTSs are divided in two classes. *Action machines* are the model-based equivalent for test cases expressed with action words, whereas *refinement machines* give executable contents to action words, that is, refinement from action words to keywords. In the following we formalize these concepts.

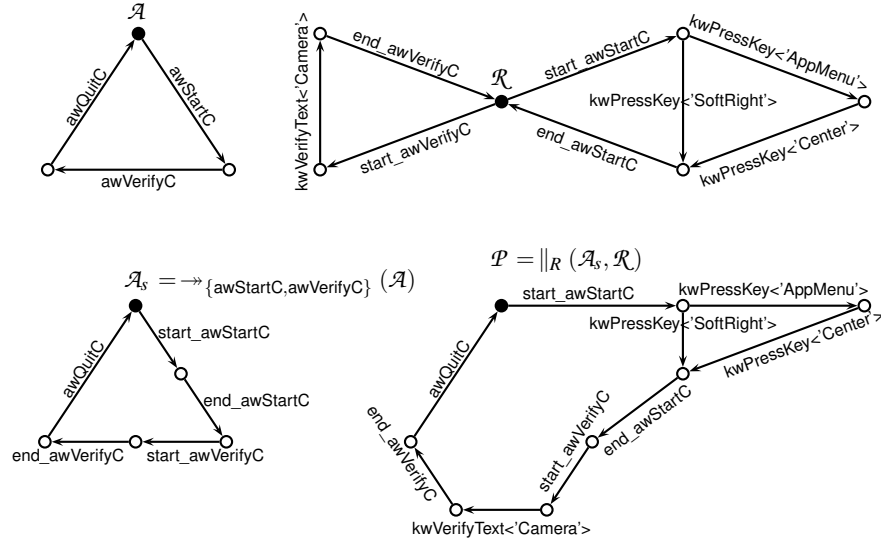


Fig. 1. Transition splitter and parallel composition

Definition 1 (LTS). A labeled transition system, abbreviated LTS, is defined as a quadruple $(S, \Sigma, \Delta, \hat{s})$ where S denotes a set of states, Σ is a set of actions (alphabet), $\Delta \subseteq S \times \Sigma \times S$ is a set of transitions and $\hat{s} \in S$ is an initial state. \square

Our test model is a deterministic LTS. An LTS $(S, \Sigma, \Delta, \hat{s})$ is deterministic if there is no state in which any leaving transitions share the same action name (label). For example, there are four such LTSs in Figure 1, with their initial states marked with filled circles.

Action machines and refinement machines are LTSs whose alphabets include action words and keywords, respectively. In Figure 1, \mathcal{A} is an action machine and \mathcal{R} is a refinement machine. Action machines describe *what* should be tested at action word level. In \mathcal{A} , application should be first started, then verified to be running and finally quitted. After quitting, the application should be started again, and so on. Refinement machines specify *how* action words in action machines can be implemented. Keyword sequences that implement an action word a are written in-between start_a and end_a transitions. In Figure 1, \mathcal{R} refines two action words in \mathcal{A} . Firstly, it provides two alternative implementations to action word awStartC . To start an application, a user can either use a short cut key (by pressing “SoftRight”) or select the application from a menu. Secondly, verification that the application is running is always carried out by checking that there is text “Camera” on the screen. The action word for quitting the application is not refined by \mathcal{R} , but another refinement machine can be used to do that.

During the test execution, we keep track of the current state of the test model, starting from the initial state. One of the transitions leaving the current state is chosen. If the label of the transition is not a keyword, the test execution continues from the destination state of the transition. Otherwise, the action corresponding to the keyword is taken: a key is pressed, a text is searched on the display etc. These actions either succeed or fail.

For example, text verification succeeds if and only if the searched text can be found on the display. Because sometimes failing an action is allowed, or even required, we need a way to specify the expected successfulnesses of actions in the test model. For that, we use the labeling of transitions. There can be two labels (with and without a tilde) for some keywords; $\text{kwVerifyText}\langle\text{'Clock alarm'}\rangle$ and $\sim\text{kwVerifyText}\langle\text{'Clock alarm'}\rangle$, for instance. The former label states that in the source state of the transition searching text ‘Clock alarm’ is allowed to succeed and the latter allows the search to fail.

If the taken action succeeded (failed) a transition without (with) a tilde is searched in the current state. If there is no such transition, an error has been found (that is, the behavior differs from what is required in the test model). Otherwise, the test execution is continued from the destination state of the transition.

Hence, our testing method resembles “exploration testing” introduced in [6]. However, we do not need separate output actions. This is because the only way we can observe the behavior of the SUT is to examine its GUI corresponding to the latest screen capture. In addition, there are many actions that are neither input (keyword) nor output actions. They can be used in debugging (in the execution log, one can see what action word we tried to execute when an error was detected) and in measuring the coverage (for instance, covered high-level actions can be found out).

2.3 Composing a Test Model

We use parallel composition for two different purposes. The main purpose is to create test models that enable extensive testing of concurrent systems. This means that we can test many applications simultaneously. It is clearly more efficient than testing only one application at a time, because now interactions between the applications are also tested. The other purpose is to refine the action machines by injecting the keywords of their refinement machines in correct places in them.

Refinement could be carried out to some extent by replacing transitions labeled by action words with the sequences of transitions specified in refinement machines. However, this kind of macro expansion mechanism would expand action words always to the same keywords, which might not be wanted. For example, it is handy to expand action word “show image” to keywords “select the second menu item” and “press show button” when it is executed for the first time. Later on, the second item should be selected by default in the image menu, and therefore the action word should be expanded to keyword “press show button”. We avoid the limits of macro expansion mechanism by using *transition splitting* on action machines and then letting the parallel composition to do the refinement.

The transition splitter divides transitions with given labels in two by adding a new state between the original source and destination states. If the label of a split transition is “ a ” then the new transitions are labeled as “start_ a ” and “end_ a ”.

Definition 2 (Transition splitter “ \rightarrow_A ”). Let \mathcal{L} be an LTS $(S, \Sigma, \Delta, \hat{s})$ and A a set of actions. $S_{new} = \{s_{s,a,s'} \mid (s, a, s') \in \Delta \wedge a \in A\}$ is a set of new states ($S \cap S_{new} = \emptyset$). Then $\rightarrow_A(\mathcal{L})$ is an LTS $(S', \Sigma', \Delta', \hat{s}')$ where

$$- S' = S \cup S_{new}$$

- $\Sigma' = (\Sigma \setminus A) \cup \{\text{start}_a \mid a \in A\} \cup \{\text{end}_a \mid a \in A\}$
- $\Delta' = \{(s, a, s') \in \Delta \mid a \notin A\}$
 $\cup \{(s, \text{start}_a, s_{s,a,s'}) \mid (s, a, s') \in \Delta \wedge a \in A\}$
 $\cup \{(s_{s,a,s'}, \text{end}_a, s') \mid (s, a, s') \in \Delta \wedge a \in A\}$
- $\hat{s}' = \hat{s}$ □

In Figure 1, LTS \mathcal{A}_s is obtained from \mathcal{A} by splitting transitions with labels `aw_StartC` and `aw_VerifyC`.

As already mentioned, we construct the test model with parallel composition. We use a parallel composition that resembles the one given in [7]. Whereas traditional parallel compositions synchronize syntactically the same actions of participating processes, our parallel composition is given explicitly the combinations of actions that should be synchronized and the results of the synchronous executions. This way we can state, for example, that action a in process \mathcal{P}_x is synchronized with action b in process \mathcal{P}_y and their synchronous execution is observed as action c (the result). The set of combinations and results is called *rules*. The parallel composition combines component LTSs to a single composite LTS in the following way.

Definition 3 (Parallel composition “ \parallel_R ”). $\parallel_R (\mathcal{L}_1, \dots, \mathcal{L}_n)$ is the parallel composition of n LTSs according to rules R . LTS $\mathcal{L}_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$. Let Σ_R be a set of resulting actions and \surd a “pass” symbol such that $\forall i: \surd \notin \Sigma_i$. The rule set $R \subseteq (\Sigma_1 \cup \{\surd\}) \times \dots \times (\Sigma_n \cup \{\surd\}) \times \Sigma_R$. Now $\parallel_R (\mathcal{L}_1, \dots, \mathcal{L}_n) = (S, \Sigma, \Delta, \hat{s})$, where

- $S = S_1 \times \dots \times S_n$
- $\Sigma = \{a \in \Sigma_R \mid \exists a_1, \dots, a_n : (a_1, \dots, a_n, a) \in R\}$
- $((s_1, \dots, s_n), a, (s'_1, \dots, s'_n)) \in \Delta$ if and only if there is $(a_1, \dots, a_n, a) \in R$ such that for every i ($1 \leq i \leq n$)
 - $(s_i, a_i, s'_i) \in \Delta_i$ or
 - $a_i = \surd$ and $s_i = s'_i$
- $\hat{s} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ □

A rule in a parallel composition associates an array of actions (or “pass” symbol \surd) of input LTSs to an action in resulting LTS. The action is the result of the synchronous execution of the actions in the array. If there is \surd instead of an action, the corresponding LTS will not participate in the synchronous execution described by the rule.

In Figure 1, \mathcal{P} is the parallel composition of \mathcal{A}_s and \mathcal{R} with rules

$$R = \{ \langle \text{start_awStartC}, \text{start_awStartC}, \text{start_awStartC} \rangle, \\ \langle \text{end_awStartC}, \text{end_awStartC}, \text{end_awStartC} \rangle, \\ \langle \text{start_awVerifyC}, \text{start_awVerifyC}, \text{start_awVerifyC} \rangle, \\ \langle \text{end_awVerifyC}, \text{end_awVerifyC}, \text{end_awVerifyC} \rangle, \\ \langle \text{aw_QuitC}, \surd, \text{aw_QuitC} \rangle, \\ \langle \surd, \text{kwPressKey}\langle \text{'AppMenu'} \rangle, \text{kwPressKey}\langle \text{'AppMenu'} \rangle \rangle, \\ \langle \surd, \text{kwPressKey}\langle \text{'Center'} \rangle, \text{kwPressKey}\langle \text{'Center'} \rangle \rangle, \\ \langle \surd, \text{kwPressKey}\langle \text{'SoftRight'} \rangle, \text{kwPressKey}\langle \text{'SoftRight'} \rangle \rangle, \\ \langle \surd, \text{kwVerifyText}\langle \text{'Camera'} \rangle, \text{kwVerifyText}\langle \text{'Camera'} \rangle \rangle \}$$

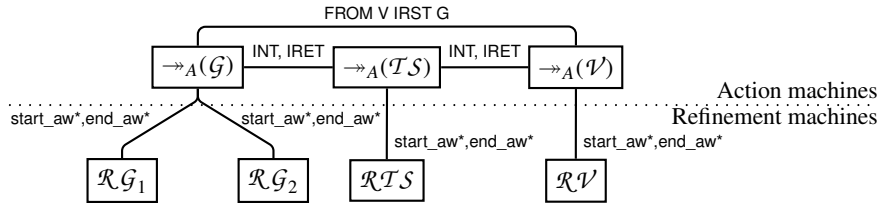


Fig. 2. Test model architecture

When using such rules, results of the parallel composition include action words (with `start_` and `end_` prefixes) and keywords (and possibly some other actions). However, when test models are walked through during a test run, communication with a SUT takes place only when keywords are encountered.

2.4 Test Model Architecture

In the SUT, several applications can be run simultaneously, but only one can be active at a time. The active application receives all user input except the one that activates a task switcher. The user can activate an already running application with the task switcher.

This setting forces us to restrict the concurrency (interleavings of actions) in the test model. Otherwise, the test model would allow executing first one keyword in one application and then another keyword in another application without activating the other application first. This would lead to a situation where the test model assumes that both applications have received one input, but in reality, the first application received two inputs and the other none.

Because the activation itself must be expressed as a sequence of keywords, it is natural to model the task switcher as a special application, a sort of a scheduler. The task switcher starts executing when an active application is interrupted, and stops when it activates another (or the same) application. Although the absence of interleaved actions might make the parallel composition look an unnecessarily complicated tool for building the model, it is not. The composition generates a test model that contains all combinations of states in which the tested application can be inactive. Thus, it enables rigorous testing of every application in every combination of states of the other applications in background.

Technically, we have one action machine for every application to be tested, and one action machine for task switching: action machines \mathcal{G} (Gallery application), \mathcal{V} (Voice recorder application) and \mathcal{TS} (task switcher), for instance. Action machines are synchronized with each other and with their refinement machines, as shown in Figure 2. Before the synchronization, all action words of action machines are split.

In the figure, lines that connect action machines to refinement machines represent synchronizing the split action words of the connected processes. For instance, we have a rule for synchronizing $\rightarrow_A(\mathcal{G})$ and \mathcal{RG}_1 with action `start_awVerifyImageList` and another rule for $\rightarrow_A(\mathcal{G})$ and \mathcal{RG}_2 with `start_awViewImage`. There are also rules that allow execution of every keyword in refinement machines without synchronization.

Synchronizations that take care of task switching are presented with lines that connect \mathcal{G} and \mathcal{V} to \mathcal{TS} in the figure. Both \mathcal{G} and \mathcal{V} include actions INT and IRET that represent interrupting the application and returning from interrupt. Initially, Gallery application is active. If \mathcal{G} executes INT synchronously with \mathcal{TS} , \mathcal{G} goes to a state where it waits for IRET. On the other hand, \mathcal{TS} executes keywords that activate another (or the same) application in the SUT and then execute synchronously IRET with the corresponding action machine.

Finally, there is a connector labeled FROM V IRST G in Figure 2. It represents “go to Gallery” function in Voice recorder. In our SUT, the function activates Gallery application and opens its sound clips menu. Voice recorder is deactivated but left in background. In the test model, action FROM V IRST G is the result of synchronizing actions IGOTO<Gallery> in \mathcal{V} and IRST<VoiceRecorder> actions in \mathcal{G} . The first action leads \mathcal{V} to an interrupted state from which it can continue only by executing IRET synchronously with \mathcal{TS} . The second action lets \mathcal{G} to continue from an interrupted state, but forces it to a state where sound clip menu is assumed to be on the screen.

Formally, our test model \mathcal{TM} is acquired from the expression:

$$\mathcal{TM} = \parallel_R (\rightarrow_A (\mathcal{G}), \rightarrow_A (\mathcal{TS}), \rightarrow_A (\mathcal{V}), \mathcal{RG}_1, \mathcal{RG}_2, \mathcal{RTS}, \mathcal{RV})$$

where set A contains all the action words and rule set R is as outlined above.

One advantage of this architecture is that it allows us to reuse the component LTSs with a variety of SUTs. For example, if the GUI of some application changes, all we need to change is the refinement machine of the corresponding action machine. If a feature in an application should not be tested, it is enough to remove the corresponding action words from application’s action machine. If an application should not be tested, we just drop out its LTSs from the parallel composition. Accordingly, if a new application should be tested, we add the action and the refinement machine for it (also \mathcal{TS} and \mathcal{RTS} must be changed to be able to activate the new application, but they are simple enough to be generated automatically). Moreover, if we test a new SUT with the same features but with completely different GUI, we redraw the refinement machines for the SUT but use the same action machines.

While refinement machines can be changed without touching their action machines, changing an action machine causes easily changes in its refinement machines. If a new action word is introduced in an action machine, either its refinement machine has to be extended correspondingly or a new refinement machine added to the parallel composition. In addition, changing the order of action words inside an action machine may cause changes in its refinement machine. For example, action word awChooseFirstImageInGallery can be unfolded to different sequences of keywords depending on the state of the SUT in which the action word is executed. In one state, Gallery application may already show the image list. Then the action can be completed by a keyword that selects the first item in the list. However, in another state, Gallery may show a list of voice samples, and therefore the refinement should first find out the list of images before the first image can be selected. Thus, action words may contain hidden assumptions on SUT’s state where the action takes place. Of course, one can make these assumptions explicit, for example, by extending the action label: awChooseFirstImageInGalleryWhenImageListIsShown.

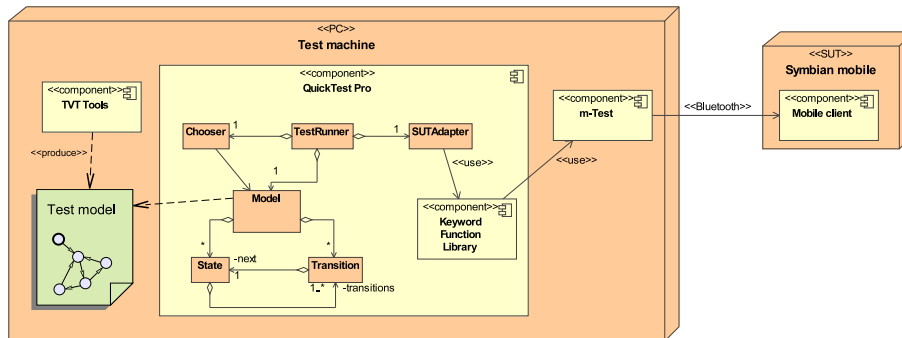


Fig. 3. Test environment

3 System Testing on Symbian Platform

The above theory was developed in conjunction with an industrial case study. In this section, we will describe the case study including the test environment and setting that we used. Moreover, we outline the implementation of our model-based test engine, and explain the modeling process concerning keyword selection and creation of the test model itself. In addition, we will briefly evaluate our results.

3.1 Test Environment and Setting

The system we tested was a Symbian-based mobile device with Bluetooth capability. The test execution system was installed on a PC, and it consisted of two main components: test automation tool, including our test execution engine, and remote control software for the SUT. The test environment is depicted as a UML deployment diagram in Figure 3. We applied TVT tools for creating the test model. As a test automation tool we used Mercury's QuickTest Pro (QTP) [8]. QTP is a GUI testing tool for MS Windows capable of capturing information about window resources, such as buttons and text fields, and providing access to those resources through an API. The tool also enables writing and executing test procedures using a scripting language (Visual Basic Script, VBScript in the following) and recording a test log when requested.

The remote control tool we used was m-Test by Intuwave [9]. It provides access to the GUI of the SUT and to some internal information such as a list of running processes. m-Test makes it possible to remotely navigate through the GUI (see Figure 4, on the left-hand side). GUI resources visible on the display cannot be obtained, only the bitmap of the display is available. m-Test can also recognize the text visible on the display back to characters (see Figure 4, on the right-hand side). m-Test supports various ways to connect to the SUT; in the study we used a Bluetooth radio link.

Moreover, in the beginning of the study, we obtained a VBScript function library. It was originally developed to serve as a library of keyword implementations for conventional test procedures in system testing of the SUT. For example, for pushing a button there was a function called 'Key' etc.

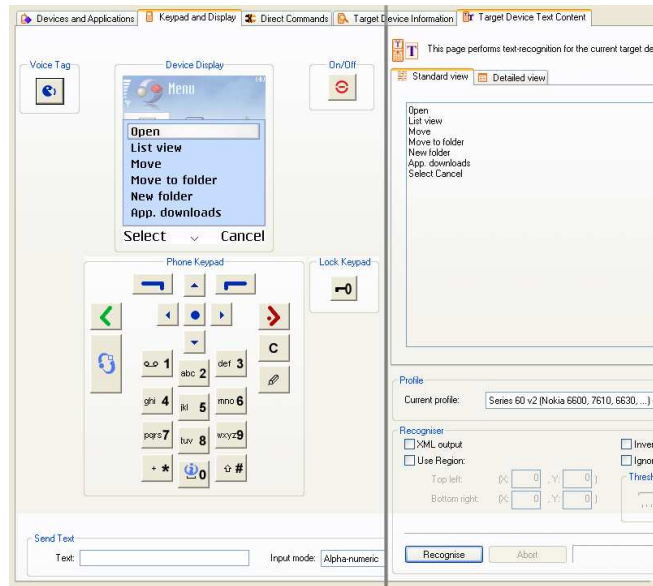


Fig. 4. Inputs and outputs of SUT as seen from m-Test

3.2 Test Engine

The test execution engine, which executes the LTS state machine, consisted of four parts: execution engine TestRunner, state model Model, transition selector Chooser, and keyword proxy SUTAdapter (see Figure 3). TestRunner was responsible for executing transition events selected by Chooser using the keyword function library via SUTAdapter. Based on the result of executing a keyword, TestRunner determines if the test run should continue or not. If the run can continue, the cycle continues until the number of executed transitions exceeds the maximum number of steps. The test designer determines the step limit that is provided as a parameter.

Model was constructed from states (State), transitions (Transition), and their connections to each other. The test model (LTS) is read from a file and translated to a state object model, which provides access to the states and transitions.

Chooser selects a transition to be executed in the current state. The selection method can be random or probabilistic based on weights attached to the transitions. Naturally, more advanced Chooser could also be based on an operation profile [10] or game theory [11], for instance. Since the schedule of our case study was tight, we chose the random selection algorithm because it was the easiest to implement.

The keyword function library that we obtained served as our initial keyword implementation. However, during the early phases of the study it became apparent that it was not suitable for our purposes. The library had too much built-in control over the test procedure. In contrast, our approach requires that the test verdict must be given by the test engine. The reason is that sometimes a failure of a SUT is actually the desired

Table 1. Keyword categories

Category	Keyword	Param.	Description
Command	kwPressKey	'keyLeft'	A key press
	kwWriteText	'Hello'	Send text
Navigate	kwSelectMenu	'Move'	Select a menu item
	kwSelectAppMenu	'Clock'	Activates an application if started
Query	kwVerifyText	'Move'	Verifies that given text is visible on the display
Control	kwSetTarget	'Phone1'	Activates a device to receive subsequent commands
	kwStartApp	'Recorder'	Start an application
State verification	kwIsMenuSelected	'Move'	Confirms that the given menu item is selected

result. In addition, since the flow control was partly embedded in the library, we did not have any keyword that would report the status of the SUT. For that reason, we created a keyword proxy (SUTAdapter). Its purpose was to hide original function library keywords, use or re-implement them to fit our purpose, and to add some new keywords.

3.3 Keyword Categories

We discovered that there must be at least five types of keywords: command, navigate, query, control, and state verification. As an example, some keywords from each category are shown in Table 1. The command type keywords are the most obvious ones: They send input to the SUT, for instance, “press key” or “write text”. Navigation keywords, such as “select menu item”, are used to navigate in the GUI. Query keywords are used to compare texts or images on the display. Control keywords are used to manage the state of the SUT. These four keyword groups are well suited for most of the common testing situations. However, our approach allowed us to create several situations where also the state verification keywords were needed.

State verification keywords verify that the SUT is in some particular state (for instance, “Is menu text selected”) or that some sporadic event, like a phone call, has occurred. These keywords were essential, because the environment did not allow us to capture such information otherwise. The state of the SUT was only available through indirect clues that were extracted from the display bitmap. Because of this, the test model occasionally misinterpreted the state of the SUT or missed an event. This made test modeling somewhat more complicated than we anticipated. The biggest difference between the query and state verification keywords is in the intent of their use. Queries are used to determine the presence of texts etc. on the display, whereas state verification keywords check if the GUI is in a required state. The latter are used to detect if the SUT is in a wrong state, i.e. the failure has occurred.

The missing of an event was the most common error in the model, which occurred often when exact timing was required (like testing an alarm). This problem was probably caused by the slow communication between QTP and m-Test. There were several occasions when some event was missed just because the execution of a keyword was too slow or the execution time varied between runs.

3.4 Modeling Process

When we started our model creation process, we had three constraints: tight schedule, many features to test, and no specifications what so ever. However, we obtained a user guide of the device and a more mature product from the same product family. The latter was used as a test oracle when developing the test model. Using the two, we were able to create a mental model³ of the behavior of the SUT in various situations. One of our objectives was to find concurrency-related defects. We explored manually through several applications and tried several basic exploration techniques like opening the same application in different ways. Within days, we found the first defect.

Since we believed that “bugs are social creatures”, we put more emphasis on the particular application involved with the first defect. After a couple of days, we found several minor defects and one major defect. All these defects where found during the development of the model.

The model itself consisted of seven components (as shown in Figure 2) and modeled a system where two applications run concurrently. After composing the component models, the final test model consisted of 297 states and 351 transitions, and used 17 action words and 18 keywords.

We found only one minor defect while executing the model. However, it should be noted that the SUT was already thoroughly tested before our case study began, and that the model was relatively small. With a larger model and the use of more advanced heuristics, we hope to find more defects. The fact that several defects were discovered before the test model was executed is mainly due to the overall benefits of precise modeling, i.e. it reveals defects very effectively. This is in line with the similar observation in [12].

3.5 Evaluation of Results

Initially, our hypothesis was that model-based testing is suitable for GUI testing and it can find more bugs than linear script-based testing. We also acknowledged that model-based GUI testing does face the same problems as conventional GUI testing, for instance in detecting the state of the SUT. In the following, we evaluate our findings and reflect those to our hypothesis.

Defects. We found six defects in total: two related to the Gallery application, three to the co-operation of Gallery and other applications (Real Player and Voice Recorder), and one to Voice Recorder only. In Table 2, the defects are described in more detail.

The only critical defect was #4, which made it almost impossible to use the device after the failure. The defect was the only one that had been reported previously; others were previously unreported. Defects #2 and #3 can be considered moderate since they allow continuing the normal use of the device. They only interfere with the use of the Gallery application. Defects #1, #5, and #6 are minor since they do not even interfere with the use of the applications.

³ According to El-Far, human testers develop mental models and the basic idea behind model-based testing is to formalize those models and use them for automatic testing [12].

Table 2. Defects and their resulting states

Defect #	Effect	Result
1	Top bar disappears from Gallery	The top bar remains missing
2	Gallery crashes I	Gallery dies while Camera stays alive
3	Gallery crashes II	Gallery dies while Voice Recorder stays alive
4	Device has to be rebooted	No playback with Real Player and no access to Gallery
5	GUI in busy loop	Recorder's GUI blinks, no sound, and no error message
6	Off by one	The selection in Gallery's menu changes either up or down

Defect #6 was the only one that we found while running the model. Even though we are not sure whether #2–#4 are different instances of the same error, this defect was clearly related to GUI component reuse (we had another product of the same family).

As discussed above, defects #1–#5 were found while getting used to the device and developing the model. Excluding #5, they could have been found by executing the model. Automatic detection of the bugs similar to #5 is generally very difficult; the blinking of the screen was so rapid that we could not capture two subsequent screens being different. It should be noted that this problem relates to the underlying GUI test automation tools, not our components or the model.

Even though our case study was not very extensive since the test model covered only a fraction of the functionality of the SUT, it gave us some promising indications towards the suitability of the approach. Especially the development of the test model was considered beneficial, because we found most of the defects while doing that.

Tools. Since our purpose was not to create a model-based GUI testing tool but to prove the concept of model-based GUI testing, the components we implemented were somewhat limited. The major problem was that we did not have a coverage-based selection for the transitions and, thus, we did not measure any model coverage. In addition, we did not provide any indication of what part of the model is being tested and when. In practice, the tester should see the test run advancing by observing the model visually. Another disadvantage was the performance of the underlying GUI testing tools; it took 5–20 seconds to execute a single keyword on an efficient PC.

One limiting factor we found was the scripting language. VBScript has several good features, but it also lacks several key features (dictionaries, inheritance etc.) that are included in other versions of Visual Basic. Those features are not so important in developing linear scripts. However, they would be very helpful in implementing state model executor and similar complex components needed in model-based testing.

4 Assessment from Industrial Point of View

Overall, the Symbian case study proved out to be beneficial. In the most practical sense, it helped by finding a few defects. It was also useful to start considering how model-

based testing methodology would fit into the normal way of working. The methodology seems to be an ever-growing promise as a testing approach, but there has not been so many industrial experiences published, especially in the GUI setting.

4.1 The First Step

One starting goal was to evaluate how existing test assets could be used in transition to model-based testing. It seems that the step towards it might not be as steep as assumed. Naturally, model-based testing is a new concept to most testers. However, with some real-life testing scenarios and demonstrations it was possible to disseminate information while keeping a practical point of view.

After initial phase, the lift-off was rapid and we could get some results quite fast. Because of the selected approach, the model was generated manually. Naturally, it would be more interesting to be able to take it into use directly from a modeling tool used by designers or generate it semi-automatically. Creation of the test model was quite straightforward and the model created in the beginning kept its original basic content quite well, i.e. maintenance effort associated with the model was not an issue here.

The tools selected as an underlying GUI test automation have been used for testing the actual products. There were certain functionality and performance problems, but otherwise the tools suited quite well for the new approach. However, Visual Basic Script was found to be quite limited with regard to programming capabilities.

It would have been interesting to see how the practical deployment of the approach would have succeeded. The commercial tools used in the study are familiar to testers of the products, but the TVT toolset is an unsupported university prototype, and its deployment would have taken some effort.

4.2 The Next Step

From a practical test management point of view, model-based testing requires a mindset change from test suites and test cases to test models. Reporting the test results is different from the traditional test cases. With a good model, you can cover many test cases and more, but it is not easy to map fully the coverage of the model to the coverage of the conventional test cases. The test management aspect would also require further studies on how to tackle the situation from the point of view of metrics collecting and test design. This has also been noticed earlier by Robinson [13]. Additional questions would also include whether model-based testing is metrics-wise more effective than conventional techniques as Apfelbaum and Doyle [14] suggest. It is also unclear how much time and money the deployment of model-based testing would take, and how the competences of the testing personnel should be developed.

Another interesting study topic would be to investigate where to obtain the model. If we would obtain a design model, how it should be modified to be used as a test model? More research is also needed on how well model-based testing responds to the changes in the product family, i.e. how portable the action word and keyword architecture is when a SUT undergoes changes at different levels (operating system vs. application software, GUI languages and locales), which all create new product variants.

Reusability and portability are promises of model-based testing. Thus, it should be investigated how the testing of relatively similar kinds of products within one product family would benefit from a model-based approach. In further studies, model-based techniques should be used in a real-life testing project already from the beginning. This would allow us to observe what kinds of defects it would reveal. The product development cycle would be enhanced if we could find the most critical ones first.

5 Related Work

There has been some research in the area of model-based testing of GUI systems. The idea of using general purpose GUI test automation tools for model-based testing originates from Robinson [15]. Ostrand et.al. [16] proposed a visual test design environment to create, edit, and maintain test scripts. They used commercial test tool to capture GUI information and replay that information back to the SUT.

Memon proposed in his Ph.D. thesis [17] a framework for testing GUI applications. The framework is based on the knowledge of GUI components. The author derives test cases from GUI structure and usage, measures test coverage and determines the correct actions of the GUI using an oracle based on previously generated test cases and runtime execution information. Belli [18] extended state machines to show not only correct GUI actions but also incorrect transitions. However, in this context, the author prefers regular expression to state machines.

In the Symbian setting, there are some fundamental restrictions compared to conventional GUI testing. Unlike usually, there is no access to the GUI resources, which makes comparisons much harder. Instead of comparing values of text fields, for instance, bitmaps and strings obtained by text recognition must be used.

Compared to Buwalda's work [3], instead of defining finite state machines using spreadsheets, we use LTSs, which is probably the simplest visual formalism for the purpose. To avoid the usual problem of visual models being cluttered, we restrict to small component models that are composed automatically. We map action words to keywords using separate refinement machines that are also defined as LTSs. Furthermore, a separate heuristic component is used to walk through the model.

An example of a related industrial tool for model-based testing is Conformiq Test Generator [19]. It uses UML state machines for test modeling and communicates with a SUT using a SUT-specific adapter. Compared to LTSs, obviously, UML state machines are much more expressive. Naturally, this helps in test modeling and introduces possibilities for data variation, for instance, by generating random data for input fields. However, making mistakes becomes easier with a formalism that is more expressive, especially when the modelers are not UML experts.

6 Conclusions

We have demonstrated how to leverage model-based testing practices in system testing through a GUI. For test modeling, we propose combining Labeled Transition Systems (LTSs) with action words, a proven method for GUI testing. Such action machines are composed in parallel with refinement machines that map the action words to keywords

corresponding to the navigation in the GUI. Finally, the composite model is read into a general-purpose GUI test automation system, which executes the model using a heuristic component, and handles the reporting. We have also conducted an industrial case study in which we tested a mobile device and found previously unreported bugs.

To summarize, our results on model-based GUI testing are quite promising. The approach is built on solid theory using proven concepts. Moreover, the associated test automation architecture is simple and includes general-purpose components. From the practical point of view, the execution of test steps should be more dependable and faster, but the issue is in the general-purpose components that are hard to replace. In the future work, a special emphasis should be placed on the topics addressed in the assessment of our results. This could pave the way for an industrial use of the approach.

References

1. Kaner, C., Bach, J., Pettichord, B.: *Lessons Learned in Software Testing*. Wiley (2001)
2. Fewster, M., Graham, D.: *Software Test Automation*. Addison-Wesley (1999)
3. Buwalda, H.: Action figures. *STQE Magazine*, March/April 2003 (2003) 42–47
4. Symbian: Symbian Operating System homepage. (At URL <http://www.symbian.com>)
5. Virtanen, H., Hansen, H., Nieminen, J., Erkkilä, T.: Tampere verification tool. In: *Proceedings of TACAS 2004*. Volume 2988 of LNCS. Springer-Verlag (2004)
6. Helovuuo, J., Leppänen, S.: Exploration testing. In: *Proc. 2nd IEEE International Conference on Application of Concurrency to System Design*. (2001) 201–210
7. Karsisto, K.: A new parallel composition operator for verification tools. Doctoral dissertation, Tampere University of Technology (number 420 in publications) (2003)
8. Mercury Interactive: QuickTest Pro homepage. (At URL <http://www.mercury.com>)
9. Intuwave: m-Test homepage. (At URL <http://www.intuwave.com>)
10. Musa, J.D.: Software reliability engineering in industry. In: *Proc. SAFECOMP'99*. Number 1698 in LNCS, Springer-Verlag (1999)
11. Kervinen, A., Virolainen, P.: Heuristics for faster error detection with automated black box testing. In: *Proc. International Workshop on Model Based Testing (MBT'04)*. Number 111 in *Electronic Notes in Theoretical Computer Science*, Elsevier (2004)
12. El-Far, I.K.: Enjoying the perks of model-based testing. In: *Proc. Software Testing, Analysis, and Review Conference (STARWEST) 2001*. (2001)
13. Robinson, H.: Obstacles and opportunities for model-based testing in an industrial software environment. (At URL http://www.geocities.com/harry_robinson_testing/ObstaclesAndOpportunities.pdf)
14. Apfelbaum, L., Doyle, J.: Model based testing. *Software Quality Week Conference* (1997)
15. Robinson, H.: Finite state model-based testing on a shoestring. (*Software Testing, Analysis, and Review Conference (STARWEST) 1999*. At URL http://www.geocities.com/model_based_testing/shoestring.htm)
16. Ostrand, T., Anodide, A., Foster, H., Goradia, T.: A visual test development environment for GUI systems. In: *ISSTA '98: Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, ACM Press (1998) 82–92
17. Memon, A.M.: A comprehensive framework for testing graphical user interfaces. PhD thesis, University of Pittsburgh (2001)
18. Belli, F.: Finite-state testing of graphical user interfaces. In: *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, (IEEE CS) 34–43
19. Conformiq Software: Conformiq Test Generator homepage. (At URL <http://www.conformiq.com>)