TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Antti Jääskeläinen

Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Applications

Julkaisu 948 • Publication 948

Tampere 2011

Tampereen teknillinen yliopisto. Julkaisu 948
Tampere University of Technology. Publication 948

Antti Jääskeläinen

Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Applications

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 28th of January 2011, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2011

ISBN 978-952-15-2513-1 (printed)
ISBN 978-952-15-3287-0 (PDF)
ISSN 1459-2045

ABSTRACT

Model-based testing is a testing methodology in which the creation of tests can be performed automatically instead of manually. This is achieved by creating a test model, which is a formal description of the aspects of system to be tested. In practice, a test model for a realistic system is often too large and complicated to create all at once. It is therefore usually a better idea to create a number of smaller and simpler model components which can be combined into the test model.

The flexibility of model-based testing can be improved by assembling the components into a model library. From the library a tester can choose a number of model components to be composed into a test model. This way, tests can be generated from a model which best corresponds to current testing needs.

This thesis focuses on the design, implementation and use of a model library for GUI (graphical user interface) testing of smartphone applications. Modern smartphones can run complex applications which interact with each other; moreover, different phones communicate with each other, adding a further level of concurrency. As such, smartphone applications present a challenging domain for testing.

We present the special considerations to be taken into account when creating model components intended to become a part of a model library, and our technical and methodological solutions to them. Flexibility is especially important: the model components have to be usable in different combinations according to the testing needs. This way features irrelevant to the tests to be generated can be left out of the model. Conversely, it is possible to create complex test models to test a variety of applications concurrently, or to test several devices and the communication between them. Furthermore, properly designed model components can be used in many different products, which can greatly reduce the effort needed for the creation of the models. Our experiences and case studies show that a well-designed model library can fulfill these needs.

PREFACE

First and foremost I would like to thank my supervisor Mika Katara, not only for his invaluable help in composing this thesis, but also for guiding me into and through my entire postgraduate studies. It is safe to say that I would not have gotten this far without his support. I would also like to thank Keijo Heljanko, Stephan Schulz and Mark Utting for their work as pre-examiners and opponents, and Antti Kervinen for reading a draft of this thesis and providing valuable feedback.

The publications presented as part of this thesis have been co-authored with Henri Heiskanen, Mika Katara, Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, Tommi Takala, Antti Valmari and Heikki Virtanen. My thanks go to all of them, and additionally to Shahar Maoz for his helpful comments on one of these publications. I would further like to thank Fredrik Abbors, Mila Keren, Johan Lilius, Qaisar A. Malik, Antti Nieminen, Julia Rubin, Dragos Truscan and Tali Yatzkar-Haham for their co-authorship in other publications which unfortunately did not fit into this thesis.

The research projects which resulted in this thesis have been funded by TEKES (Finnish Funding Agency for Technology and Innovation) and Academy of Finland (grant number 121012), as well as the companies Conformiq, Cybercom Plenware, F-Secure, Ixonos, Nokia, Prove Expertise, Qentinel and Symbio, for which I thank them all. I am also grateful to Tampere Doctoral Programme in Information Science and Engineering for funding my post-graduate studies and Tampere University of Technology for its contribution to the research effort.

Finally, I want to offer thanks to everyone in the research teams in which I have been working, including Henri Heiskanen, Marek Janicki, Mika Katara, Antti Kervinen, Merja Leppänen, Lotta Liikkanen, Mika Maunumaa, Jukka Mikkonen, Mika Mäenpää, Antti Nieminen, Tuomas Pajunen, Mikko Satama, Tommi Takala and Heikki Virtanen. This thesis would not have been possible without their efforts.

TABLE OF CONTENTS

LIST OF INCLUDED PUBLICATIONS

[P1] A. Jääskeläinen, M. Katara, A. Kervinen, H. Heiskanen, M. Maunumaa, and T. Pääkkönen, "Model-based testing service on the web," in *Proceedings of the Joint Conference of the 20th IFIP International Conference on Testing of Communicating Systems and the 8th International Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES 2008)*, ser. Lecture Notes in Computer Science, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, Eds., vol. 5047. Berlin, Heidelberg: Springer, Jun. 2008, pp. 38–53.

[P2] A. Jääskeläinen, A. Kervinen, and M. Katara, "Creating a test model library for GUI testing of smartphone applications," in *Proceedings of the 8th International Conference on Quality Software (QSIC 2008) (short paper)*, H. Zhu, Ed. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2008, pp. 276–282.

[P3] A. Jääskeläinen, A. Kervinen, M. Katara, A. Valmari, and H. Virtanen, "Synthesizing test models from test cases," in *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software, Verification and Testing (HVC 2008)*, ser. Lecture Notes in Computer Science, H. Chockler and A. J. Hu, Eds., vol. 5394. Berlin, Heidelberg: Springer, May 2009, pp. 179–193.

[P4] A. Jääskeläinen, "Filtering test models to support incremental testing," in *Proceedings of the 5th Testing: Academic & Industrial Conference – Practice and Research Techniques (TAIC PART 2010)*, ser. Lecture Notes in Computer Science, L. Bottaci and G. Fraser, Eds., vol. 6303. Berlin, Heidelberg: Springer, Sep. 2010, pp. 72–87.

[P5] H. Heiskanen, A. Jääskeläinen, and M. Katara, "Debug support for model-based GUI testing," in *Proceedings of the 3rd IEEE International Conference*

on Software Testing, Verification, and Validation (ICST 2010). Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2010, pp. 25–34.

[P6] A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen, "Automatic GUI test generation for smartphone applications – an evaluation," in *Proceedings of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2009, pp. 112–122 (companion volume).

[P7] A. Jääskeläinen, T. Takala, and M. Katara, "Model-based GUI testing of smartphone applications: Case S60 and Linux," in *Model-Based Testing for Embedded Systems*, ser. Computational Analysis, Synthesis, and Design of Dynamic Systems, J. Zander, I. Schieferdecker, and P. J. Mosterman, Eds. CRC Press, 2011, to appear.

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CSP	Communicating Sequential Processes
GUI	Graphical User Interface
LSTS	Labeled State Transition System
MMS	Multimedia Messaging Service
OCR	Optical Character Recognition
SMS	Short Message Service
SUT	System Under Test
TEMA	Test Modeling Using Action Words
UI	User Interface
UML	Unified Modeling Language

1. INTRODUCTION

To begin, we will examine software testing and test automation, leading to model-based testing. Special attention will be paid to testing through a GUI (graphical user interface) and in the domain of smartphone applications. We will also introduce the particulars of the research, including tools, research methods, related work and the publications included in the thesis.

1.1 Software Testing

Software testing is the process of examining and exercising software in ways likely to uncover potential errors in it. In practice it is the primary method of ensuring the viability of the software. Methods based on the examination of design artifacts are called *static testing*, while those based on running the software or parts of it are called *dynamic testing*. In practice, the term 'testing' is often used to refer specifically to dynamic testing.

Dynamic testing can be roughly divided into two categories: *black-box testing* and *white-box testing* [9]. Black-box testing is based solely on the observable behavior of the system under test (SUT). Conversely, white-box makes use of the implementation of the SUT, in practice the source code. Sometimes a third category, *grey-box testing*, is used to denote methods which make use of limited information on the implementation but cannot be considered pure white-box testing.

Testing can also be classified according to the level of the software development process it focuses on [9]. *Unit testing* considers a single class, module or other relatively small component of the software. Increasingly large combinations of the units are tested in *integration testing*, which eventually leads to *system testing* when the whole software has been assembled. Typically this process begins with white-box testing and gradually moves to black-box testing as the SUT increases in complexity. One

special case is *GUI testing*, which focuses on testing through the graphical user interface of the software.

Testing focusing on functional requirements of the software is called *functional testing*. Its purpose is to ensure that the software does what it is supposed to. There are also various kinds of *non-functional testing*, such as *performance testing*, which makes sure that the software works fast enough; *stress testing*, which tests the software under difficult conditions like exceptional workloads; and *usability testing*, which focuses on the ease of use.

Dynamic testing is generally performed via *test cases*. A test case is a description of the inputs to be given to the SUT and the outputs the SUT is expected to yield. The level of abstraction can vary, from cases which are little more than guidelines to manual testers, to a detailed enumeration of every single step.

Regardless of the methods used, testing is an expensive process. Intuitively, the resources required for proper testing increase faster than the size of the SUT. With the ever more complex modern software it is necessary to find some means for reducing these expenses.

1.2 Test Automation

One way to reduce the costs of testing is to automate parts of it. Traditionally *test automation* has focused on automating the execution of test cases. Over time, various approaches have been developed [10].

Capture and replay technologies were the first test automation methods. Their idea was to record a manually performed test, which could then be repeated automatically. The results were mixed. The method was only capable of automatically executing tests which had already been performed manually, which effectively limited it to regression testing. Furthermore, even the smallest change in the SUT could render the recordings obsolete and new ones had to be created from scratch.

The worst shortcomings of the capture and replay methods are fixed in *programmatic scripts*. They are essentially small programs designed to perform an individual test run. A script can be written based on documentation, without having to perform the test on the SUT; moreover, an obsolete script can be updated to match the current

state of the product. Finally, programmatic control structures such as branches and loops enabled the design of more complex tests in a more concise form.

Many programmatic scripts can be improved by separating the test case information from the execution logic. That is, the contents of the test are described using a separate simple format, and executed with a program designed for that format. The most important benefit of these *data-driven scripts* is that the execution logic can be created by programmers, and testers can concentrate on designing effective tests. The simple format also reduces maintenance effort somewhat.

One way to further improve maintainability is to separate the tested functionality and its implementation. The test cases are defined with *action words*, which describe the functionality of the SUT in an abstract level. An action word might, for example, denote launching an application or checking that the results of a calculation are correct. Every action word has its implementation defined in *keywords*, which correspond to the user interface (UI) or application programming interface (API) events, such as pressing a key or reading the contents of a text field. A single action word can be used in several test cases, while its implementation needs to be defined only once. This means that any changes to the UI/API only require updates to the implementations of the action words, whereas the test cases only need to be modified if functionality is altered. These ideas are described in [6, 10], though the terminology is used slightly differently.

1.3 Model-Based Testing

As explained above, the traditional test automation focuses on automating test execution. However, the tests are still created manually. *Model-based testing* is a methodology which seeks to automate this part of the testing process.

Model-based testing is based on a formal description of the SUT, called a *test model*. The model may contain information on such things as available commands, expected responses, legal and illegal data values, and so on. What is modeled depends on the type of the SUT and the aspects to be tested. The tests are then generated based on the modeled information.

There are two basic methods of model-based testing [14]. The simpler one is called *off-line testing*. Its idea is to use the model to generate a sequence of actions, which

are then treated and executed as traditional test cases. Thus, off-line testing fits easily into traditional testing process as a replacement for or a supplement to manual test creation.

A more advanced alternative is *online testing*, in which tests are executed as they are being generated. Separate test cases are not used at all. When an action is chosen in the model, it is sent to the SUT for execution. The results of execution are then returned to test generation, where they may affect the selection of the future test steps. This methodology makes online testing especially well suited for testing nondeterministic systems, where the results of execution are not always known beforehand. For example, if the SUT includes a communication channel that may occasionally fail, an online test can automatically adapt to a failure and try again. The downside is that online testing does not fit as well into traditional testing process, and test generation may be more difficult.

Model-based testing offers two main advantages over manual test creation. First, a model-based test can examine the modeled aspects of the SUT in any combination. Such tests are not limited to the creativity of human designers, and may exercise the SUT in ways no one has considered. The second advantage is maintainability. Models can contain a large amount of information in a very concise form. An update which might cause modifications to dozens of test cases in a test suite might be performed by a single change to the model.

Although model-based testing is compatible with automated test execution, combining the two is not necessary: it is perfectly possible to use the models to generate abstract, manually executable tests. This may be a useful technique if automatic execution of tests is very difficult or expensive to arrange. In practice, though, automatic test execution is immensely valuable, and if it is available there is ample reason to design the models to take advantage of it.

1.4 GUI Testing

If the SUT includes a GUI, it is important to perform testing through it. This does not mean just testing the GUI widgets; rather, the purpose of GUI testing is to test the whole SUT as the user experiences it. GUI testing is usually the last part of the testing process, since it touches all parts of the SUT. GUI testing can be functional or

non-functional; here we will focus on the former.

GUIs also present a greater challenge to automatic test execution than textual UIs and APIs do [21]. The most obvious problem is evaluating outputs. Instead of an unambiguous textual or programmatic value, the output of a GUI is an image. The simplest way to handle this would be to define a specific image as the expected output. However, such a solution is impractical, since even insignificant differences in the image will cause a test to fail. Some method for extracting the relevant information is needed.

In the best case the SUT offers an API, possibly one designed specifically for testing, through which the contents of its display can be queried. This will effectively enable GUI testing through an API. In practice, it does not work quite perfectly, though. In many cases the display contents returned through the API are not an exact match for what is actually visible on the display; for example, text in a window partially hidden behind another might still be considered fully visible.

An alternative is to take a screenshot of the display and to use optical character recognition (OCR) to extract the text from it. In theory, this will exactly yield the actual contents of the display. However, OCR tends to be somewhat unreliable, occasionally missing or misreading characters, though these problems may be mitigated by manipulating the image. Processing images is also significantly slower than API calls. This is particularly significant in online testing, where execution times may affect the choice of test generation algorithms [34].

1.5 Testing Smartphone Applications

Our research has focused on model-based online functional GUI testing of smartphone applications. Modern smartphones are akin to handheld computers, capable of executing programs, creating and presenting multimedia, and so on. Thus, their applications require the same kind of testing as PC software.

There are many different types of smartphones, or *products*. Each product has a set of *applications* it can execute, such as Messaging and Calendar. Many applications are available on several different products. However, the applications may have very different UIs on different products, even if the underlying functionality is mostly the same. In contrast, the different physical phones of an individual product, or *devices*,

can be considered identical except for some identifying traits such as phone numbers. Products can be grouped together into *product families*, such as Symbian [35] or MeeGo [29] phones.

The smartphone market tends to emphasize short lead times and the simultaneous development and production of several different products within a single product family [3]. This poses an extra difficulty for the use of model-based testing in the smartphone domain, since creating new models from scratch for every product becomes impractical. As such, the models should be designed with reusability in mind. Preferably at least product families should be able to share the same models for functionality, even if the specifics of the GUIs have to be modeled separately. With careful design models may be reused even in different product families.

1.6 Toolset Overview

In our research, we have developed a toolset for facilitating online model-based testing, called the TEMA toolset. TEMA stands for Test Modeling Using Action Words, the name of the project in which most of the tools were first developed [44]. The toolset has been designed for the smartphone domain, but could be used with other domains as well. Figure 1 illustrates the architecture of the toolset.

The first part of the toolset is concerned with test modeling. The main tool here is Model Designer, which is used to create and organize the models. Test modeling also makes use of various model utilities to analyze the created models. The main output here is a *model library* consisting of various model components; their nature will be explained later.

Once the models are ready, the tools of the Test Design portion are used to design and launch the execution of a model-based test. This is generally done through Web GUI [P1], which can be used to examine the available models and to create *test configurations*. A test configuration defines an executable model-based test, including what kinds of devices are involved, how the test model is to be assembled, what kind of a test is to be generated, and according to which parameters. Usually test configurations are executed through the Web GUI using Test Controller, but they can also be packed into a *test execution script*, which can later be used to generate the test without using Web GUI, for example in a continuous integration cycle [11].

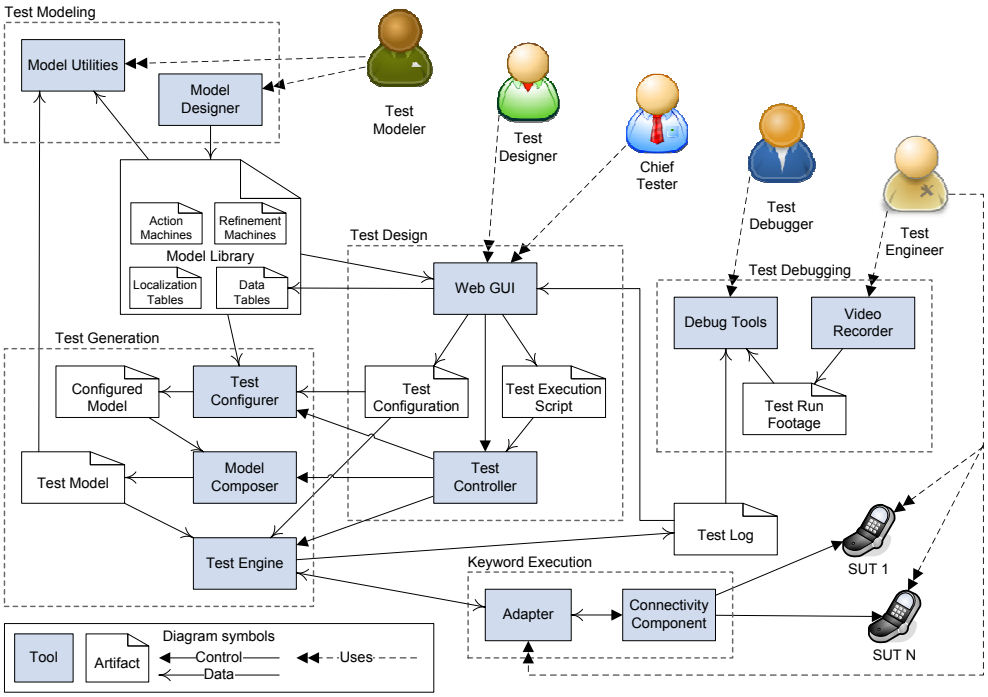

Fig. 1. *The TEMA toolset and the associated user roles [P7].*

Actual test generation is performed in its own portion of the toolset. Here the components in the model library are first configured by Test Configurer to match the requirements of the test as defined in a test configuration, and then composed by Model Composer into a test model. Test Engine can then begin the execution according to the parameters defined in the test configuration.

The automatic execution of the test requires a method for connecting into the SUT. This is the responsibility of the Keyword Execution portion. The actual communication is handled by an appropriate Connectivity Component, whose nature depends on the product family of the SUT. Between it and Test Engine is Adapter, which takes the keywords executed in the test model and converts them into a form understood by Connectivity Component, and feeds the results of the execution back to Test Engine. The result of keyword execution is a simple Boolean value representing the success or failure of execution. A difference between the received result and the one expected by the test model indicates an error, either in the SUT or in the model.

Test Engine records the events of the test run into a *test log*, which is stored into Web GUI. The log can be used to examine what has been tested, but is more importantly

needed by the Test Debugging portion of the toolset, which is used to locate the cause of any errors and inconsistencies occurring during the test runs. There are various debug tools to ease this process. Of particular note is Video Recorder, which is used to record footage of the smartphone display during the test run. This footage can be a great help in debugging.

Figure 1 also shows the roles of the various people involved in the model-based testing process. First, the test modeler is responsible for the creation and maintenance of the models. Second, the test designer defines what kinds of tests are to be created and oversees their execution. Third, the test engineer takes care of the physical devices and prepares them for testing, and also sets Video Recorder to record them as needed. Fourth, test debugger is responsible for tracing any anomalies found back to their cause. Finally, chief tester oversees the whole testing process.

This distribution of concerns has significant advantages. Most importantly, it allows specialization; for example, most of the testing personnel need not have any understanding of the models or their creation. Furthermore, sensitive prototype devices need not ever be given into the hands of most of the testing personnel, and possibly not even shown if the documentation is extensive enough. As a consequence, different parts of the process need not be performed at the same location; for example, it is possible to generate tests for phones connected to a network in another country. Of course, it is still perfectly possible and sometimes desirable for a single person to assume more than one role.

1.7 Related Work

The practical aspects of model-based testing are described in great detail by Utting and Legeard [46]. Utting et al. [47] have also developed a taxonomy for model-based testing. A detailed analysis of how our methodology would be classified can be found in [24]; in short, we use transition-based test models of the environment to generate online tests, either randomly or according to requirements or model coverage.

Much of model-based testing research has focused on testing through APIs, such as protocol testing in [45]. Model-based GUI testing has been researched for example by Robinson [41], who used existing GUI test automation tools for executing model-based tests, and Ostrand et al. [37], who developed their own testing tool based on

capture-and-replay methodologies. Memon [31] also developed his own framework for GUI testing.

The formal methods used in the handling of the models owe much to research in *software verification*, which seeks to mathematically prove the correctness of the SUT. Many of our methods specifically are originally based on CSP (Communicating Sequential Processes) [42], and further developed by Helovuo and Valmari [16] and Karsisto [22]. More generally, our methodology is based on work by Helovuo and Leppänen [15], and Kervinen and Virolainen [24, 25]. The work presented in this thesis has been applied outside of our research group for example by Mikkola [33].

Other modeling formalisms [26] can also be used in model-based testing. For example, there has been research on model-based testing using UML (Unified Modeling Language) [36] models. This approach can be seen side by side with ours in [30]. The UML-based approach also has commercial tool support, such as Conformiq Tool Suite [8, 17].

Yet another approach can be seen in the NModel framework [32] developed by Microsoft. It is based on *model programs*, which are executable specifications for the SUT, written in C#. The framework is described in [20]. NModel offers a simple method for combining several model programs into a more complex one, which can be used for testing several applications concurrently. The methodology is somewhat less flexible than ours, though; in particular, while model programs can be interleaved, there is no inbuilt mechanism for controlling the switches between them. Microsoft also makes use of another model-based testing tool called Spec Explorer [7].

Apart from different formalisms, there are variations in what is modeled. For instance, Belli et al. [1, 2] have proposed a holistic modeling method, in which the system models describing how the SUT functions are supplemented with fault models which describe how it does not function. This would enable a more comprehensive testing of faulty inputs, whose handling is often a secondary concern in product development and implementation. Bouquet et al. [4] annotated their model with requirement information in order to trace the generated tests back to requirements; this is in contrast to our approach, where requirements can be expressed in terms of the model, but not as a part of it. Model-based testing can also be used to generate test data instead of or in addition to control decisions, such as in [5,27,28], where Legeard et al. describe the generation of boundary-value test cases.

1.8 Research Method

The research described in this thesis has been conducted mostly as practical experimentation using a constructive research method. This has involved creating tools and models, running tests, and analyzing the results. The analysis has been mostly qualitative because of the difficulties involved in proper quantitative analysis.

Quantitative analysis could be performed for example by testing a new product concurrently with traditional and model-based methods and comparing the results. This would show whether the model-based testing process can find the bugs revealed by the traditional methods, and whether it can find bugs the traditional methods miss, as well as providing a comparison between the efforts required for each approach. However, in practice this is difficult to arrange. Performing such experiments within the ordinary development process would double the workload, which is rather impractical at least in the smartphone domain with the short lead times of product creation. Conversely, performing the experiments externally would require giving outsiders access to prototypes, something product developers are reluctant to allow. Some measure of quantitative analysis might also be achieved by examining the reports of found bugs and estimating whether the model-based methods could or would have found them, but this approach faces similar confidentiality problems.

Despite the lack of quantitative analysis, case studies with the industrial partners have been a valuable addition to the research. They have allowed us to try our methods in a realistic environment and occasionally with prototypes to which we would not otherwise have access. Likewise, the experiences of new users have been of great value in tool development, such as in [33].

1.9 Included Publications and Author's Contributions

This thesis includes seven publications. The author has made several contributions to the TEMA toolset and publications, mostly related to modeling. He developed the concept of model library, created most of the models used in the research, developed many of the modeling techniques, and made some contributions to the model formalism originally developed by Antti Kervinen [24]. The work on proper model libraries was begun in the author's Master's Thesis [18], which introduced the Model Designer

tool. The author also developed and implemented some utility tools for analyzing the models, and contributed to the Test Configurer and debug tools. Furthermore, he has had an active part in executing and debugging tests, and has participated in all of the case studies mentioned in this thesis. He has also taken an active part in the writing of the publications.

[P1], "Model-based testing service on the web," presents a web interface (Web GUI of the TEMA toolset) designed to offer the means to 'order' a model-based test, meaning generating and executing the test without having to deal with the models directly. The author's contributions concerned the presentation of the models and their actions in the interface; he also wrote the sections concerning the modeling formalism and the toolset in the publication. The interface was implemented by Henri Heiskanen, and has seen significant use during the research projects. A second version based on the first was developed later on to enable more flexible test generation and better usability.

[P2], "Creating a test model library for GUI testing of smartphone applications," presents the author's creation of an open-source model library for testing of smartphone applications, with focus on presenting the structure of a working model library and the techniques used to create it. The formalism used in modeling was designed by Antti Kervinen with minor contributions by the author, while the practical modeling techniques described in the publication were developed by the author during the modeling process. The main content of the publication is based on the author's Master's Thesis.

[P3], "Synthesizing test models from test cases," presents a methodology for creating a test model by combining existing test cases at common action sequences. The basic principle was first proposed by Antti Valmari, and Antti Kervinen developed a tool to support it. However, while the principle was sound, it proved insufficient in combining real test cases, which were often irreconcilable due to simple differences in setup. The author developed a method and rudimentary tools for separating the setup information from the test cases so that they could be effectively combined, and reattaching the information afterward. The improved process was successfully applied to a number of real-world test cases in a case study and produced an executable model. In the publication, the author wrote the sections on the synthesizing process and the case studies.

[P4], “Filtering test models to support incremental testing,” addresses the problem of models containing functionality that cannot be executed on the SUT. This may occur during development when the models are completed before implementation, or after testing when functionality is found faulty. The publication presents a method and tools for filtering unexecutable functionality out of the models without making them infeasible for test generation. The research was done and the publication written entirely by the author.

[P5], “Debug support for model-based GUI testing,” presents two practical methods for debugging model-based tests: a video recording method, in which the test run is recorded on video and synchronized with test log material in order to give a human debugger a clearer idea of the events of the test, and a trace incrementation method, in which progressively larger portions of the error trace are executed until the error is reproduced. The main part of the research was done by Henri Heiskanen with help and supervision from the author. The author contributed in the design and implementation of both methods, and wrote the section on model-based testing and the comparisons between the trace minimization techniques in the publication. The developed debugging methods have since proven very useful in the case studies.

[P6], “Automatic GUI test generation for smartphone applications – an evaluation,” presents an overview of the model-based testing research performed in the TEMA project, with focus on executing long-period tests on S60 smartphones. The author’s main contributions were the creation of the model library, the modeling tools, and the modeling techniques. The author also made some smaller contributions to the model semantics and the tools handling the models before and during test generation, and participated in the case studies. The author’s contribution to the writing of the publication was minor.

[P7], “Model-based GUI testing of smartphone applications: Case S60 and Linux,” presents two case studies of the use of our methodology on testing applications on different platforms. The presentation of the former is an elaboration on the work described more briefly in [P6]. The author created the test models for S60 and Mobile Linux smartphone applications, and also developed methods for model-based testing of real-time properties. Tommi Takala developed the adaptation tools which allow the automated execution of the generated tests. In the publication, the author wrote the sections on the theory of modeling and test generation, as well as the ones on the modeling in the case studies.

1.10 Structure of the Thesis

The remainder of the thesis is structured as follows: Chapter 2 presents the formalism used in creating the models, and the semantics related to their use. Chapter 3 details various modeling techniques developed and used during the research, as well as an example of a model library. Chapter 4 gives a detailed description on how the models are used throughout the testing process. Finally, Chapter 5 draws the conclusions.

2. MODEL FORMALISM

The TEMA toolset is designed to work with test models composed of numerous model components, each of which depicts only a small part of the SUT. The reason for this is complexity: the test model for even a small SUT is likely to be prohibitively large when viewed as a whole, and therefore very difficult to create and maintain all at once. It is much easier to create a number of smaller and simpler components, and then combine them automatically into a working test model. In this chapter, we will examine the structure of the individual components and the methods used in combining them, as well as test data.

2.1 Model Components

Individual model components are used to describe specific and limited aspects of the SUT. Such an aspect might be a specific view in an application, such as the images view in the Gallery application, or a specific task, such as creating a multimedia message. This section presents the formal definition of the model components and explains how we use them.

2.1.1 Behavioral Models

The model components are behavioral models depicting the functionality of the SUT. Although many different formalisms can be used to express behavior, all of them can be reduced to state machines: the model is always in some specific state, and there are ways to move it to different states. A formalism may emphasize the states or the transitions between them, or include information on both. Our choice of formalism is the labeled state transition system (LSTS), which belongs to the last category, as its name implies [13]. An LSTS is defined as follows:

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 practice we use LSTS mostly as a transition-based formalism. The actions encode the inputs as well as the expected outputs; thus, the models function as the test oracle. The attributes serve various secondary roles: some act as shorthand for recurring model structures, others mark important states for test generation, and yet others exist merely to improve the legibility of the models. Note that while transitions are always labeled with exactly one action, a state may have any number of attributes.

Given its general nature, models of other behavioral formalisms can be easily converted into LSTSs. This allows us to use other formalisms where they are useful. However, Test Engine handles all models in LSTS form.

2.1.2 Component Types

The TEMA toolset divides the model components into two main categories: *action machines* and *refinement machines*. The division corresponds to action words and keywords. Each application on the SUT is modeled in one or more action machines, which describe the functionality of those applications in action words. Apart from action words, the action machines also contain many synchronization actions, with which they can be connected to other action machines when a test model is composed. Figures 2 and 3 show examples of action machines. Like action words, action machines are independent of the UI. This enables using the same action machines on many different products, which makes both modeling and maintenance easier.

Correspondingly, refinement machines are used to define the GUI implementation of the functionality of the SUT. Each action machine has one or more refinement machines, in which the implementations of the action words are specified using keywords. Unlike action machines, refinement machines are product-specific, since the same functionality may be implemented differently on different products. A typical refinement machine has the action refinements as loops on its initial state, as shown

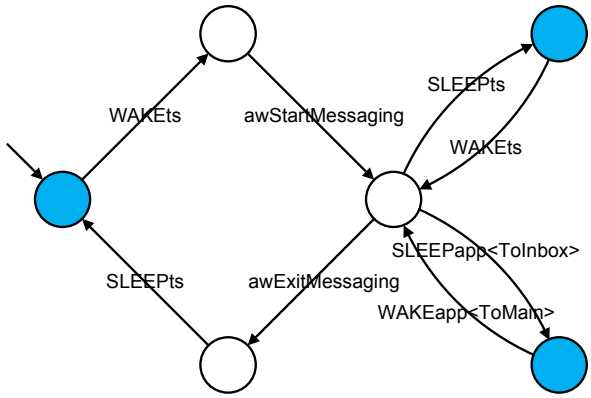

Fig. 2. *A simple action machine for the main screen of the Messaging application, with sleeping states colored blue.*

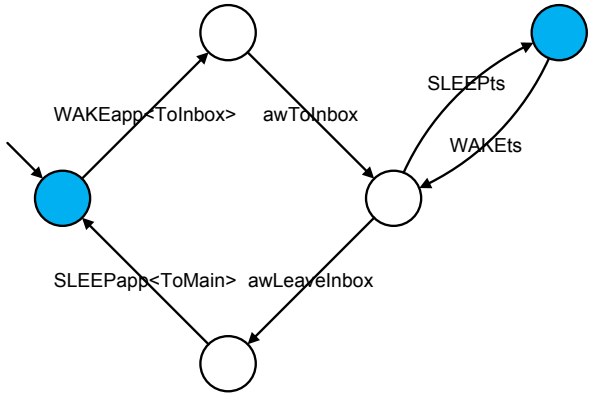

Fig. 3. *An action machine for opening and closing the inbox in the Messaging application.*

in Figure 4. Most refinements are simple linear sequences, but branches and loops are also possible. An individual action refinement is bracketed by a starting and ending synchronization that define which action word it refines.

Action machines are designed so that only one of them is *active* at a time, and the others are *sleeping*. This is modeled by dividing the states of the action machine into active and sleeping states (the latter are generally marked with the attribute *SleepState*, represented with blue coloring in the figures of this thesis). Action words are placed solely between active states; thus, only the active action machine can perform actions on the SUT. Other actions are used to synchronize action machines with each other. Depending on the type, they may be placed between active states (queries

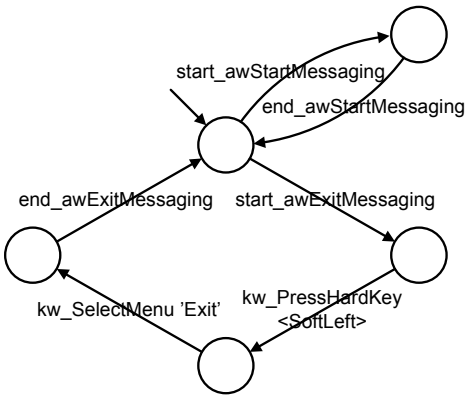

Fig. 4. *A refinement machine for the Messaging Main action machine shown in Figure 2. The action word* awStartMessaging *can be given an empty implementation, since the application is launched automatically with the activation of the action machine.*

and messages), between sleeping states (responses to the former), going from active to sleeping states (sending the current action machine to sleep), or from sleeping to active states (activating the action machine). The separation of sleeping and active states is part of the well-formedness rules and has no semantic effect, with the exception of one graph transformation which makes use of the *SleepState* attribute. The primary purpose of the separation is to make the models easier to understand, though the attributes may also be useful in test generation.

The active action machine can yield control either to another action machine directly or to a *task switcher*. A task switcher is a special action machine which acts as a scheduler for a single device, or *target*, with the ability to activate any action machine prepared to take control. It also has its own refinement machine, which automatically activates the appropriate application on the SUT when another action machine is activated. This ensures that further keywords will affect the correct application. Task switchers are usually generated automatically based on the other model components involved. However, it would also be possible to create one manually in order to customize the possible switches between the components.

Following are the definitions of the generated task switcher and its refinement machine. In the definitions, string literals are written within quotation marks; ".*" means an arbitrary string of characters as in Python regular expressions [40]. The symbol '&' denotes string concatenation. Figure 5 shows an example of a task switcher.

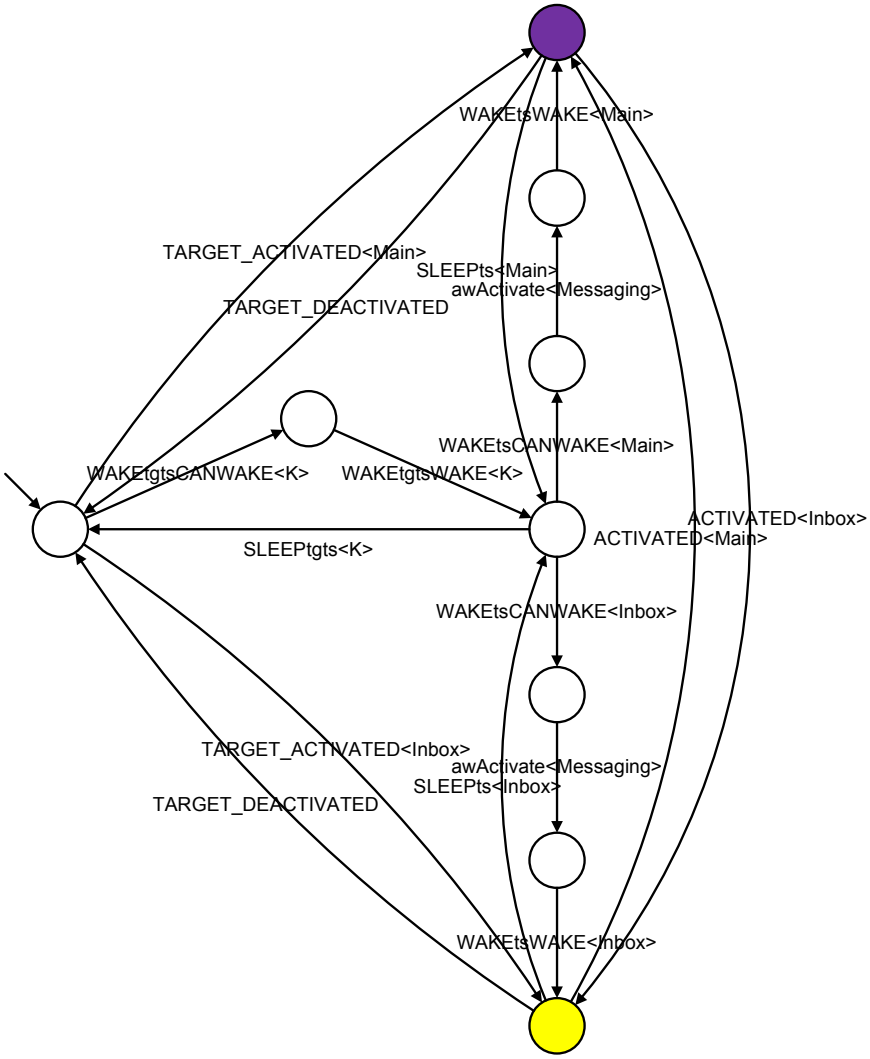

Fig. 5. *An example task switcher, generated for target K with model components Main and Inbox (Figures 2 and 3). The purple and yellow coloring in the states mark the attributes* Main running *and* Inbox running*, respectively.*

Definition 2 (Generated task switcher $TsS_{K,an}$):
$TsS_{K,an}(L_1,\ldots,L_n)$ *is the* generated task switcher *for the target K with the action machine LSTSs* $L_1,\ldots,L_n$*, with the* activation name function $an : \{L_1,\ldots,L_n\} \longrightarrow$ ".*".
$TsS_{K,an}(L_1,\ldots,L_n) = (S,\Sigma,\Delta,\hat{s},\Pi,val)$*, where*

- $S = \{s_i, s_i', s_i'' \parallel 0 \leq i \leq n\}$
- $\Sigma = \{$"WAKEtgtsCANWAKE<"&$name(K)$&">",
  "WAKEtgtsWAKE<"&$name(K)$&">","SLEEPtgts<"&$name(K)$&">",
  "TARGET_DEACTIVATED"$\}\cup$
  $\{$"WAKEtsCANWAKE<"&$name(L_i)$&">",
  "awActivate<"&$an(L_i)$&">","WAKEtsWAKE<"&$name(L_i)$&">",
  "SLEEPts<"&$name(L_i)$&">","ACTIVATED<"&L_i&">",
  "TARGET_ACTIVATED<"&$name(L_i)$&">" $\parallel 1 \leq i \leq n\}$
- $\Delta = \{(s,a,s') \in S \times \Sigma \times S \parallel$
  $(s = s_0 \wedge a =$"WAKEtgtsCANWAKE<"&$name(K)$&">"$\wedge s' = s_0') \vee$
  $(s = s_0' \wedge a =$"WAKEtgtsWAKE<"&$name(K)$&">"$\wedge s' = s_0'') \vee$
  $(s = s_0'' \wedge a =$"SLEEPtgts<"&K&">"$\wedge s' = s_0) \vee$
  $\exists i; 1 \leq i \leq n :$
  $(s = s_0'' \wedge a =$"WAKEtsCANWAKE<"&$name(L_i)$&">"$\wedge s' = s_i) \vee$
  $(s = s_i \wedge a =$"awActivate<"&$an(L_i)$&">"$\wedge s' = s_i') \vee$
  $(s = s_i' \wedge a =$"WAKEtsWAKE<"&$name(L_i)$&">"$\wedge s' = s_i'') \vee$
  $(s = s_i'' \wedge a =$"SLEEPts<"&$name(L_i)$&">"$\wedge s' = s_0'') \vee$
  $(\exists j; 1 \leq j \leq n \wedge j \neq i :$
  $s = s_j'' \wedge a =$"ACTIVATED<"&$name(L_i)$&">"$\wedge s' = s_i'') \vee$
  $(s = s_0 \wedge a =$"TARGET_ACTIVATED<"&$name(L_i)$&">"$\wedge s' = s_i'') \vee$
  $(s = s_i'' \wedge a =$"TARGET_DEACTIVATED"$\wedge s' = s_0)\}$
- $\hat{s} = s_0$
- $\Pi = \{name(L_i)\&$" running" $\parallel 1 \leq i \leq n\}$
- $val(s) = \{name(L_i)\&$" running" $\parallel 1 \leq i \leq n \wedge s = s_i''\}$

Definition 3 (Generated task switcher refinement $TsS_{an} - rm$):
$TsS_{an} - rm(L_1,\ldots,L_n)$ *is the* generated task switcher refinement *for the*

LSTSs $L_1,\ldots,L_n$*, with the activation name function* $an : \{L_1,\ldots,L_n\} \longrightarrow$ ".*". $TsS_{an} - rm(L_1,\ldots,L_n) = (S,\Sigma,\Delta,\hat{s},\Pi,val)$*, where*

- $S = \{s_0\} \cup \{s_i, s_i' \,\|\, 1 \leq i \leq n\}$
- $\Sigma = \{$"start_awActivate<"&$an(L_i)$&">",
  "kw_LaunchApp '"&$an(L_i)$&"'","end_awActivate<"&$an(L_i)$&">" $\|$
  $1 \leq i \leq n\}$
- $\Delta = \{(s,a,s') \in S \times \Sigma \times S \,\|\, \exists i; 1 \leq i \leq n :$
  $(s = s_0 \wedge a =$"start_awActivate<"&$an(L_i)$&">"$\wedge s' = s_i) \vee$
  $(s = s_i \wedge a =$"kw_LaunchApp '"&$an(L_i)$&"'"$\wedge s' = s_i') \vee$
  $(s = s_i' \wedge a =$"end_awActivate<"&$an(L_i)$&">"$\wedge s' = s_0)\}$
- $\Pi = \emptyset$
- $val(s) = \emptyset$

It is possible to create test models which act on multiple targets at once, such as modeling one phone sending a text message to another. Test Configurer will automatically create individual copies of the model components and the task switcher for each target. These are combined with another automatically generated model component called *target switcher* which acts as a scheduler for different devices just as the task switchers act for the model components of their respective devices. The structure of the target switcher is correspondingly very similar to that of the task switchers, with targets substituted for model components. The automatically generated target switcher and its refinement machine are defined below, with an example shown in Figure 6. Using multiple targets also requires one more automatically generated model component called *synchronizer*, which is used in forming connections between model components on different targets.

Definition 4 (Generated target switcher $TgtS$):
$TgtS(K_1,\ldots,K_n)$ *is the* generated target switcher *for the targets* $K_1,\ldots,K_n$*.* $TgtS(K_1,\ldots,K_n) = (S,\Sigma,\Delta,\hat{s},\Pi,val)$*, where*

- $S = \{s_0\} \cup \{s_i, s_i', s_i'' \,\|\, 1 \leq i \leq n\}$

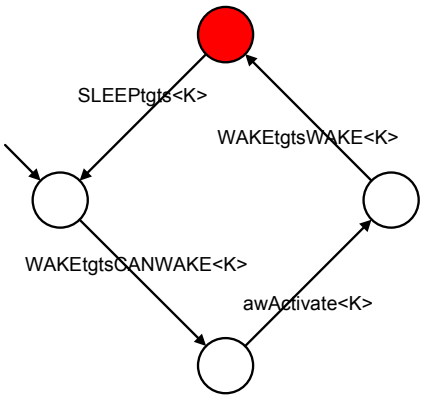

Fig. 6. *An example target switcher, generated for target K. The state colored red has the attribute* K running.

- $\Sigma = \{$"WAKEtgtsCANWAKE<"&$name(K_i)$&">", "awActivate<"&$name(K_i)$&">","WAKEtgtsWAKE<"&$name(K_i)$&">", "SLEEPtgts<"&$name(K_i)$&">","ACTIVATED<"&K_i&">" $\| \, 1 \leq i \leq n\}$
- $\Delta = \{(s,a,s') \in S \times \Sigma \times S \| \, \exists i; 1 \leq i \leq n :$
  $(s = s_0 \wedge a =$"WAKEtgtsCANWAKE<"&$name(K_i)$&">"$\wedge s' = s_i) \vee$
  $(s = s_i \wedge a =$"awActivate<"&$name(K_i)$&">"$\wedge s' = s'_i) \vee$
  $(s = s'_i \wedge a =$"WAKEtgtsWAKE<"&$name(K_i)$&">"$\wedge s' = s''_i) \vee$
  $(s = s''_i \wedge a =$"SLEEPtgts<"&$name(K_i)$&">"$\wedge s' = s_0) \vee$
  $(\exists j; 1 \leq j \leq n \wedge j \neq i :$
  $s = s''_j \wedge a =$"ACTIVATED<"&$name(K_i)$&">"$\wedge s' = s''_i)\}$
- $\hat{s} = s_0$
- $\Pi = \{name(K_i)\&$" running" $\| \, 1 \leq i \leq n\}$
- $val(s) = \{name(K_i)\&$" running" $\| \, 1 \leq i \leq n \wedge s = s''_i\}$

Definition 5 (Generated target switcher refinement $TgtS - rm$):
$TgtS - rm(K_1, \ldots, K_n)$ *is the* generated target switcher refinement *for the targets* $K_1, \ldots, K_n$. $TgtS - rm(K_1, \ldots, K_n) = (S, \Sigma, \Delta, \hat{s}, \Pi, val)$, *where*

- $S = \{s_0\} \cup \{s_i, s'_i \| \, 1 \leq i \leq n\}$

- $\Sigma = \{$“start_awActivate<”&$name(K_i)$&“>”,
  “kw_SetTarget $(OUT=”&$name(K_i)$&“.id)$”,
  “end_awActivate<”&$name(K_i)$&“>” $\| \, 1 \leq i \leq n\}$

- $\Delta = \{(s,a,s') \in S \times \Sigma \times S \| \, \exists i; 1 \leq i \leq n :$
  $(s = s_0 \wedge a =$“start_awActivate<”&$name(K_i)$&“>”$\wedge s' = s_i) \vee$
  $(s = s_i \wedge a =$“kw_SetTarget $(OUT=”&$name(K_i)$&“.id)$”$\wedge s' = s_i') \vee$
  $(s = s_i' \wedge a =$“end_awActivate<”&$name(K_i)$&“>”$\wedge s' = s_0)\}$

- $\Pi = \emptyset$

- $val(s) = \emptyset$

2.1.3 Special Semantics

Specific action names may have properties beyond what is usual for actions of their category. These include static text replacements, data access, and negated actions. Some model structures are also generated based on attributes placed in the models.

The strings “@PARENT” and “@TARGET” occurring in action names get replaced before the model is executed. “@PARENT” is replaced by the name of the application it belongs to, and “@TARGET” by the name of the target the model is assigned to. Their main use is to keep synchronizations unambiguous. For example, if a model component containing the action *ALLOWtgt<@TARGET:X>* is assigned to two different targets, the text replacement will cause it to be synchronized with a different action on each, as opposed to *ALLOWtgt<X>*, which might get synchronized with any suitable action regardless of targets.

Ordinarily the execution of a keyword is expected to succeed; a failure indicates a failed test run. However, a keyword can be *negated* by adding a “$\sim$” in front of the action name, in which case its execution is expected to fail. For example, if *kwVerifyText* checks that a specific text is visible on the display of the SUT, $\sim$*kwVerifyText* checks that the text is not visible. A special case is a situation where both ordinary and negated versions of a keyword begin from the same state. This is a *branching* keyword, which is allowed to succeed or fail. When either transition is selected for execution in the model, the keyword is sent to Test Engine for execution. If the execution succeeds, the ordinary version of the keyword is executed in the model,

otherwise the negated one. Thus, branching keywords allow the action refinements to adapt to the state of the SUT, which is useful in testing nondeterministic systems. Obviously this is only useful in online testing.

Branching can also be applied to action words. Whether the action word succeeds or fails depends on its refinement: a negated ending synchronization means a failed action word. While the effects of branching keywords are limited to within a single action word implementation, branching action words can alter the course of the whole test run. This property, while often useful, can make test generation much more difficult.

Some attributes are used to generate actions and transitions into the model components. Attributes whose names begin with "sv", so-called *state verifications*, are shorthand for simple action loops. A state containing the state verification *svX* is provided with a looping action by the same name. These are treated semantically just like action words, but can be useful as markers in test generation; for example, they might be always executed whenever encountered in order to verify the state of the SUT as often as possible. There are also *target allow* attributes of the form *taX*, which cause the generation of the action *ALLOW<X>* from all states with the *Sleep-State* attribute to the state with the target allow. An individual target allow attribute may be placed in only a single state.

2.2 Parallel Composition

Individual model components are of little use separately. To be used, they must be combined into a test model through a process called parallel composition. In our methodology parallel composition serves two purposes. First, it combines all the model components of an individual application into a unified whole, where each component acts in its specific role. Second, it combines the model components of the different applications in such a way that their actions can be interleaved, which allows effective concurrency testing. This section will explain the composition process and the parameters with which we use it.

2.2.1 Definition

The model components are combined together with process algebraic *parallel composition*. Parallel composition allows us to treat a number of model components as a single composite model. The states and transitions of the composite model are combinations of the corresponding elements of the model components. The actions of the composite model likewise correspond to those of the model components; executing an action in the composite model means executing specific actions in the model components *synchronously*.

There are many different methods of parallel composition, mostly differing on how they synchronize the actions of the model components. The version we use is based on a rule set which explicitly defines the synchronizations [22]. The formal definition is the following:

Definition 6 (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, such that* $\forall i,j; 1 \leq i < j \leq n : \Pi_i \cap \Pi_j = \emptyset$. *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* $\|_R\ (L_1,\ldots,L_n) = repa((S,\Sigma,\Delta,\hat{s},\Pi,val))$, *where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \Sigma_R$
- $((s_1,\ldots,s_n),a,(s'_1,\ldots,s'_n)) \in \Delta$ *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* $s_i = s'_i$
- $\hat{s} = (\hat{s}_1,\ldots,\hat{s}_n)$
- $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$
- $val((s_1,\ldots,s_n)) = val_1(s_1) \cup \cdots \cup val_n(s_n)$
- *repa is a function restricting LSTS to contain only the states which are reachable from the initial state* $\hat{s}$.

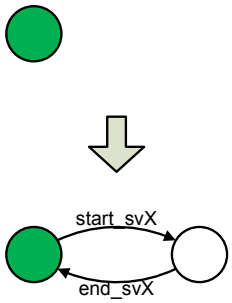

Fig. 7. *Creating state verification loops. The state marked in green has the attribute* svX.

2.2.2 Graph Transformations

We do not apply parallel composition directly on the model components, but first perform some automatic graph transformations on the action machines. The transformation rules are listed below. They are used in the given order to all applicable model structures; thus, branching action words will be handled before non-branching ones. In the rules, *A* MATCHES *B* means that *A* must match the Python regular expression [40] defined by *B*.

- Expand state verifications:

  $\forall s \in S : \forall \pi \in val(s); \pi$ MATCHES "sv.*":

  - $S \longleftarrow S \cup \{s_{s,\pi}\}$
  - $\Sigma \longleftarrow \Sigma \cup \{\text{“start\_”}\&\pi, \text{“end\_”}\&\pi\}$
  - $\Delta \longleftarrow \Delta \cup \{(s, \text{“start\_”}\&\pi, s_{s,\pi}), (s_{s,\pi}, \text{“end\_”}\&\pi, s)\}$

  States with state verifications are provided with corresponding two-part loops. In the composed model the refinement for the state verification will appear between the two parts (Figure 7).

- Split branching action words:

  $\forall a \in \Sigma; a$ MATCHES "aw.*": $\forall s, s', s'' \in S; (s, a, s') \in \Delta \wedge (s, \text{“}\sim\text{”}\&a, s'') \in \Delta :$

  - $S \longleftarrow S \cup \{s_{s,a}\}$
  - $\Sigma \longleftarrow \Sigma \cup \{\text{“start\_”}\&a, \text{“end\_”}\&a, \text{“}\sim\text{end\_”}\&a\}$

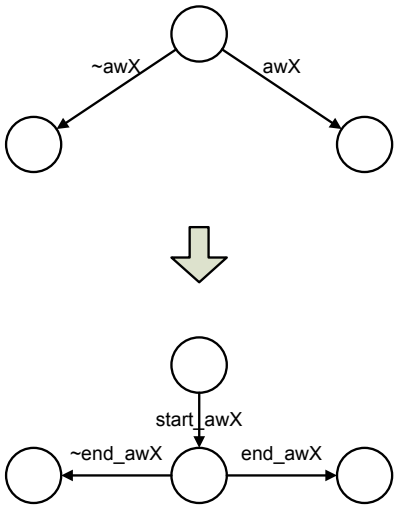

Fig. 8. *Splitting branching action words.*

– $\Delta \longleftarrow \Delta - \{(s,a,s'),(s,\text{“}\sim\text{”}\&a,s'')\} \cup$
$\{(s,\text{“start\_”}\&a,s_{s,a}),(s_{s,a},\text{“end\_”}\&a,s'),(s_{s,a},\text{“}\sim\text{end\_”}\&a,s'')\}$

Branching action words are split into two parts, the first of which is shared between both branches (Figure 8). The old actions are not removed, but will not pass through parallel composition.

- Split other action words:

  $\forall(s,a,s') \in \Delta; a$ MATCHES “aw.*”:

  – $S \longleftarrow S \cup \{s_{s,a}\}$
  – $\Sigma \longleftarrow \Sigma \cup \{\text{“start\_”}\&a,\text{“end\_”}\&a\}$
  – $\Delta \longleftarrow \Delta - \{(s,a,s')\} \cup \{(s,\text{“start\_”}\&a,s_{s,a}),(s_{s,a},\text{“end\_”}\&a,s')\}$

  The remaining action words are split into two parts (Figure 9).

- Split *WAKEts* transitions:

  $\forall(s,a,s') \in \Delta; a =$“WAKEts”:

  – $S \longleftarrow S \cup \{s_{s,a}\}$
  – $\Sigma \longleftarrow \Sigma \cup \{\text{“WAKEtsCANWAKE”},\text{“WAKEtsWAKE”}\}$

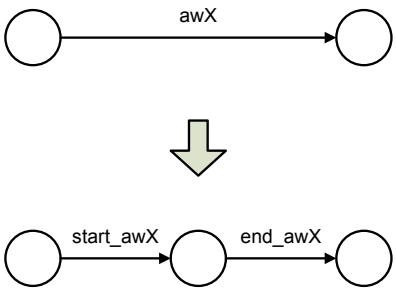

Fig. 9. *Splitting nonbranching action words.*

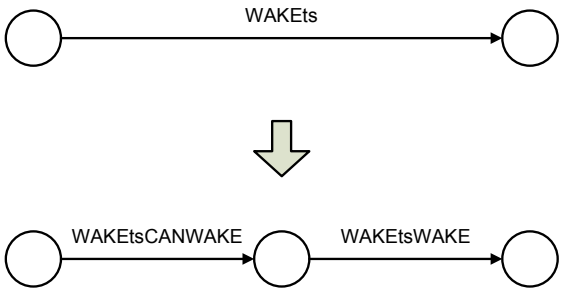

Fig. 10. *Splitting* WAKEts *transitions.*

– $\Delta \longleftarrow \Delta - \{(s,a,s')\} \cup$
$\{(s,\text{“WAKEtsCANWAKE”},s_{s,a}),(s_{s,a},\text{“WAKEtsWAKE”},s')\}$

The *WAKEts* transitions are split into two parts. In the composed model the activation sequence of the model component, as defined in the task switcher, will appear between the two parts (Figure 10).

- Expand target allows:

$\forall s,s' \in S;$“SleepState”$\in val(s)$:
$\forall x; x$ MATCHES “.*”: $\forall \pi \in val(s'); \pi =$“ta”&$x$:

– $\Sigma \longleftarrow \Sigma \cup \{\text{“ALLOW<”\&}x\text{\&“>”}\}$
– $\Delta \longleftarrow \Delta \cup \{(s,\text{“ALLOW<”\&}x\text{\&“>”},s')\}$

New allow transitions from sleeping states into a specifically marked state are created (Figure 11).

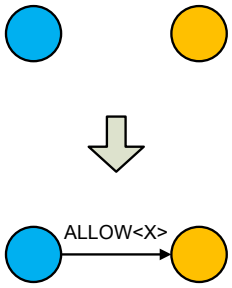

Fig. 11. *Creating* ALLOW *transitions for the target allow attribute. The state marked in orange has the attribute* taX.

2.2.3 Rules

In theory, the ability to explicitly define the synchronizations offers great flexibility. In practice, though, defining the rules separately for each action in the model components would not be worth the effort, and would make understanding the models much more difficult. Because of these, the rule set is generated automatically based on the actions occurring in the model components.

These *generation rules* are given below. The formulae describe the generation of rules R for the components $L_1,\ldots,L_n$, $L_i = (S_i,\Sigma_i,\Delta_i,\hat{s}_i,\Pi_i,val_i)$. The name of an individual component L is referred to with $name(L)$.

In the verbal explanations and figures X is used to mark arbitrary character strings in action names, A and B are names for action machines, A-*rm* is a name for a refinement machine, and K and L are names for test targets. The given explanations may be narrower than the formal rules; in these cases they signify the intended way to use the actions in question. The figures likewise describe a typical use of the rule in question. In the figures, transition colors other than black are used to signify synchronously executed actions, and dashed transitions mean sequences of actions irrelevant to the pertinent synchronization.

The composition rules are generated in two phases. First the following generation rules are applied to the model components belonging to an individual target:

- Action words and state verifications:

  $\forall i,j;1 \leq i,j \leq n \wedge name(L_j)$ MATCHES $name(L_i)$&“-rm.*”:
  $\forall s;s$ MATCHES “(start|∼?end).*”$\wedge s \in \Sigma_i \cap \Sigma_j$:

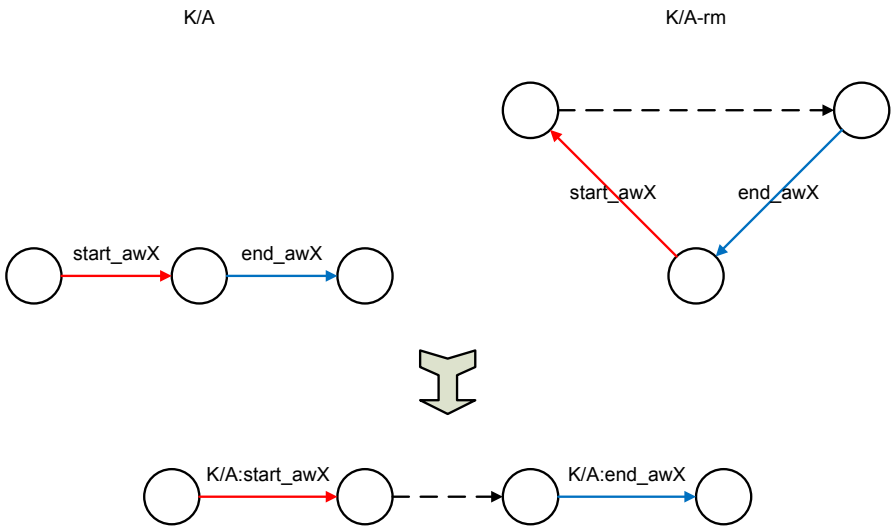

Fig. 12. *Action word synchronization.*

$\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R :$
$\sigma_i = \sigma_j = s \wedge \sigma_R = name(L_i)\&\text{“:”}\&s \wedge \forall k; 1 \leq k \leq n \wedge k \neq i \wedge k \neq j : \sigma_k = \surd$

The split action words and state verifications *startX*, *endX* and $\sim$*endX* in action machine *A* are executed synchronously with actions of the same name in a refinement machine *A-rm* (Figures 12 and 13). Refinements for the actions of a single action machine may be placed into several refinement machines.

- Keywords:

  $\forall i; 1 \leq i \leq n : \forall s; s$ MATCHES “$\sim$?(kw|vw).*”$\wedge s \in \Sigma_i :$
  $\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R : \sigma_i = \sigma_R = s \wedge \forall j; 1 \leq j \leq n \wedge j \neq i : \sigma_j = \surd$

  The keywords *kwX*, *vwX*, $\sim$*kwX* and $\sim$*vwX* in the refinement machine *R* are executed alone (Figure 14).

- Task switcher synchronizations:

  $\forall i, j; 1 \leq i, j \leq n \wedge name(L_i)$ MATCHES “TaskSwitcher.*”:
  $\forall s; s$ MATCHES “SLEEPts|WAKEtsCANWAKE|WAKEtsWAKE”$\wedge$
  $s\&\text{“<”}\&name(L_j)\&\text{“>”} \in \Sigma_i \wedge s \in \Sigma_j :$
  $\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R : \sigma_i = \sigma_R = s\&\text{“<”}\&name(L_j)\&\text{“>”} \wedge \sigma_j = s \wedge$
  $\forall k; 1 \leq k \leq n \wedge k \neq i \wedge k \neq j : \sigma_k = \surd$

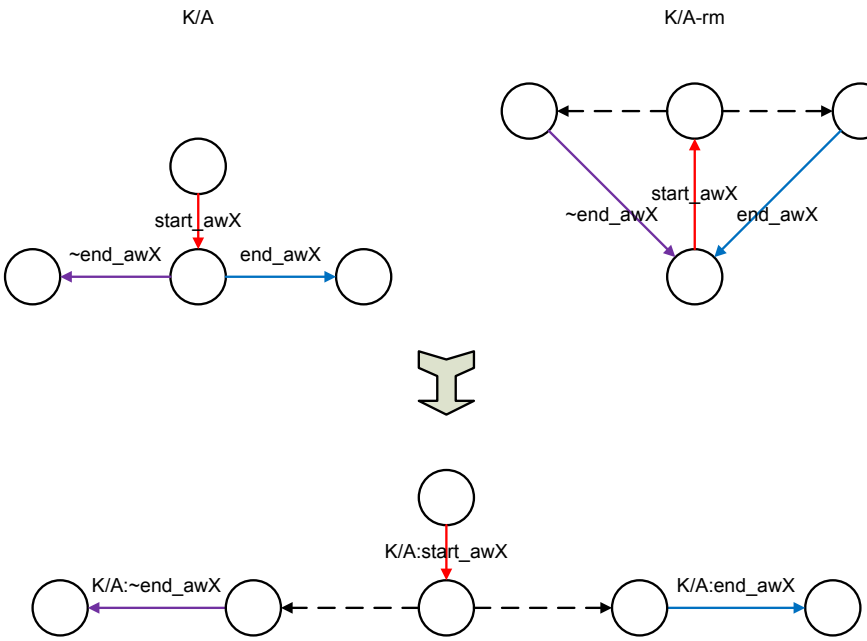

Fig. 13. *Branching action word synchronization.*

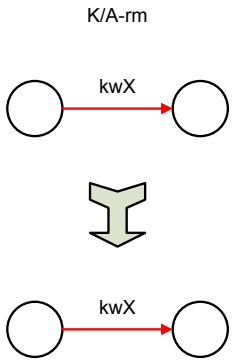

Fig. 14. *Keyword synchronization.*

The task switcher synchronizations are used to activate and deactivate model components through the task switcher. *WAKEtsCANWAKE<A>* in the task switcher is executed synchronously with *WAKEtsCANWAKE* in action machine *A*; *WAKEtsWAKE* and *SLEEPts* function in a similar way (Figure 15).

- Activation synchronizations:

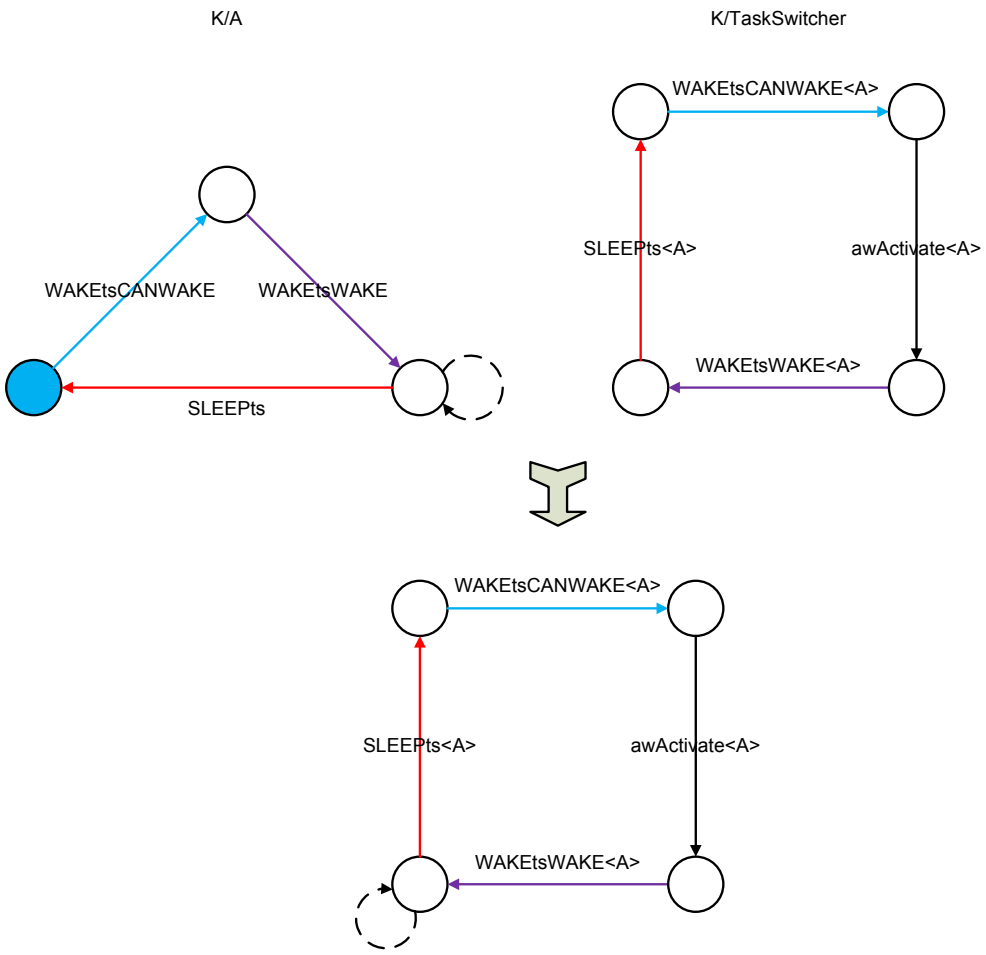

Fig. 15. *Task switcher synchronizations.*

$\forall i,j,k; 1 \leq i,j,k \leq n \wedge name(L_k)$ MATCHES "TaskSwitcher.*"$\wedge$
"ACTIVATED<"&$name(L_j)$&">"$\in \Sigma_k$:
$\forall s$;"SLEEPapp<"&s&">"$\in \Sigma_i \wedge$"WAKEapp<"&s&">"$\in \Sigma_j$:
$\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R : \sigma_i =$"SLEEPapp<"&$s$&">"$\wedge$
$\sigma_j =$"WAKEapp<"&s&">"$\wedge\sigma_k =$"ACTIVATED<"&$name(L_j)$&">"
$\sigma_R = name(L_i)$&" ACTIVATES "&$name(L_j)$&": "&$s \wedge$
$\forall l; 1 \leq l \leq n \wedge l \neq i \wedge l \neq j \wedge l \neq k : \sigma_l = \surd$

The activation synchronizations switch control directly between action machines of the same target. *SLEEPapp<X>* in action machine *A* is executed synchronously with *WAKEapp<X>* in action machine *B*, with the task

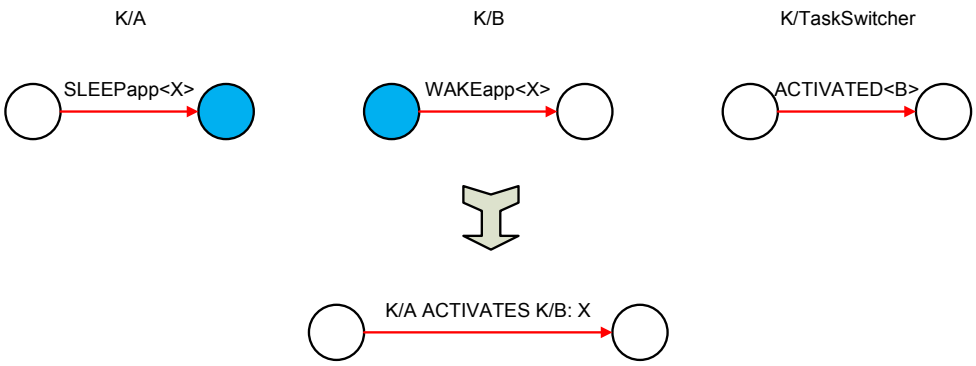

Fig. 16. *Activation synchronization.*

switcher keeping track of the currently active action machine by executing *ACTIVATED<B>* (Figure 16).

- Request synchronizations:

  – $\forall i,j;1 \leq i,j \leq n : \forall s;$“REQ<”&$s$&“>”$\in \Sigma_i \wedge$“ALLOW<”&$s$&“>”$\in \Sigma_j$:
  $\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R : \sigma_i =$“REQ<”&$s$&“>”$\wedge$
  $\sigma_j =$“ALLOW<”&s&“>”$\wedge$
  $\sigma_R = name(L_j)$&“ ALLOWS ”&$name(L_i)$&“: ”&$s \wedge$
  $\forall k;1 \leq k \leq n \wedge k \neq i \wedge k \neq j : \sigma_k = \surd$

  – $\forall i;1 \leq i \leq n : \forall s;$“REQALL<”&$s$&“>”$\in \Sigma_i$:
  $\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R : \sigma_i =$“REQALL<”&$s$&“>”$\wedge$
  $\sigma_R = name(L_i)$&“ WAS ALLOWED: ”&$s \wedge$
  $\forall j;1 \leq j \leq n \wedge j \neq i :$
  (“ALLOW<”&s&“>”$\in \Sigma_j \rightarrow \sigma_j =$“ALLOW<”&$s$&“>”) $\wedge$
  (“ALLOW<”&s&“>”$\notin \Sigma_j \rightarrow \sigma_j = \surd$)

The request synchronizations allow action machines to enquire or change the states of other action machines of the same target without switching control to them. *REQ<X>* in one action machine is executed synchronously with *ALLOW<X>* in another (Figure 17). *REQALL<X>* is executed synchronously with *ALLOW<X>* in all other action machines whose set of actions contains it (Figure 18). In particular, if there are no action machines with action *ALLOW<X>*, *REQALL<X>* is executed alone.

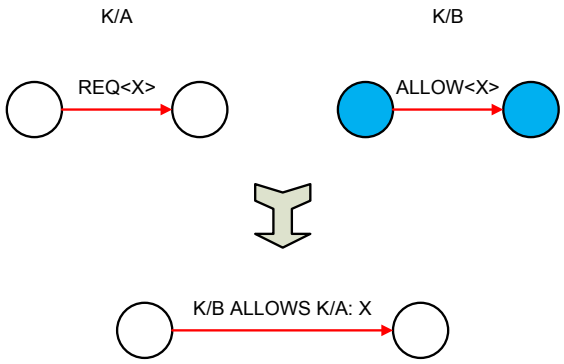

Fig. 17. *Request synchronization.*

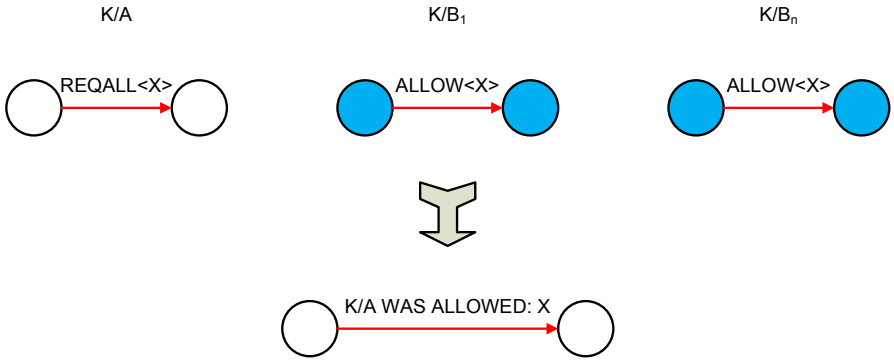

Fig. 18. *Request all synchronization.*

- Comments:

  $\forall i; 1 \leq i \leq n : \forall s; \text{“--”} \& s \in \Sigma_i :$
  $\exists(\sigma_1, \ldots, \sigma_n, \sigma_R) \in R : \sigma_i = \text{“--”} \& s \wedge \sigma_R = name(L_i) \& \text{“:--”} \& s \wedge$
  $\forall j; 1 \leq j \leq n \wedge j \neq i : \sigma_j = \surd$

  Comments $--X$ are executed alone (Figure 19).

Once all the rules for individual targets have been generated, the second phase generates the rules for synchronizations between the targets. At this phase, the model components of different targets are identified by prefixing their names with the name of the target separated with a slash. The target switcher and synchronizer do not belong to any target, and the latter thus requires a separate version of some rules.

- Target switcher synchronizations:

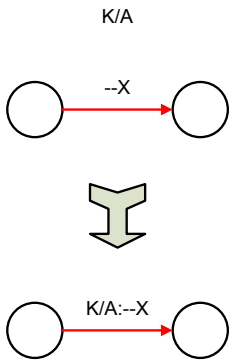

Fig. 19. *Comment synchronization.*

$\forall t : \forall i,j; 1 \leq i,j \leq n \wedge name(L_i)$ MATCHES "TargetSwitcher.*"$\wedge$
$name(L_j)$ MATCHES t&"/TaskSwitcher.*":
$\forall s; s$ MATCHES
"(SLEEPtgts|WAKEtgtsCANWAKE|WAKEtgtsWAKE)<"&t&">"$\wedge$
$s \in \Sigma_i \cap \Sigma_j$:
$\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R : \sigma_i = \sigma_j = \sigma_R = s \wedge \forall k; 1 \leq k \leq n \wedge k \neq i \wedge k \neq j : \sigma_k = \surd$

The target switcher synchronizations *WAKEtgtsCANWAKE*, *WAKEtgtsWAKE* and *SLEEPtgts* connect the target switcher to the task switchers. *WAKEtgtsCANWAKE*<*K*> in the target switcher is executed synchronously with the action of the same name in the task switcher of *K*; *WAKEtgtsWAKE* and *SLEEPtgts* function in a similar way (Figure 20).

- Target activation synchronizations:

  – $\forall t,t'; t \neq t' : \forall i,j,k,l,m; 1 \leq i,j,k,l,m \leq n \wedge$
  $name(L_i)$ MATCHES t&"/.*"$\wedge name(L_j)$ MATCHES t'&"/.*"$\wedge$
  $name(L_k)$ MATCHES t&"/TaskSwitcher.*"$\wedge$
  $name(L_l)$ MATCHES t'&"/TaskSwitcher.*"$\wedge$
  $name(L_m)$ MATCHES "TargetSwitcher.*":
  $\forall s$;"SLEEPtgt<"&s&">"$\in \Sigma_i \wedge$"WAKEtgt<"&s&">"$\in \Sigma_j \wedge$
  "TARGET_DEACTIVATED"$\in \Sigma_k \wedge$
  "TARGET_ACTIVATED<"&$name(L_j)$&">"$\in \Sigma_l \wedge$
  "ACTIVATED<"&t'&">"$\in \Sigma_m$:
  $\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R : \sigma_i =$"SLEEPtgt<"&$s$&">"$\wedge$

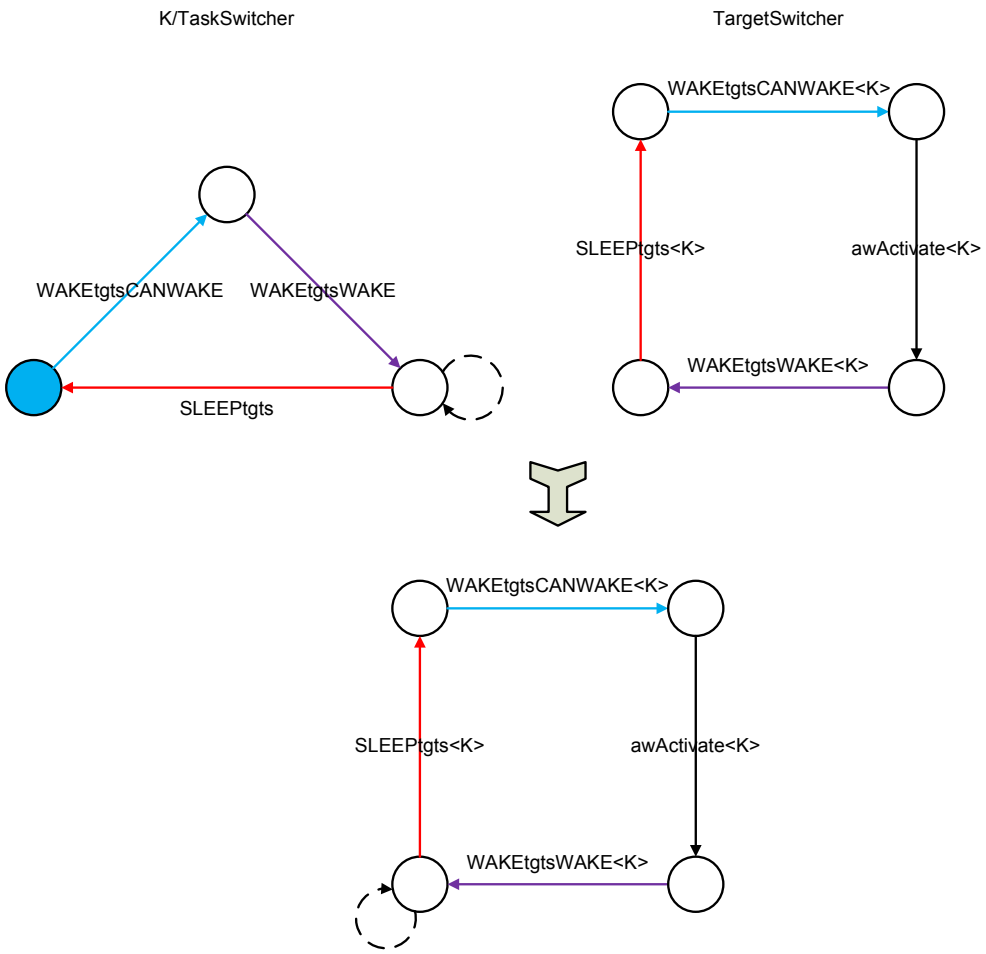

Fig. 20. *Target switcher synchronizations.*

$\sigma_j =$“WAKEtgt<”&s&“>”$\wedge\sigma_k =$“TARGET_DEACTIVATED”$\wedge$
$\sigma_l =$“TARGET_ACTIVATED<”&$name(L_j)$&“>”$\wedge$
$\sigma_m =$“ACTIVATED<”&t'&“>”$\wedge\sigma_R = t$&“ ACTIVATES ”&t'&“: ”&$s\wedge$
$\forall p; 1 \leq p \leq n \wedge p \neq i \wedge p \neq j \wedge p \neq k \wedge p \neq l \wedge p \neq m : \sigma_p = \surd$

– $\forall t : \forall i, j, k; 1 \leq i, j, k \leq n \wedge name(L_i)$ MATCHES t&“/.*”$\wedge$
$name(L_j)$ MATCHES t&“/.*”$\wedge$
$name(L_k)$ MATCHES t&“/TaskSwitcher.*”:
$\forall s;$“SLEEPtgt<”&s&“>”$\in \Sigma_i \wedge$“WAKEtgt<”&s&“>”$\in \Sigma_j \wedge$
“ACTIVATED<”&$name(L_j)$&“>”$\in \Sigma_k$:
$\exists(\sigma_1, \ldots, \sigma_n, \sigma_R) \in R : \sigma_i =$“SLEEPtgt<”&$s$&“>”$\wedge$

$\sigma_j =$"WAKEtgt<"&s&">"$\wedge\sigma_k =$"ACTIVATED<"&$name(L_j)$&">"$\wedge$
$\sigma_R = name(L_i)$&" ACTIVATES "&$name(L_j)$&": "&$s\wedge$
$\forall l; 1 \leq l \leq n \wedge l \neq i \wedge l \neq j \wedge l \neq k : \sigma_l = \surd$

– $\forall t : \forall i, j, k; 1 \leq i, j, k \leq n \wedge name(L_i)$ MATCHES t&"/.*"$\wedge$
$name(L_j) =$"Synchronizer"$\wedge$
$name(L_k)$ MATCHES t&"/TaskSwitcher.*":
$\forall s;$"SLEEPtgt<"&s&">"$\in \Sigma_i \wedge$"WAKEtgt<"&s&">"$\in \Sigma_j \wedge$
"TARGET_DEACTIVATED"$\in \Sigma_k$:
$\exists(\sigma_1, \ldots, \sigma_n, \sigma_R) \in R : \sigma_i =$"SLEEPtgt<"&$s$&">"$\wedge$
$\sigma_j =$"WAKEtgt<"&s&">"$\wedge\sigma_k =$"TARGET_DEACTIVATED"$\wedge$
$\sigma_R = t$&" ACTIVATES Synchronizer: "&$s\wedge$
$\forall l; 1 \leq l \leq n \wedge l \neq i \wedge l \neq j \wedge l \neq k : \sigma_l = \surd$

– $\forall t : \forall i, j, k; 1 \leq i, j, k \leq n \wedge name(L_i)$ MATCHES "Synchronizer"$\wedge$
$name(L_j)$ MATCHES t&"/.*"$\wedge$
$name(L_k)$ MATCHES t&"/TaskSwitcher.*":
$\forall s;$"SLEEPtgt<"&s&">"$\in \Sigma_i \wedge$"WAKEtgt<"&s&">"$\in \Sigma_j \wedge$
"TARGET_ACTIVATED<"&$name(L_j)$&">"$\in \Sigma_k$:
$\exists(\sigma_1, \ldots, \sigma_n, \sigma_R) \in R : \sigma_i =$"SLEEPtgt<"&$s$&">"$\wedge$
$\sigma_j =$"WAKEtgt<"&s&">"$\wedge$
$\sigma_k =$"TARGET_ACTIVATED<"&$name(L_j)$&">"$\wedge$
$\sigma_R =$"Synchronizer ACTIVATES "&t&": "&$s\wedge$
$\forall l; 1 \leq l \leq n \wedge l \neq i \wedge l \neq j \wedge l \neq k : \sigma_l = \surd$

The target activation synchronizations switch control between action machines which may be on different targets. They always synchronize *SLEEPtgt<X>* in action machine *A* on target *K* and *WAKEtgt<X>* in action machine *B* on target *L*. If *K* and *L* are the same target, its task switcher executes *ACTIVATED<B>*, just as in an activation synchronization. If *K* and *L* are different targets, the task switcher of *K* executes *TARGET_DEACTIVATED*, the task switcher of *L* executes *TARGET_ACTIVATED<B>*, and the target switcher executes *ACTIVATED<L>* (Figure 21). Finally, the synchronizer requires its own versions of these generation rules since it does not belong to any target. The same also applies to the target switcher, but it does not need activation synchronizations.

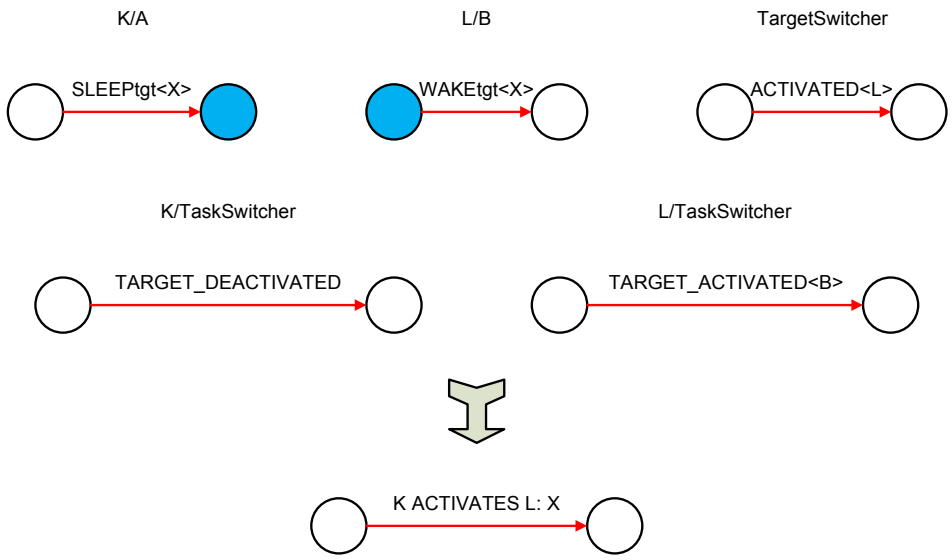

Fig. 21. *Target activation synchronization.*

- Target request synchronizations:

  – $\forall i,j;1 \leq i,j \leq n:$
  $\forall s;$“REQtgt<”&s&“>”$\in \Sigma_i \wedge$“ALLOWtgt<”&s&“>”$\in \Sigma_j:$
  $\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R: \sigma_i =$“REQtgt<”&$s$&“>”$\wedge$
  $\sigma_j =$“ALLOWtgt<”&s&“>”$\wedge$
  $\sigma_R = name(L_j)$&“ ALLOWS ”&$name(L_i)$&“: ”&$s \wedge$
  $\forall k;1 \leq k \leq n \wedge k \neq i \wedge k \neq j: \sigma_k = \surd$

  – $\forall i;1 \leq i \leq n: \forall s;$“REQALLtgt<”&$s$&“>”$\in \Sigma_i:$
  $\exists(\sigma_1,\ldots,\sigma_n,\sigma_R) \in R: \sigma_i =$“REQALLtgt<”&$s$&“>”$\wedge$
  $\sigma_R = name(L_i)$&“ WAS ALLOWED: ”&$s \wedge$
  $\forall j;1 \leq j \leq n \wedge j \neq i:$
  (“ALLOWtgt<”&s&“>”$\in \Sigma_j \rightarrow \sigma_j =$“ALLOWtgt<”&$s$&“>”$) \wedge$
  (“ALLOWtgt<”&s&“>”$\notin \Sigma_j \rightarrow \sigma_j = \surd$)

The target request synchronizations *REQtgt<X>*, *REQALLtgt<X>* and *ALLOWtgt<X>* transmit information between action machines which may be on different targets (Figures 22 and 23). Apart from that, they function just like request synchronizations.

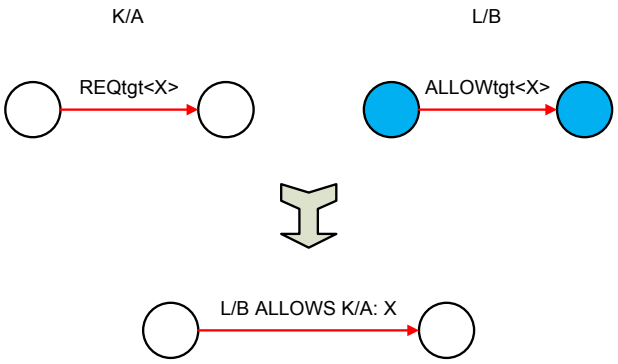

Fig. 22. *Target request synchronization.*

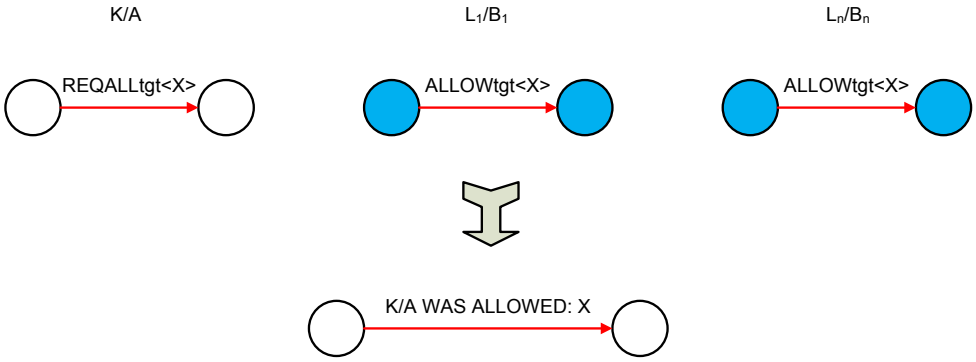

Fig. 23. *Target request all synchronization.*

No rules other than the ones described above are generated. Figure 24 shows part of a test model composed from the action machines for the main and inbox screens of the Messaging application (Figures 2 and 3), the appropriate task and target switchers (Figures 5 and 6), and the corresponding refinement machines (the one belonging to the Main model shown in Figure 4).

2.3 Test Data

LSTSs are not very well suited for modeling data. Although data values can be encoded into action names, they are difficult to change during testing and can clutter the models. Also, using states to keep track of variable values may easily lead into a serious state explosion problem [48]. Because of this, we have other means of

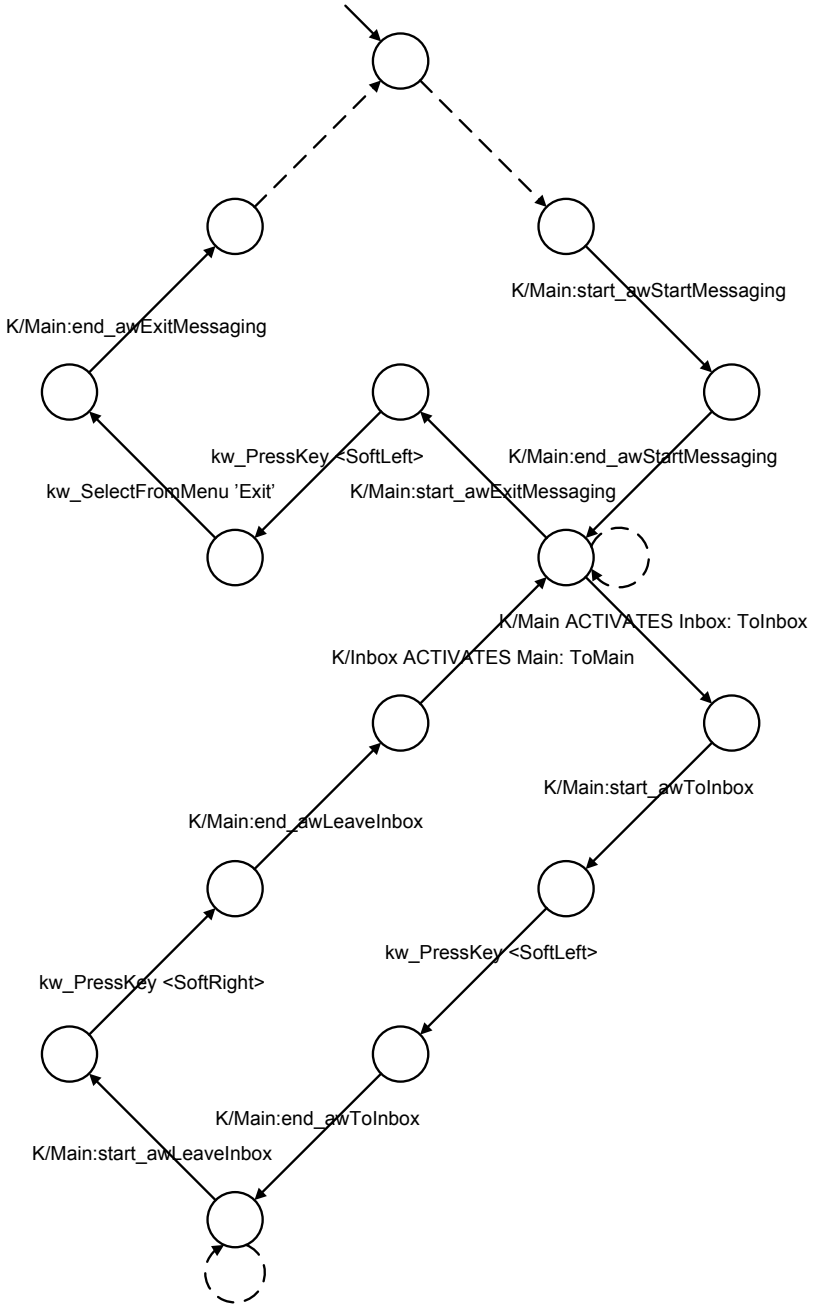

Fig. 24. *An example of a composed test model. The parts related to the task and target switchers have been abstracted away to keep the model understandable, and are represented with dashed lines. Attributes are also not marked.*

modeling data: localization tables and data tables.

2.3.1 Localization Tables

The simpler form of data is *localization tables*. They consist of a list of symbolic names and one of *locales*, with a text string defined for each name in each locale. The symbolic names can be referred to in action names (usually keywords) with "§SYMBOLIC_NAME§". When the action is executed, the name gets replaced with the corresponding text for the appropriate locale. Currently our tools require the locale to be defined when the test run is begun, but ideally the locale could be also changed during the test run.

As the name implies, the main use for localization tables is to enable testing with different language variants without altering the models. Localization tables are associated with specific products, just as refinement machines. This makes them useful also for defining product-specific constants, such as the coordinates of a specific button in some application.

2.3.2 Data Tables

More complex data, such as complete contact information, requires the use of *data tables*. Data tables contain a structural definition and a list of elements matching that definition. For example, the table

$$\begin{aligned}&\text{contacts}(\text{firstName}, \text{lastName}, \text{phoneNumber}) : \backslash \\ &\qquad [(\text{“John”}, \text{“Doe”}, 12345678), (\text{“Jane”}, \text{“Doe”}, 87654321)]\end{aligned}$$

defines that each element has the fields firstName, lastName and phoneNumber, and lists two such elements. A single element of the table is selected at any moment; accessing and changing the selected element is performed via data statements.

Data statements are parts of action names of the form "$(data statement)$". The statement consists of Python [39] code, namely a list of simple statements, which are executed before the action they are a part of. If the execution of the data statement results in a variable named "OUT", its value will be substituted for the statement in the action name; otherwise the statement is replaced with an empty string.

The currently selected element of a data table may be accessed by the name of the table and its substructure by the names of its fields. The selected element can be changed with the functions *first*, *next* and *any*, which select the first element, the element following the currently selected one (wrapping around if necessary), and an arbitrary element, respectively. As an example of using a data statement, the action *kw_Type '$(next(contacts); OUT = contacts.firstName)$'* selects the next element of the contacts table and types the first name of the newly selected contact.

Data tables and statements can also be used to perform more complex variation of data, such as the testing of boundary values. Suitable values can be placed into a data table, or generated on the fly in a data statement. Likewise, they can be used to test real-time properties by using data statements to store and compare time stamps.

3. MODELING

In this chapter we will examine the modeling techniques involved in the creation of model components. The components are assembled into a model library, from which some of them can be picked for composition into a test model. Thus, it is necessary to ensure that they function correctly both together and separately. We will first examine the techniques for avoiding name clashes, and continue into the features required by coverage requirements and test generation. We will also present guidelines for dividing functionality into model components, and for making those components usable on several different products. Finally, we will take a look at a model library created during the research.

3.1 Avoiding Name Clashes

Some basic rules are necessary simply to avoid name clashes. While a modeler could come up with suitable names as he works, it is better to have a consistent guideline. This is especially important if there are several modelers.

Most important naming rules concern synchronization labels. Differentiating the labels of different applications with "@PARENT" is a good beginning; the rest of the label should make clear what the synchronization does within the application. As an example, *SLEEPapp<@PARENT:ToMain>* might activate the main model component of the application in question. Synchronizations between different applications obviously cannot use "@PARENT", and therefore need otherwise unambiguous labels.

Any Python variables created within data statements need likewise unique names. This is not strictly necessary for a variable which is only needed during the execution of a single action word, since interleavings are not possible until the execution is finished. However, anything stored for a longer term is at risk. Since all model com-

ponents share the same Python environment, the risk of name clashes comes not only from different applications on the same device, but also from the same application and model components on different devices.

A unique identifier for a variable can be created by including both "@TARGET" and "@PARENT" in it. Unfortunately, they cannot be used directly to name a variable, because they may contain whitespace or other characters illegal in a variable name. A simple method is to use the unique identifier as a key to the *locals* dictionary, for example "$(locals()['@TARGET@PARENTVariable'] = value)$". More elegant solutions are possible with some preparation.

3.2 Supporting Coverage Requirements

In our approach, tests are often generated according to *coverage requirements*, which are explained in detail in Subsection 4.3.2. A coverage requirement describes the goal of the test in terms of specific structures in the models. The structures most commonly used for this purpose are actions and attributes, the labels for transitions and states, respectively. This is because the labels can be made readable to human testers, whereas states and transitions themselves can usually be uniquely identified only with numerical information. Thus, it is important that actions and attributes are given clear names; specifically, their purpose should be understandable given only their name and the name of the model component they are a part of.

Even the most expressive label names cannot always convey all the necessary information to the testers, or to other modelers for that matter. Because of this, Model Designer also offers the option to write comments to the actions; the comments are visible in Web GUI during the creation of a test configuration. In comments the modeler may give more details about the intended use of the action as needed. Finally, actions and attributes can be designated as interesting. The distinction is somewhat arbitrary, but as a rule of thumb, ordinary use cases should be definable in terms of interesting labels. For example, essential actions such as launching an application or opening a received message are interesting, whereas incidental ones such as scrolling down a list of items are not. A tester using Web GUI may choose to view only the interesting labels, which can make the creation of a coverage requirement easier.

Coverage requirements are meant to be freely combinable with the operators de-

scribed in Subsection 4.3.2. However, even though requirements cannot be inherently contradictory, it is in principle possible that the execution of one would necessarily make the execution of another impossible. To prevent this from happening, the test model is required to be *strongly connected*, meaning that all of its states must be reachable from all other states. This ensures that all combinations of executable coverage requirements are also executable; if nothing else, the model may be taken back to the initial state after one coverage requirement is finished and the execution of the next one begun from there.

3.3 Supporting Test Generation

Strong connectivity is important in test generation, too. Even if a coverage requirement is executable in principle, finding a way to execute it in a model not strongly connected can be extremely difficult. This is because executing any action in such a model may lead to a part of the model from where there is no return; thus, no action may be safely executed unless it is known to leave the coverage requirement still fulfillable. In practice, this would make online testing impossible, since a safe route to the very end of the test would have to be computed before anything could be executed. With a strongly connected test model these problems do not exist, because no action has irreversible effects. The guidance algorithm may therefore safely concentrate on one part of the coverage requirement at a time, perform random deviations, or guess the best direction for the test.

While strong connectivity ensures that the guidance algorithm cannot irreversibly ruin the test run, other features of the model can still make reaching the designated goals difficult. Branching actions are a notable case, since the guidance algorithm cannot reliably predict where their execution will lead. Figure 25 illustrates a particularly problematic structure, where algorithms may easily get stuck forever.

The simplest solution is to store the information the branching action depends on into a *memory model*. Memory models such as the one in Figure 26 are model components with a specific structure designed to store just such information. Figure 27 shows how a memory model can be used in practice. The memory model effectively tells the guidance algorithm what must be done in order to reach the desired part of the model. However, as useful as they are, memory models cannot solve all problems caused by branching. We have yet to develop a truly general solution, but it will most

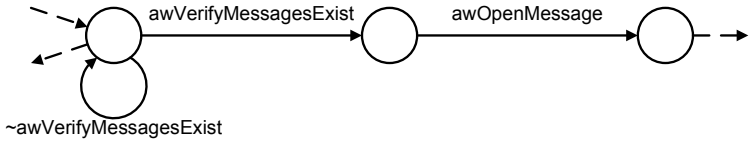

Fig. 25. *A problematic structure with the branching action* awVerifyMessagesExist*, which will fail every time if no messages exist. In that case an optimistic algorithm seeking to execute* awOpenMessage *ends up checking the existence of messages repeatedly, rather than creating a new message.*

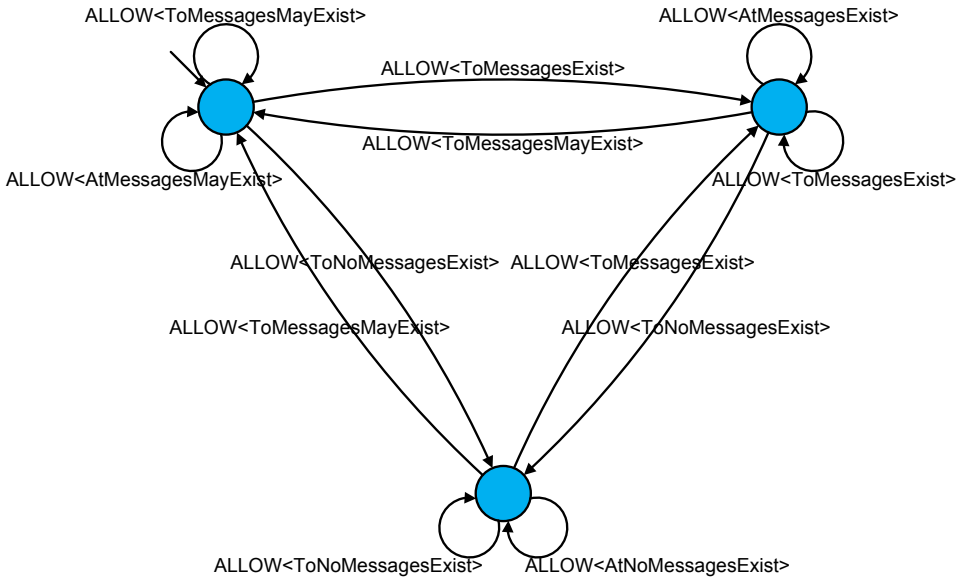

Fig. 26. *A memory model with the potential values* MessagesExist, NoMessagesExist *and* MessagesMayExist*. The third one reflects an unknown situation, and acts as the initial value.*

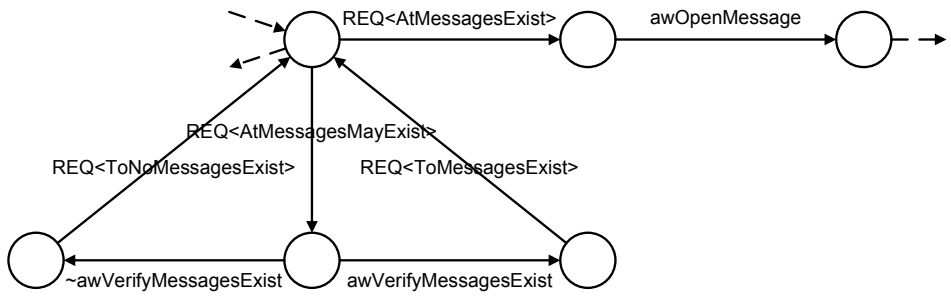

Fig. 27. *A better solution to the situation shown in Figure 25, making use of the memory model in Figure 26. The existence of messages can only be checked when unknown, and becomes known after checking. This forces the guidance algorithm to actually create a message.*

likely involve both improvements to guidance algorithms and new requirements for models.

3.4 Dividing Functionality into Components

One of the most important design choices in the creation of a test model library is the division of the modeled functionality into model components. A naive approach would be to create a component for each application. However, in practice most applications are far too complicated to handle as a single entity, and must therefore be divided into several model components. These components can be connected to each other by means of activation synchronizations.

A good starting point in dividing an application into model components is to create a separate component for each important view of the application. For example, in Messaging these could be Main, Inbox, Create SMS and Create MMS (SMS and MMS stand for Short Message Service and Multimedia Messaging Service, and refer to text messages and multimedia messages, respectively). For Messaging, we also need a memory model, Messages, to keep track of whether there are any messages in the inbox.

It may also make sense to separate into its own model component any functionality that is especially complicated or used in several places. In messaging, such functionality includes the sending and reception of messages and finding a recipient from the

list of contacts; these can be modeled with components Sender, Receiver and Add Recipient. All three contain complex synchronization sequences, which would significantly impair the readability of the core models. As separate components, Sender and Add Recipient can also be easily accessed from both Create SMS and Create MMS, and thus their functionality does not need to be modeled twice.

It is not always entirely clear to which model component an individual action belongs. For example, is the action word *awToInbox*, which opens the inbox, part of the Main model (since it is executed in the main view) or the Inbox model (since it activates the inbox view)? It is generally preferable to place such an action into the model component to whose functionality it is more closely related, in this example Inbox. For one thing, it ensures that if Inbox is left out of the composed test model, *awToInbox* will also be left out, which is probably as desired. On the other hand, this approach may in some cases lead to more complicated models, so the modeler should use his judgment.

3.5 Modeling for Multiple Products

If action machines are designed properly, they can be used in multiple products without any changes. For example, creating an SMS consists of designating the receiver and writing the message text, no matter what kind of device is used to do it. Taking advantage of this property can save a lot of work in modeling and maintenance.

In general, designing action machines usable on multiple products is not difficult; the most important thing to remember is to not include references to the UI, but only to the functionality accessed through it. On the other hand, it is perfectly possible to include action words for functionality that is not implemented on all products. They can be left without implementation in the corresponding refinement machine, which will prevent their execution in the model. Of course, care should be taken that their loss does not break the strong connectivity of the model.

Sometimes the differences between products are not limited to the UI, but extend into the functionality. For example, Messaging might offer audio messages on one product but not on another. Such issues can often be solved with the use of unimplemented or branching action words. However, such solutions tend to make the models less readable. The need for them can also be difficult to anticipate, leading to the

need to modify existing action machines each time a new product is introduced.

At some point, it becomes more sensible to create entirely different action machines rather than trying to refit one to serve two purposes. Where this point occurs depends on tool support. With poor tools, the only way to introduce a choice of action machines may be to create two entirely separate versions of the whole application, which should obviously be the last resort. On the other hand, the task can be quite simple with proper tools. For example, in Model Designer it is easy to include an action machine in the application for some products only; this way, multiple versions can be created and a suitable one chosen for each product. In such a case several simple action machines are probably preferable to a single complicated one.

3.6 Example

The first model library designed using these semantics and techniques was created during the TEMA project [P2]. It focused on modeling the applications of a product of the S60 product family. It was mostly created over three months of time by the author. The library is available from the TEMA website [44] under the MIT Open Source License.

The model library contains 13 applications: Bluetooth, Calendar, Contacts, File Manager, Gallery, Log, Messaging, Music Player, Notes, RealPlayer, Rotate, Telephony and Voice Recorder. The thoroughness of modeling varies; the Gallery models contain most of the functionality of the application, while the Bluetooth models only offer the option to turn Bluetooth on or off.

The applications are modeled in about 110 action machines, which contain approximately 1700 actions, 1300 states and 3200 transitions. The corresponding refinement machines contain roughly 3000 actions, 3000 states and 4100 transitions. The estimated number of reachable states in a test model composed from all of the components on a single device would be somewhere around 10^{19} states.

The model library had success in finding bugs from the tested products, which were already in the mass markets at the time. About two thirds of the issues were found during the careful examination of the SUT in the modeling phase, with the remaining third found during test execution. Incidentally, we also found several bugs in the adapter framework we used for automated execution [P6].

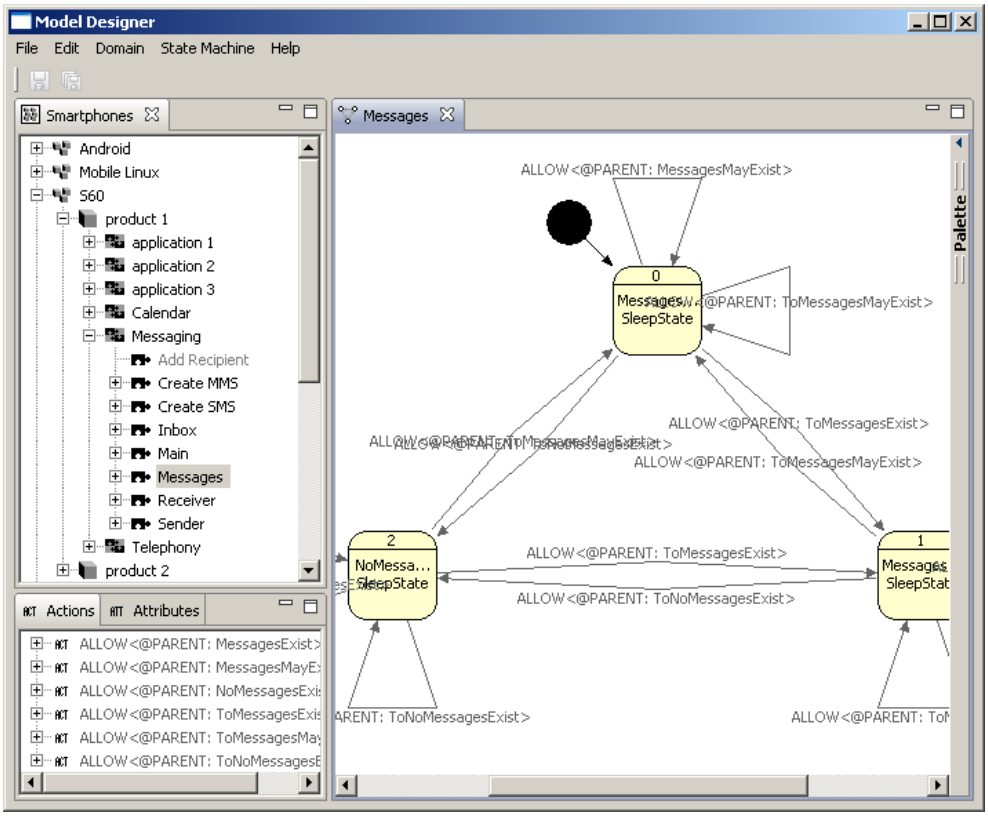

Fig. 28. *A model library in Model Designer, with the Smartphones domain structure in the view on top left. Some application and product names have been redacted for confidentiality reasons.*

More recently we have been working on a new version of the model library, with the intention of applying what we learned in the making of the first one. The new library does not yet span as many applications as the original one, but it does include several products from different product families. Figure 28 shows the library open in Model Designer.

4. MODELS IN TESTING

In a model-based testing process, models pass through a number of phases, from their creation and deployment through test generation and execution to debugging. In this chapter we will examine these phases in more detail.

4.1 Creation

Model-based testing begins with the creation of the models, which has been discussed in the previous chapters. The models may be based on a variety of software design artifacts, from requirements or specifications documentation to a working SUT. In general, the use of higher level artifacts is preferable, both because it enables the static testing of these artifacts and because it allows the modeling to begin earlier, at least in a V-model type of development process. Different methods have been compared in [30].

Creating models based on requirements documentation brings model-based testing into the development process at the earliest possible stage. This is very useful because it enables testing the design specifications against the model and locating errors before implementation. It can also act as static testing of the requirements themselves, highlighting problems in them. Technically, it might be possible to use requirements-based models as the specification for implementation. In practice this may be difficult, because it presumes model-literate programmers. Also, while requirements can be used to create action machines, they do not contain the UI information needed for refinement machines. If automatic test execution is desired, refinement machines must be created separately based on some other design artifacts.

The second option is to use the design specifications as the basis for models. This is likely to be easier than requirement-based modeling, because specifications tend to be far more detailed. Of course, in practice no specification is complete, so there

will be room for interpretation in the modeling process. However, modeling can act as a way to validate the specifications: the level of detail required in the models is high, and seeking to fill in those details can bring to light ambiguous or contradictory specifications.

The third approach is to create the models by reverse-engineering a (perhaps partially) working SUT. This technique greatly resembles exploratory testing [21]. It is especially suited for forms of testing where the effects of actions are readily apparent, such as GUI testing. Furthermore, it suffers from the same problems of ambiguity and interpretation as modeling based on design specifications, since there is no way to know for sure what the SUT is supposed to do. That is not to say that it is ineffective, though. In practice, erroneous functionality would have to be both logical and consistent in order to pass unnoticed: illogical functionality should be recognized as erroneous by the modeler, and inconsistent functionality can be discovered in test execution. Furthermore, models reverse-engineered from applications which work fine separately can still reveal concurrency issues.

One more option is to create test models by synthesizing them from test cases [P3]. This testing approach is not totally model-based, since at least some test cases have to be created manually. However, it can serve to expand the functionality covered by conventional testing, and may act as a way to introduce model-based testing into the product development process.

While modeling can be started early on in the design process, it might seem that the actual test generation would have to wait until the SUT is close to complete. After all, tests generated from a complete model are not necessarily executable on an incomplete SUT. However, this is not an insurmountable problem, since appropriate tools can be used to limit test generation to the features ready for testing [P4].

Model-based testing is compatible with incremental and iterative software design processes. This can be accomplished simply through iterative modeling: adding new features to the models as they are added to the software. Of course, requirements and solutions may be incorporated into the models in advance inasmuch as they are known.

4.2 Deployment

Once the model components have been created, they have to be prepared for execution. The simplest way to do this would be to create the models directly in the testing environment and compose them all. Such an approach has definite disadvantages, though. First, it would result in needlessly large and complicated test models for simple tests. Second, the assembled model components would be limited to one specific configuration of test targets (such as a single phone).

In theory, these problems can be avoided by manually tweaking the parallel composition. Excluding unneeded model components can reduce the size of the model, and multiple targets could be added to the test by including the model components several times and by suitably modifying the task switcher. But as the number of components and potential configurations increases the manual approach becomes untenable. Our solution is twofold: an annotated collection of model components, and tool support for its automated handling.

All the model components are assembled into a model library. In addition to the components themselves, the library also contains associated organizational information, such as which components comprise an application on which product, and which components are dependent on each other. Collecting and organizing this information is the philosophy behind Model Designer.

From a complete model library it is possible to assemble and configure the components needed for a specific test. This task is performed by Test Configurer and guided by Web GUI or a script it has generated. Web GUI is used to define a number of test targets, assign model components for each of them, and finally associate a device for each target. Based on this information, Test Configurer assembles the necessary model components for each target, and generates the structures needed to combine them into a single test model, such as the task switchers and the target switcher. At simplest, the result may be a test model for a single application on a single device. On the other hand, it is possible to assemble a model for half a dozen devices, some of them of different types, with a unique combination of applications on each. Finally, Model Composer sets up the parallel composition by creating the rule set according to the generation rules; the actual composition will be performed *on the fly* during the test run, since it may easily result in a test model too large to compute all at once [48], as seen in the example of Section 3.6. On-the-fly composition means that

the composed model is only computed around the current state to the degree necessary for test generation. Thus, this methodology allows us to create a single artifact, the model library, which can nonetheless be efficiently used for a large variety of different testing purposes.

4.3 Test Generation

The composed test model is ready for execution by Test Engine. However, before the actual test generation begins, it is necessary to define what kind of a test is to be generated. Only then can we begin executing actions in the model.

4.3.1 Testing Modes

The first thing to do in generating a model-based test is to decide what the test should do. The different goals for the tests can be divided into three categories: use cases, coverage and bug hunting. The testing modes were first presented in somewhat different form in [23].

Use case tests are generated to test specific functionality of the SUT. Their goal is expressed as a combination of the actions of the models, usually action words. At simplest this means a linear sequence of actions. More complicated cases may contain alternative paths or combine a number of use cases, perhaps even an entire test suite, into a single test run. It is important to note that there is no need to specify every single step of the test run, but only those which are the actual purpose of the test. The test will be automatically generated to include whatever other actions are necessary to reach the specified ones (assuming the specified ones are at all reachable in the model, of course). For example, if the definition of a use case test includes entering the inbox in the model shown in Figure 24, the test execution will also automatically include the launching of Messaging, because its execution is the only way to reach the desired functionality in the model. On the other hand, a sparsely defined test is more difficult to generate, since the sought-after actions are farther apart in the model. Thus, there is a trade-off between the ease of definition and generation of tests.

Coverage tests seek to traverse specific types of model structures, such as actions or transitions. They are a good way to ensure that the basic functionality of the SUT

works as intended, and have the added benefit of producing clear metrics. 100% action word coverage can be considered the very minimum of model-based testing, since it will show that all of the functionality of the models works at least some of the time. Using state and transition coverages can produce very thorough tests, but may be unfeasible in large test models.

Bug hunt is a general name for testing with no or few predefined directions. The idea is to exercise the SUT in a way likely to uncover bugs. Bug hunt can act as additional robustness testing after the use cases have been passed and the sought-after coverages fulfilled.

4.3.2 Coverage Requirements

Use case tests are defined with coverage requirements [23]. A coverage requirement is an expression which combines model actions with the operators **THEN**, **AND** and **OR**, and parentheses. *A* **THEN** *B* means that *B* must be executed at some point after *A* has been executed. Thus **THEN** defines action sequences, and is the operator most commonly used in individual use cases. *A* **AND** *B* requires that both *A* and *B* are executed, in either order or even interleaved, if they are expressions. **AND** is less common in use cases, but can be used to combine several of them into a test suite. *A* **OR** *B* states that at least one of *A* and *B* must be executed. In practice, **OR** is not used very often.

Notably there is no operator **NOT** or another way to forbid actions. Likewise, *A* **THEN** *B* only requires that *B* gets executed after *A*, but does not forbid the execution of *B* before *A*. This is because coverage requirements are meant to define only what must be done, whereas the model defines what may be done. As a consequence, coverage requirements may be freely combined with any operators with no danger of contradiction.

Individual actions are written into coverage requirements in the form **action** *actionname* or **actions** *actionname*, where *actionname* is a Python regular expression [40]. The two forms require the execution of one action matching the expression and all such actions, respectively. Use cases are normally defined using the first form with exact action names as regular expressions. The latter allows coverage requirements to express some coverage mode goals, such as executing all action words in the model.

In the future, coverage requirements may be extended to also refer to model structures other than actions.

As an example of a coverage requirement, a use case for entering the inbox and then leaving Messaging in the model of Figure 24 could be expressed as **action** *.*end_awToInbox* **THEN action** *.*end_awExitMessaging*. The action names refer to the end parts of the action words to ensure that their implementations get executed before the coverage requirement is fulfilled. In this case the names of the model components can be safely abstracted into ".*", since the action names remain unique even without them.

4.3.3 Guidance Algorithms

With the goal of the test defined, it is the task of a *guidance algorithm* to fulfill that goal. There are many different kinds of algorithms that can be used, with different strengths and weaknesses. Most, though not all, seek out specific targets in the test model such as actions, states or more complex model structures. Different algorithms are compared in [34].

The simplest algorithm is the completely random one. While the actual guidance is minimal, a random algorithm does have the advantage of being very fast. This property can make it useful in coverage mode tests that look for common targets such as states. Random guidance can raise the coverage quickly early on, especially on a fast SUT capable of executing dozens of actions per second or more. It is not very effective in taking the coverage very high, though, and should be swapped for another algorithm at some point. Finally, random guidance is a good choice for bug hunt, where there are no specific targets to look for anyway.

Tabu guidance is a slightly more complicated algorithm. Like random, it picks one of immediately available transitions without calculating ahead. However, it seeks to avoid targets which it has visited recently. Thus, tabu guidance is more likely to explore new areas in the model. It is suited to much the same tasks as random guidance.

Use case mode tests require a true graph search algorithm which can seek the desired targets from deeper within the model. The simplest version of these is the breadth-first search, which will seek out the closest desired target. Apart from use case tests,

it also works fairly well in coverage mode.

Another graph search algorithm is the fixed depth search. Such an algorithm searches the model in all directions until it reaches a specific distance from its starting point, and then chooses the best of all the paths thus covered. Its advantage over breadth-first search is that it can recognize target-rich areas of the model, whereas breadth-first search stops looking when it finds the first one. Fixed depth search can be a very good choice for use case and coverage mode tests. However, large search depths can make it very slow.

Bug hunt works well with random algorithms. It can also make use of graph search algorithms with suitably defined targets, such as switches between pairs of applications. More specialized algorithms are possible, with properties such as keeping a large number of applications in execution simultaneously.

4.4 Keyword Execution

If the test is generated off-line, it is merely necessary to create a list of executed actions. This list becomes a script which can be executed on the SUT later on. In online testing, the execution happens concurrently with generation. In both cases, only keywords are executed on the SUT.

Keywords are first sent to Adapter, which translates them into a form understood by Connectivity Component. Often the translation can be fairly simple, such as with keywords designating key presses. On the other hand, Adapter may provide complex features far beyond those offered by Connectivity Component, such as a keyword which automatically selects the desired item from a multilayered menu. While such composite actions could be implemented within the model, they would have to be implemented separately in every case where they are needed, whereas Adapter only needs to implement them once. Thus, it is usually most efficient to implement in the Adapter any complicated action sequences that are needed even occasionally.

Once the keyword has been executed in the SUT, Connectivity Component tells whether the execution succeeded or not by returning a Boolean value, which Adapter conveys to Test Engine. With some keywords such as key presses the return value is mostly a formality, but with others such as text verifications it is the very purpose of the keyword.

In off-line testing, it is only necessary to check that the returned value is what the script defines it should be; if not, the test has failed. In online testing there is a further possibility of a branching keyword, in which case the return value determines the continued direction of the test.

One thing to take into account is that keywords' effects may not fully play out by the time the success or failure of execution is clear and returned. For example, a launched application may require several seconds before it is ready for use. In such a case it may be necessary to wait a while before executing the next keyword. In practice it is most convenient to add a suitable delay for each keyword in the Adapter, depending on how much time is usually required for all the effects to play out. If exceptional delays are needed, such as for a key press moving an application to a view which takes several seconds to open, the delay is best specified in the model.

4.5 Debugging

When testing pays off and an error is found, it is necessary to find out what caused it. Model-based testing can cause some extra difficulties in this process. First, especially online model-based tests can be very long, possibly hours or days in duration. Finding the cause of the error in such a long trace of execution can be difficult. Second, since the test model can be quite complex, it may not be easy to tell whether the error is in the model or in the SUT.

The main debug information comes from the test log, which lists all the actions executed during the test run. However, debugging a test from the log alone is not always easy. Especially helpful in GUI testing is the ability to see what is happening on the SUT. This can be done in several ways, such as using a suitable application to record the GUI or taking periodic screenshots. If nothing else, it is possible to use a separate camera to record the SUT. Such recordings can be very valuable in debugging. For example, if the error is caused by a battery notification appearing in the middle of a keyword execution, the cause can be nigh on impossible to determine from the log yet immediately apparent from the recording.

Another approach to handling a long error trace is to try to create a shorter trace which nonetheless reproduces the error. There are various methods for finding such traces, such as removing loops from the original error trace. One option is to simply

execute parts of the error trace until one that reproduces the error is found. These debugging methods, as well as video recording, are explored in detail in [P5].

5. CONCLUSIONS

In this thesis we have presented the approach of using a model library for online GUI testing. The model library is a collection of model components, each of which depicts some part or aspect of the SUT. Components from the library can be composed into a test model, which can then be used to generate tests. The ability to choose the components according to the needs of testing makes the approach very flexible. Test models can include exactly the components for the applications of interest, and may be created to span one or more devices, possibly of different types, each with its own assembly of model components.

The TEMA toolset we have developed offers support in every phase of the model-based testing process, from the creation and maintenance of the models through test generation and execution to debugging. From the point of view of the model library the most important tools are Model Designer, which is used in its creation, and Web GUI, which controls the assembly of test models and test generation.

Our experiences show that a functional model library can be created with reasonable effort, and used effectively. Modeling does require some expertise, though, and may be best left to a designated test modeler. In contrast, generating and executing tests with TEMA toolset does not require insight into the structure of the models or the generation algorithms.

Our methodology has proven quite flexible as far as the domain of application is concerned. At first our focus was solely on the S60 smartphones on the Symbian platform [P6], but the methods have since been successfully used also on various Mobile Linux platforms such as Android [12] in several case studies [P7] [19, 43]. There is no reason why the methodology would not work well outside the smartphone domain as well; in [38] it is used to test a Java Swing application.

The case studies we have conducted have shown not only the adaptability of our methodology, but also its effectiveness. In many cases we have been able to find bugs

in products which have already been through testing and released to mass market. Notably the majority of the bugs have been found during the modeling phase; a clear advantage, since modeling can be done much earlier in the product lifecycle than automated test execution.

This far our case studies and experiments have employed almost purely qualitative metrics. Although proper quantitative analysis of our methods would be extremely valuable, it is also very difficult to arrange in practice. Still, we might seek at least some kind of quantitative metrics in the future.

Another task for the future is the further development of the tools. Although quite serviceable as they are, there are several aspects in which they could be improved. For example the support for combining model components created by different modelers into a single library is currently seriously lacking in Model Designer. Likewise, Web GUI is under constant development in order to improve the ease and flexibility of test generation.

Although there is still much room for development, our approach has clearly shown its effectiveness in terms of flexibility, maintainability, and the ability to find bugs. Hopefully it can in its part further the practical adoption of model-based testing, and thereby improve the quality of software.

BIBLIOGRAPHY

[1] F. Belli, “Finite-state testing of graphical user interfaces,” in *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2001, pp. 34–43.

[2] F. Belli, C. J. Budnik, and L. White, “Event-based modelling, analysis and testing of user interactions: Approach and case study,” *Software Testing, Verification and Reliability (STVR)*, vol. 16, no. 1, pp. 3–32, Mar. 2006.

[3] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. New York, NY, USA: ACM Press/Addison–Wesley, May 2000.

[4] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting, “Requirements traceability in automated test generation – application to smart card software validation,” in *Proceedings of the 1st International Workshop on Advances in Model-Based Software Testing (A-MOST 2005)*. New York, NY, USA: ACM Press, May 2005, pp. 1–7.

[5] F. Bouquet and B. Legeard, “Reification of executable test scripts in formal specification-based test generation: The Java Card transaction mechanism case study,” in *Proceedings of the International Symposium of Formal Methods Europe (FME 2003)*, ser. Lecture Notes in Computer Science, K. Araki, S. Gnesi, and D. Mandrioli, Eds., vol. 2805. Berlin, Heidelberg: Springer, Sep. 2003, pp. 778–795.

[6] H. Buwalda, “Action figures,” Software Testing and Quality Engineering Magazine, pp. 42–47, Mar./Apr. 2003.

[7] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, “Testing concurrent object-oriented systems with Spec Explorer,” in

Proceedings of the International Symposium of Formal Methods Europe (FME 2005), ser. Lecture Notes in Computer Science, J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds., vol. 3582. Berlin, Heidelberg: Springer, Jul. 2005, pp. 542–547.

[8] Conformiq Inc., http://www.conformiq.com/, 2010, cited Dec. 2010.

[9] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Artech House, Jan. 2002.

[10] M. Fewster and D. Graham, *Software Test Automation: Effective Use of Test Execution Tools*. Addison–Wesley, Sep. 1999.

[11] M. Fowler, "Continuous integration," http://www.martinfowler.com/articles/continuousIntegration.html, 2006, cited Dec. 2010.

[12] Google Inc., "Android.com," http://www.android.com/, 2010, cited Dec. 2010.

[13] H. Hansen, H. Virtanen, and A. Valmari, "Merging state-based and action-based verification," in *Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD 2003)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2003, pp. 150–156.

[14] A. Hartman, M. Katara, and S. Olvovsky, "Choosing a test modeling language: A survey," in *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing (HVC 2006)*, ser. Lecture Notes in Computer Science, E. Bin, A. Ziv, and S. Ur, Eds., vol. 4383. Berlin, Heidelberg: Springer, Mar. 2007, pp. 204–218.

[15] J. Helovuo and S. Leppänen, "Exploration testing," in *Proceedings of the 2nd International Conference on Application of Concurrency to System Design (ACSD 2001)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2001, pp. 201–210.

[16] J. Helovuo and A. Valmari, "Checking for CFFD-preorder with tester processes," in *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, ser. Lecture Notes in Computer Science, S. Graf and M. Schwartzbach, Eds., vol. 1785. Berlin, Heidelberg: Springer, Mar. 2000, pp. 283–298.

[17] A. Huima, "Implementing Conformiq Qtronic," in *Proceedings of the Joint Conference of the 19th IFIP International Conference on Testing of Communicating Systems and the 7th International Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES 2007)*, ser. Lecture Notes in Computer Science, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds., vol. 4581. Berlin, Heidelberg: Springer, Jun. 2007, pp. 1–12.

[18] A. Jääskeläinen, "A domain-specific tool for creation and management of test models," Master's thesis, Tampere University of Technology, Tampere, Finland, Jan. 2008.

[19] A. Jääskeläinen, T. Takala, and M. Katara, "Model-based GUI testing of Android applications," in *Software Test Automation Experiences*, D. Graham and M. Fewster, Eds. Addison-Wesley (Pearson Education), to appear.

[20] J. Jacky, M. Veanes, C. Campbell, and W. Schulte, *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

[21] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley, Jan. 2002.

[22] K. Karsisto, "A new parallel composition operator for verification tools," Doctoral dissertation, Tampere University of Technology, Tampere, Finland, 2003, number 420 in publications.

[23] M. Katara and A. Kervinen, "Making model-based testing more agile: A use case driven approach," in *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing (HVC 2006)*, ser. Lecture Notes in Computer Science, E. Bin, A. Ziv, and S. Ur, Eds., vol. 4383. Berlin, Heidelberg: Springer, Mar. 2007, pp. 219–234.

[24] A. Kervinen, "Towards practical model-based testing: Improvements in modelling and test generation," Doctoral dissertation, Tampere University of Technology, Tampere, Finland, Nov. 2008, number 769 in publications.

[25] A. Kervinen and P. Virolainen, "Heuristics for faster error detection with automated black box testing," in *Proceedings of the 1st Workshop on Model Based Testing (MBT 2004)*, ser. Electronic Notes in Theoretical Computer Science, vol. 111. Elsevier, Mar. 2005, pp. 53–71.

[26] A. van Lamsweerde, "Formal specification: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering (ICSE 2000)*. New York, NY, USA: ACM, Jun. 2000, pp. 147–159.

[27] B. Legeard and F. Peureux, "Generation of functional test sequences from B formal specifications – presentation and industrial case study," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 01)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2001, pp. 377–381.

[28] B. Legeard, F. Peureux, and M. Utting, "Automated boundary testing from Z and B," in *Proceedings of the International Symposium of Formal Methods Europe (FME 2002)*, ser. Lecture Notes in Computer Science, L.-H. Eriksson and P. A. Lindsay, Eds., vol. 2391. Berlin, Heidelberg: Springer, Jul. 2002, pp. 21–40.

[29] Linux Foundation, "MeeGo," http://meego.com/, 2010, cited Dec. 2010.

[30] Q. A. Malik, A. Jääskeläinen, H. Virtanen, M. Katara, F. Abbors, D. Truscan, and J. Lilius, "Model-based testing using system vs. test models – what is the difference?" in *Proceedings of the 17th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2010) (poster session)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2010, pp. 291–299.

[31] A. M. Memon, "A comprehensive framework for testing graphical user interfaces," Ph.D. dissertation, University of Pittsburgh, Pittsburgh, PA, USA, 2001.

[32] Microsoft, "NModel," http://nmodel.codeplex.com/, 2010, cited Dec. 2010.

[33] J. Mikkola, "Mallipohjainen testaus Symbian-ympäristössä [Model-based testing in the Symbian environment]," Master's thesis, Tampere University of Technology, Tampere, Finland, Aug. 2009, in Finnish.

[34] A. Nieminen, A. Jääskeläinen, H. Virtanen, and M. Katara, "A comparison of test generation algorithms for testing application interactions," under submission.

[35] Nokia, "Forum Nokia – Symbian platform," http://www.forum.nokia.com/Devices/Symbian/, 2010, cited Dec. 2010.

[36] Object Management Group, Inc., "UML," http://www.omg.org/spec/UML/, 2010, cited Dec. 2010.

[37] T. Ostrand, A. Anodide, H. Foster, and T. Goradia, "A visual test development environment for GUI systems," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1998)*. New York, NY, USA: ACM, Mar. 1998, pp. 82–92.

[38] T. Pajunen, "Model-based testing with a keyword-driven test automation framework," Master's thesis, Tampere University of Technology, Tampere, Finland, Jun. 2010.

[39] Python Software Foundation, "Python programming language," http://python.org/, 2010, cited Dec. 2010.

[40] Python Software Foundation, "Python v2.7.1 documentation – Regular expression operations," http://docs.python.org/library/re.html, 2010, cited Dec. 2010.

[41] H. Robinson, "Finite state model-based testing on a shoestring," Software Testing, Analysis, and Review Conference (STARWEST), 1999, available at http://www.geocities.com/model_based_testing/shoestring.htm, cited Dec. 2010.

[42] A. W. Roscoe, *The Theory and Practice of Concurrency*, C. A. R. Hoare and R. Bird, Eds. Upper Saddle River, NJ, USA: Prentice Hall, Nov. 1997.

[43] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an Android application," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2011, to appear.

[44] Tampere University of Technology, "TEMA project," http://practise.cs.tut.fi/project.php?project=tema, 2010, cited Dec. 2010.

[45] J. Tretmans, "A formal approach to conformance testing," Ph.D. dissertation, Twente University, Enschede, the Netherlands, 1992.

[46] M. Utting and B. Legeard, *Practical Model-Based Testing – A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[47] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing,” Department of Computer Science, University of Waikato, Hamilton, New Zealand, Working Paper 4, Apr. 2006, available at http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf, cited Dec. 2010.

[48] A. Valmari, “The state explosion problem,” in *Lectures on Petri Nets I: Basic Models*, ser. Lecture Notes in Computer Science, W. Reisig and G. Rozenberg, Eds., vol. 1491. Berlin, Heidelberg: Springer, 1998, pp. 429–528.

[P1] With kind permission from Springer Science+Business Media: Proceedings of the Joint Conference of the 20th IFIP International Conference on Testing of Communicating Systems and the 8th International Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES 2008), "Model-Based Testing Service on the Web", ser. Lecture Notes in Computer Science vol. 5047, 2008, pp. 38–53, A. Jääskeläinen, M. Katara, A. Kervinen, H. Heiskanen, M. Maunumaa, and T. Pääkkönen,

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 [7, 6] 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 attemps 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.*

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.

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 problem. 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.

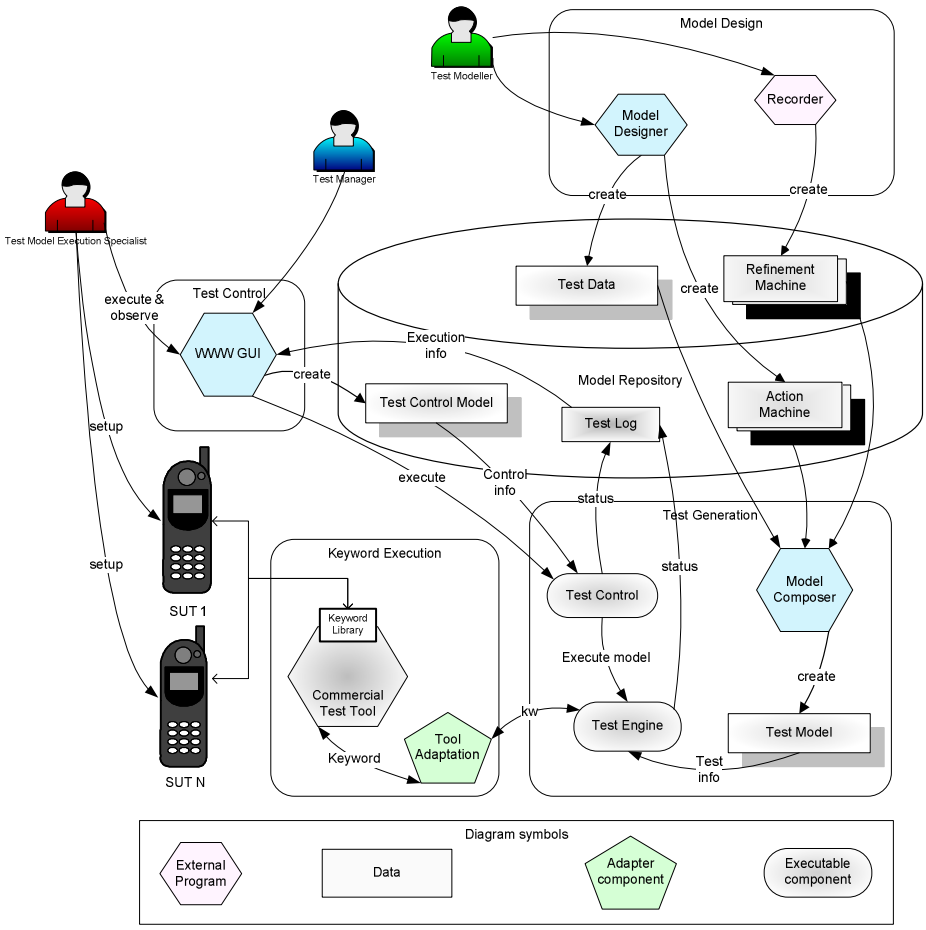

Fig. 1. Test tool architecture.

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 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. The latter 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.

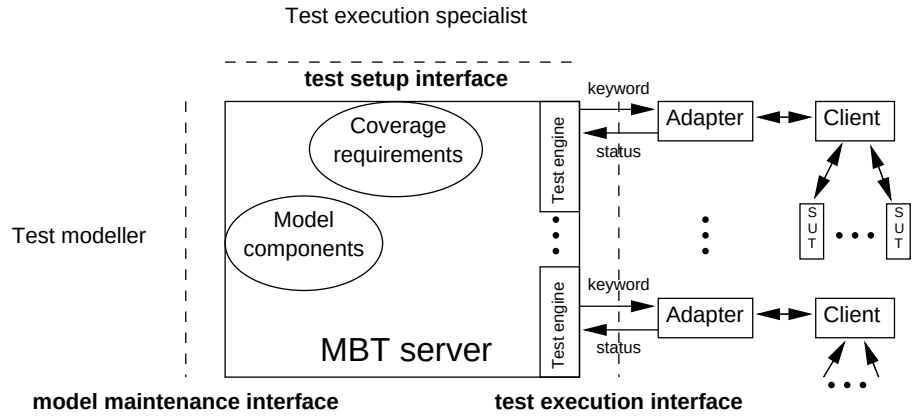

Fig. 2. MBT testing server, adapters, clients, and SUTs.

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
```

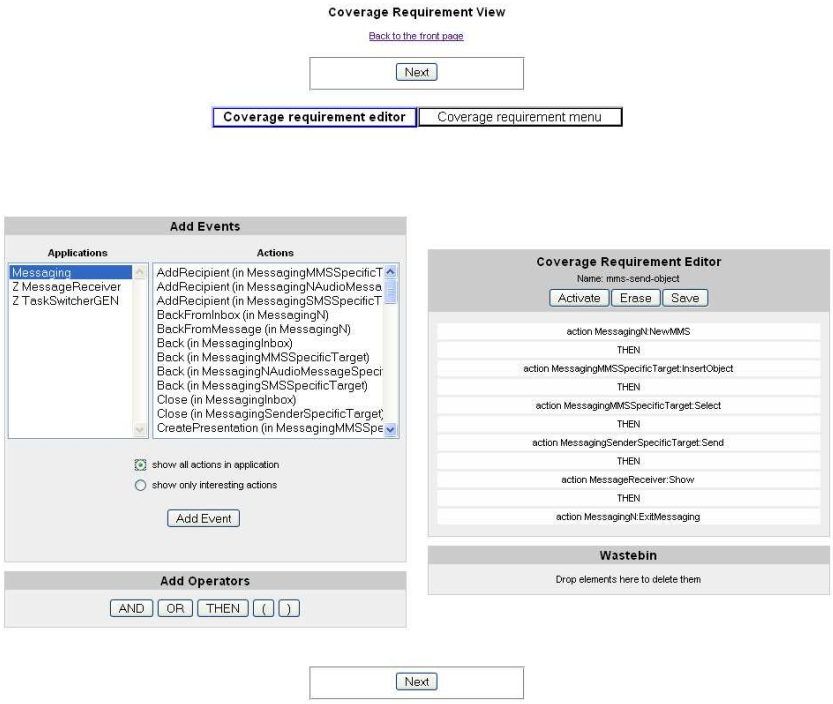

Fig. 3. Coverage requirement editor.

```
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.

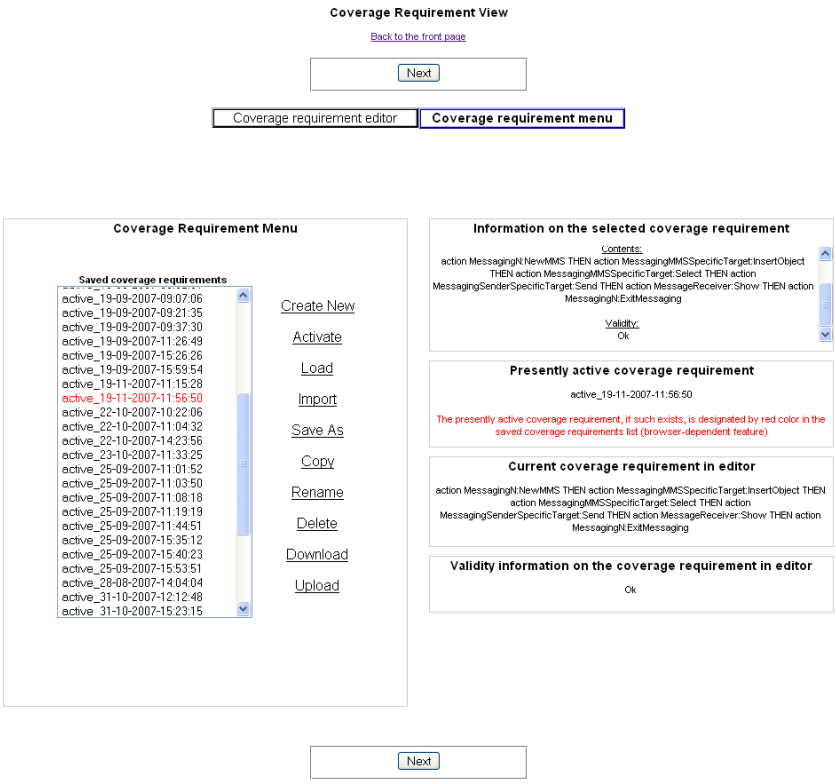

Fig. 4. Coverage requirement menu.

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

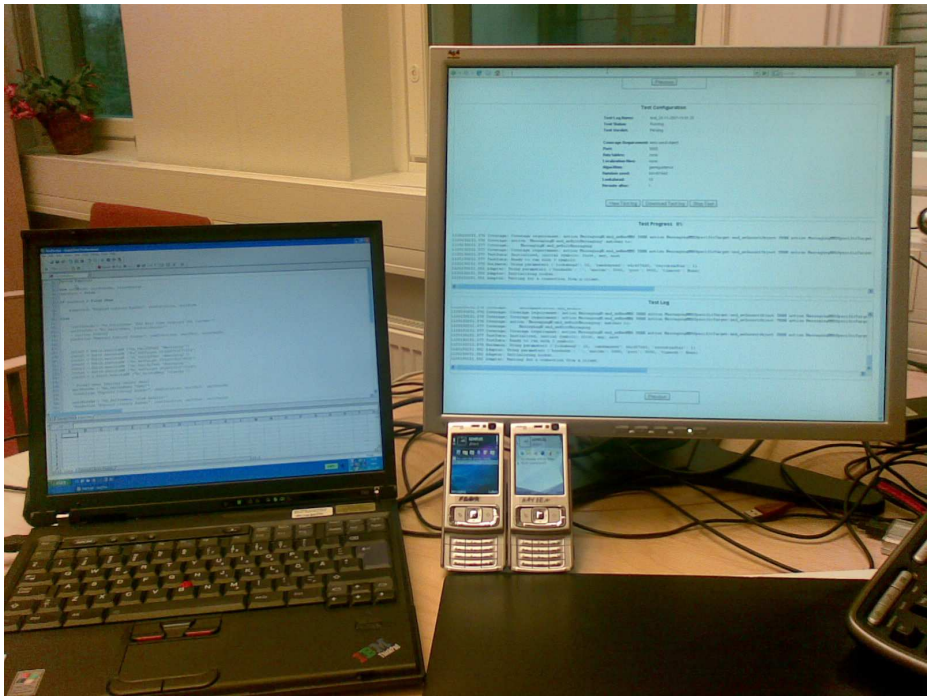

Fig. 5. Test setup with two SUTs.

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 Available at `http://www.model-based-testing.org/ObstaclesAndOpportunities.pdf`. Cited May 2007.
4. Hartman, A.: AGEDIS project final report. Available at `http://www.agedis.de/documents/FinalPublicReport%28D1.6%29.PDF` (2004) Cited May 2007.

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 May 2007)
7. S60. (http://www.s60.com. Cited May 2007)
8. OMG: UML testing profile, v 1.0. (http://www.omg.org/technology/documents/formal/test_profile.htm. Cited Dec 2007.)
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 Jul. 2007)
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 Dec 2006.
22. HP: Mercury Functional Testing homepage. (http://www.mercury.com/us/products/quality-center/functional-testing/. Cited Jul 2007.)
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.

Creating a Test Model Library for GUI Testing of Smartphone Applications

Antti Jääskeläinen, Antti Kervinen, and Mika Katara
Tampere University of Technology, Department of Software Systems
P.O.Box 553, FI-33101 Tampere, FINLAND
{antti.m.jaaskelainen,antti.kervinen,mika.katara}@tut.fi

Abstract

Smartphones are becoming increasingly complex, and the interactions between the different applications make testing even more difficult given the time-to-market pressures and the limits of current test automation systems. Towards these ends, we have built an open source test model library for Symbian S60 GUI testing. This paper describes and analyzes our experiences in building the library.

1 Introduction

Unlike many other areas of engineering, most software defies formal analysis due to its great complexity. The verification and validation of software is mostly performed by extensive testing, which can never yield 100% certainty. When developing products for mass consumer markets, testing is often done through a graphical user interface (GUI). While GUI testing is sub-optimal in many respects compared to other kinds of tests, it is still considered important when checking the functionality and performance from the viewpoint of the end-user experience.

An attempt to decrease the costs of GUI testing has been made by partially automating the process. The results have been varied, from early capture and replay methods yielding few practical benefits to data-driven scripts which have proven reasonably efficient in some forms of testing [4]. However, they have usually proved inefficient in finding new defects in the software. Some of these problems may be solved with model-based testing (MBT) [12].

In MBT the essential functionality of the system under test (SUT) is formally modeled, and then tests are generated from the model. Unlike script-based testing, where test cases are run automatically but must be designed manually, MBT can be used to automatically generate the test cases as well. However, it requires a viewpoint very much different from traditional test automation.

We are developing a model-based GUI testing solution and trying to find ways in which it could be adopted as a testing method in practice. One of the core ideas is *domain-specificity*, as it should ease the adoption process. In our case, the domain is GUI testing of Symbian smartphone applications. There have been cumulatively over 150 million Symbian smartphones shipped [10]. We concentrate on devices with the S60 GUI platform, the most commonly found framework among the current phone models. In addition to device manufacturers, a large number of third party software developers make applications on top of the platform.

We have also created a test model library that enables the testing of new smartphone applications in conjunction with others, including the predefined ones. In [9] we reported our initial experiences in testing this domain; most of the issues found in the SUTs were concurrency related. With the help of such a model library, the testing of new applications can be facilitated: the library can be easily extended with new model components to be included in the final test models. Such test model libraries could allow device and platform manufacturers to set common quality requirements for third-party application in a way similar to the Symbian Signed program [11]. Another example is the TTCN-3 SIP Test Suite [3] available for conformance testing.

The background of our approach has been introduced previously in [6, 7, 8, 9]. In this paper, based on [5], we concentrate on the test model library. In Section 2 we review the background of our contributions. In Section 3 we discuss our experiences in creating the test models and the contents of the test model library. Section 4 contains some concluding remarks.

2 Background

2.1 Action words and keywords

A solution to some of the problems of automated GUI testing is to separate the functionality of the SUT from its implementation [2]. The functionality is described in *action words*, which correspond to the actions the SUT can perform. The level of detail and complexity of action words can vary greatly, from simple ones that denote little more

than a key press to very complicated ones that correspond to whole use cases. However, no matter what they describe, action words should never be directly tied into the specifics of the interfaces, graphical or otherwise.

Where action words describe the functionality of the SUT, *keywords* are used to describe the implementation of that functionality. Each keyword corresponds to some basic event in the interface of the SUT, such as pressing a key or verifying that the text output matches expectations. Just as action words should be independent of interfaces, keywords should be independent of any functionality they may be used to implement. While generally simple, keywords may also be complex in nature; for example, a keyword to type text into a phone may perform the equivalent of a very complicated series of key presses. Because of this, there is no hard line between action words and keywords in terms of what they can express, as a complex keyword may be more complicated than a simple action word. The true difference is in how they are used.

In the context of testing mobile applications, a typical action word could correspond to a user opening an application for sending SMS messages when the phone is in the idle state. In this case the action could be refined to a sequence of keywords corresponding to the key strokes needed to open the application in the real phone. There may be multiple ways for opening such an application (short cut, menu, etc.), and each of these could be encoded as a separate sequence implementing the same action word.

The advantage of using action words and keywords is that test cases can be created as sequences of action words. In this way, they are no longer tied directly to volatile interfaces, but rely on the relatively more stable functionality of the SUT. When the interface changes, only the implementations of the action words must be modified to match. Only a change in the functionality of the system may require modifications in the test cases, but such changes are much less common. Therefore, by using action words and keywords, one very significant problem of automated testing, namely maintenance, can be avoided.

2.2 Modeling formalism

The test models are in Labeled State Transition System (LSTS) format as defined in [6]. They 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; as such, they are the main focus of the modeling work. Keyword implementations for action words are defined in refinement machines. Together action machines and refinement machines form most of the model architecture. The remaining two types are more specialized. 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. Initialization machines are executed before the actual test run, and 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; as a consequence, explicitly defined launch and initialization machines are rarely needed.

In order to be used in a test run, the model components must be combined into a test model in *parallel composition*. The components involved are action machines, refinement machines and launch machines (explicitly defined and automatically generated), as well as two automatically created models called the *task switcher* and the *synchronizer*. The task switcher manages the synchronizations between the model components of a single testing target, whereas the synchronizer helps in the handling of synchronizations between multiple targets. In the composition, the model components are examined and rules generated for them according to the semantics in order to determine what actions can be executed in a given state. 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. The latter method has been found preferable, because combining a large amount of models can easily result in a state explosion problem [13] serious enough to overpower any computer. The definition of parallel composition for LSTSs can be found in [6].

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

2.3 Action types

The actions to be used in the creation of the models can be divided into five types. These are action words, keywords, action machine synchronizations, action word synchronizations and comments (which have no semantic meaning). The action machine synchronizations can be further divided into task switcher synchronizations, activation synchronizations and request synchronizations.

Action words are used to describe the functionality of the modeled systems and therefore form the core of the test models. Generally action words must be executed correctly in a well-functioning SUT, since they describe its valid use. However, it is also possible to use so-called failing action words to cope with nondeterministic SUTs by directing the

model into a different state in case of failure in execution. The success or failure of an action word's execution is determined by its keyword implementation.

The other cornerstone of the models are the *keywords*, which describe events in the SUT and are used to refine the action words. Just as action words, keywords may be required to succeed or allowed to fail as need be. In addition, they may be required to fail, such as in verifying that a given text has disappeared from the screen. Their success of execution depends directly on SUT events.

The first category of action machine synchronizations, *task switcher synchronizations*, contains exactly two actions: $Wake_{TS}$ and $Sleep_{TS}$. $Wake_{TS}$ may only go from a sleeping state to running state and allows the task switcher to grant control to this model. $Sleep_{TS}$, conversely, goes from a running state to sleeping state and gives up control to the task switcher.

Activation synchronizations also change control between the component models. Very much like task switcher synchronizations in most respects, activation synchronizations, however, bypass the task switcher and switch control directly between the two models.

Request synchronizations come in three distinct types: *Allow*, *Req* and *ReqAll*. They handle communication between test models. *Req* must be executed synchronously with a corresponding *Allow*, and *ReqAll* with all corresponding *Allows*. If the required *Allow(s)* cannot be executed, then neither can the *Req*/*ReqAll*. Depending on their placement, request synchronizations can be used to ask for permission for an operation, for instance.

Action word synchronizations are used to synchronize keywords to action words. They are rarely placed manually, however, but are created by the modeling tools as needed. The only action word synchronization commonly placed manually is *return*, which is used in relation to failing action words.

2.4 Machine semantics

As explained earlier, the four types of machines created by the test modeler are action machines, refinement machines, launch machines and initialization machines. Apart from the action semantics, the only common semantic requirement for all four types is that they must be *deterministic*, meaning that no state may have more than one outgoing transition labeled by the same action.

An action machine must be *strongly connected*, i.e., each of its states must be reachable from every other state. In effect, this means that the machine may not contain *deadlock states* (states with no outgoing transitions) unless it consists of only a single state. Action machines may have action words and action machine synchronizations as actions. They may also have *state verifications*, which are state labels with a special semantic meaning. A state verification will be refined by the appropriate refinement machine just as action words; the resulting action sequence becomes a loop in the state where the state verification was placed. This feature can be useful in verifying that the model state corresponds to the state of the SUT.

Refinement machines consist of implementations for action words. The implementing keyword sequences are usually constructed as loops in a single central state, beginning and ending with action word synchronizations. Such a structure ensures that all action words can always be refined, since a refinement machine should not restrict the functionality of an action machine. The sequences may contain branches and even loops, though the latter are strongly discouraged. Like action machines, keyword machines must be strongly connected.

Though they are used in different situations, launch machines and initialization machines have exactly the same semantics. Unlike action and refinement machines, they are not strongly connected; instead, a deadlock state must be reachable from each of the machine's possible states. When the execution reaches a deadlock state, its task is considered to be finished. These types of models may only contain keywords as actions. Their have essentially the same structure as the keyword sequences in refinement machines.

2.5 Data integration

Typically, many keywords require one or more parameters to define their function. Sometimes these parameters represent real-world data; a date or a phone number, for example. Embedding such information directly into the models is not advisable, because they would be limited to a fixed set of data values and possibly tied to specific test configuration. Furthermore, storing data into state machines can drastically increase the number of states in the composed model. To solve these problems, we have developed two methods of varying the data in models: localization data and data statements with data tables.

Localization data functions as a simple text replacement, where an identifier in an action is replaced by a text string from a localization table. As the name implies, localization tables are usually used to hold GUI text strings in different languages. However, they may also be used with other kinds of data, if text replacement is all that is needed.

More complicated use of data can be accomplished by placing *data statements* into actions. The statements are Python [1] statement lists which may use any Python functionality. They can also access *data tables*, which are lists of structured data elements created for this purpose. The data statements can be used to create a text string to replace the statement in the action and to alter the persistent state of the Python interpreter for the following data statements.

3 Building the S60 GUI test model library

3.1 Observations in modeling

As we began the development of our methodology one of the tasks was the creation of test models for applications in S60 phones. While we had earlier experience in modeling and verification, our primary modeler had done little or no modeling apart from drawing class diagrams. This may not have been an entirely bad thing, as it allowed us to begin from scratch and it led to some new innovations.

Throughout the whole modeling process we never used any specifications for the applications we modeled. Therefore, instead of specifications, the models were based on observations and, to some extent, common sense. Obviously, up to date specification would have been potentially helpful in model creation. However, we think that the lack of specifications gave us a rather realistic setting; nowadays popular agile methods do not encourage detailed specification, instead the implementation is seen as the most important artifact. Moreover, this led us to use some exploratory testing practices to develop the models, and we were also able to find some real defects [9].

When modeling an application, we would begin by starting the application in a phone, moving it through its major screens and trying out its functionality. Once we had an idea of how the application worked, we would create models to cover its basic behavior and then add in functionality until the models appeared sufficiently detailed. Without specifications we could not always be certain whether the application was functioning correctly or not. In general, if the behavior was consistent and logical, we would assume that everything was working as intended. In these cases, more than anywhere else, good specifications might have been useful.

After some practice in drawing small single-purpose models, we took on the true task of creating a relatively complete set of test models for the S60 platform applications. We began with the Calendar application, as it appeared quite simple compared to many others. We soon discovered that even in its relative simplicity the application was too complex to fit neatly into a single model. As a solution, we divided the Calendar functionality into several portions and created test models for each, individually. Apart from the base model for the application, the models could not be woken by the task switcher but only through direct synchronizations from the other Calendar models. In this way, each model became simple enough to understand.

As the model of the Calendar application was split, the resulting action machines seemed to fall into three categories, according to how they were synchronized into the whole. While this division has since prevailed in most of our modeling work, it is intuitive rather than formal. In the first category are structural models that describe the major lines of an application. These models typically contain little functionality in themselves. What they do have is synchronizations with other models; our model library even contains a few structural models that consist of nothing else. Their internal functionality is mostly concerned with moving through the different windows of their application, although they sometimes contain simple actions within those windows, usually small enough to be modeled with a single action word. The base models of applications are nearly always structural models, and a complex application may have several structural models describing its major parts.

The second type of model is the subroutine model. Such a model describes one or more actions that can be performed in a certain state of the modeled application. Their great advantage is that they can be easily connected to multiple points in structural models. Subroutine models resemble subroutines of programming languages: they are woken through an activation synchronization, perform a limited series of actions and finally give up control through another synchronization. While they may contain an initial choice of actions, each action generally proceeds in a linear fashion to its end. Significant branches and loops are quite rare. Long actions can typically be interrupted in the middle and the control given to the task switcher; apart from these cases, synchronizations are uncommon, though sometimes requests may be necessary to determine what actions can be performed. There has been one notable exception to this rule, where a subroutine model was created to take care of a particularly complicated series of synchronizations.

The final model category consists of memory models. The original model semantics had no method for data handling, and the only persistent information was the current states of the models. Therefore, when memory was needed, the solution was to create a model that would record the necessary information in its state when sent to sleep. A memory model, typically, has a few waking states, most commonly just two, and an equivalent number of sleeping states. Normally the only actions are ones that change the current state of memory. The sleep states usually have *Allow* synchronizations so that the state of the model can be observed from the outside, and sometimes others that let the state be changed as well. One drawback of the memory models is their effect on state space: a two-state memory (a single bit) effectively doubles the size of the whole model.

Figure 1 shows an example of how the models of different categories relate to each other. The example is from the part of the Gallery application that deals with images. At top left there are three structural models for the Gallery main screen, the image selection screen and the image view screen. The last of these is connected to a subroutine model below for attaching an image to a contact, and two memory models on the right for managing the zoom level and

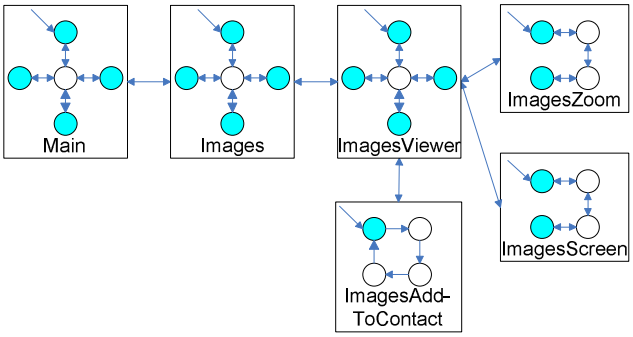

Figure 1. Example of model structure from the Gallery application.

the normal/full screen status. Note that the model pictures in the figure are not actual state machines; rather the state structures shown are iconic for models of their categories, with filled circles corresponding to sleeping states.

It was clear that memory beyond the rather limited possibilities offered by memory models would be useful. In Calendar the problem was in keeping track of the number of calendar entries. The exact number of entries could be practically stored in states only when it was very small; otherwise both the model and the state space would grow prohibitively large. However, we were unwilling to impose an artificial limit on the number of entries that could be created using the Calendar models, especially if the limit was to be a very low one. Because of these factors, we decided to ignore the exact number of entries and limit ourselves to three states: one in which entries were known not to exist, another in which they were known to exist, and the last one where entries might or might not exist. This solution was not entirely satisfactory, either; a notable deficiency was that two entries could not be deleted without creating a new one in between, as there was no way of knowing that another entry still existed after one was deleted. The solution was feasible only because of the possibility of deleting all entries at once, which made bringing the model back to initial state always possible. Eventually, these problems led to the concept of action words that are allowed to fail. The later advent of data statements has allowed us to dispose of some memory models, though others are still necessary.

3.2 Advances in semantics

Modeling the next application, Contacts, yielded one significant addition to the model semantics. The application holds the contact information and provides shortcuts for making phone calls and sending messages. While we did not yet model the sending and receiving of messages at that point, we did notice that it was possible to send a message to several receivers simultaneously. The basic method of relaying the information about the message to another model, the *Req* synchronization, was obviously not feasible with multiple recipients. The solution was the *ReqAll* synchronization, which allowed the synchronization of more than two models at once. The semantics of *ReqAll* were defined in such a way that it could be set up to act as the exact negation of *Req*, a feature which has proven very useful at times. Another good characteristic of *ReqAll* is that it works regardless of the number of corresponding *Allow* actions in the component models, none or a dozen. Because of this, *ReqAll* can be used as an announcement of sorts, with all interested models always allowing its execution, but reacting to it with non-looping *Allow* transitions.

Another realization brought on by Contacts was that some way of incorporating data into the models would be eventually required. In Calendar, it did not really matter how the entries were named; what variance was wanted could be achieved by using alternate keywords with different parameters. However, if Contacts was to be used for sending messages, it was clearly not sensible to embed names and phone numbers directly into the model. These requirements led eventually to the incorporation of localization data and, finally, data tables into the model semantics.

The modeling of Calendar and Contacts had brought out all the major requirements in model semantics. One later addition was the use of state verifications. These special state labels could be transformed into keyword sequences that would check that the SUT is in the correct state.

As the third application, Gallery, was being modeled, it finally became apparent that combining the model components resulted in a serious state space explosion problem. The early versions of Gallery, together with the more complete Calendar and Contacts, yielded a huge state machine with millions of states. Later attempts to combine yet more models proved impossible as our computers ran out of memory. Because of these problems, the use of precombined test models was mostly abandoned and parallel composition was performed on the fly for the portion of the model where it was required. While huge models could be handled in this way, the enormous state spaces still proved to be a difficult challenge for the test generation algorithms.

Later on, the modeling of the Messaging application brought to light an entirely new problem. While Messaging may be the logical application for sending text (SMS) and multimedia (MMS) messages, many others also had this capability: Contacts could send a message to a selected contact, Gallery could send a selected image as a message, and so on. In every case the actual creation and sending of the message worked in the same way, apart from some field possibly being filled from the start. Creating all but identical models for so many applications did not seem sensible; not only would that have created more work, but the redun-

dancy would also have caused difficulties in maintaining the model library. Setting all of the applications to use the same model, on the other hand, would not have allowed several applications to write messages at the same time; a dubious choice, as testing the interactions between the applications was a high priority.

The solution was the creation of *template models*. One model would act as a template, and the individual applications would automatically receive suitably modified copies of the template for their own use. The template system is entirely separate from our informal action machine classification, as any kind of action machine can be made into a template. Once the method was in use, we went back to several of the applications we had modeled earlier and converted their models into templates where possible. However, template models have since been replaced with another method called *linked models*, thanks to the automated model management features of Model Designer. Linked models remove the need for a separate template, with the equivalent models connected directly to each other.

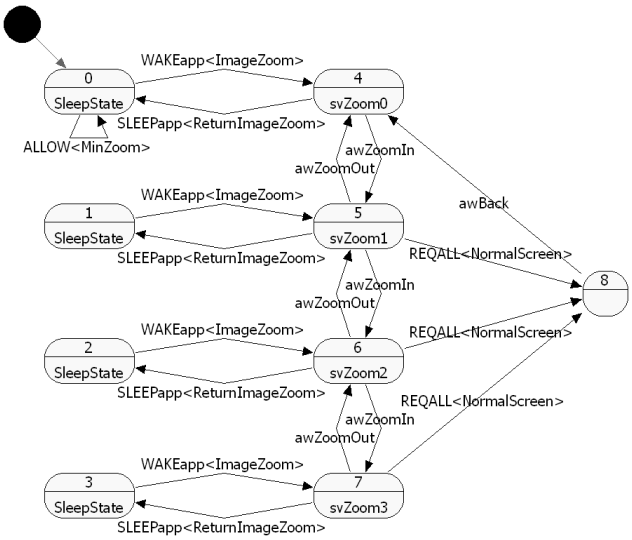

Figure 2. The action machine Gallery:ImagesZoom

3.3 The current state of the library

As the first big modeling phase reached its end, we had all the major pre-installed applications of our S60 phones modeled. In some cases the models were somewhat speculative in nature, as we did not yet have actual adapter support for all the functionality required; we could not handle multiple phones in test runs, for example. However, all the changes made since then have been minor in nature, as the broad lines of the applications had been correctly modeled.

An example of an action machine from the library can be seen in Figure 2, which shows the model for the zoom functionality in the images section of the Gallery application. According to the informal action machine classification presented earlier, the zoom action machine is a memory model: the figure shows four different levels of zoom in which the image may be set and left as the zoom model rescinds control. The zoom level may be increased or decreased incrementally, and from all levels except the first (no zoom) it is possible to return directly to the first. However, the latter functionality is only available when the image is in normal screen mode, as opposed to full screen mode. The reason for the restriction is that in full screen mode the key press which would reset the zoom level leaves full screen instead; the zoom model has to take this into account.

A refinement machine for the zoom action machine is presented in Figure 3. As is usual with refinement machines, the initial state at the center is the single ready state from which all the refinement loops for the action words begin. Also note that there are refinements for each of the state verifications defined in the action machine, with which one can verify that the SUT does indeed currently have the expected level of zoom.

At the time, the model library contained 11 different applications: Calendar, Contacts, File Manager, Gallery, Log, Messaging, Music Player, Notes, RealPlayer, Telephony (the phone call application) and Voice Recorder. The applications were modeled in some 110 action machines, with a corresponding number of refinement machines. Separately, the action machines contained about 1300 states, 1700 actions (perhaps 40% of them action words) and 3200 transitions. Refinement machines added roughly 3000 states, 3000 actions and 4100 transitions to the totals. The main modeling effort had taken about two months, starting completely from scratch; another month was spent in thoroughly checking and debugging the models and their interactions.

Ideally, we might have used the whole model library at once to run some extremely varied tests, since all significant application interactions had been modeled. However, the sheer size of the combined test model proved too much for our tools. Computational limits prevented us from calculating the exact magnitude of the combined state space of the model library, but even careful estimates gave a state count in excess of 10^{21}. And while the size of the test model was not a great limit for the on-the-fly parallel composition, it brought some of our more sophisticated test generation tools practically to a halt. This forced us to limit the model components in use to those belonging to the application(s) under test or, even better, just the necessary models.

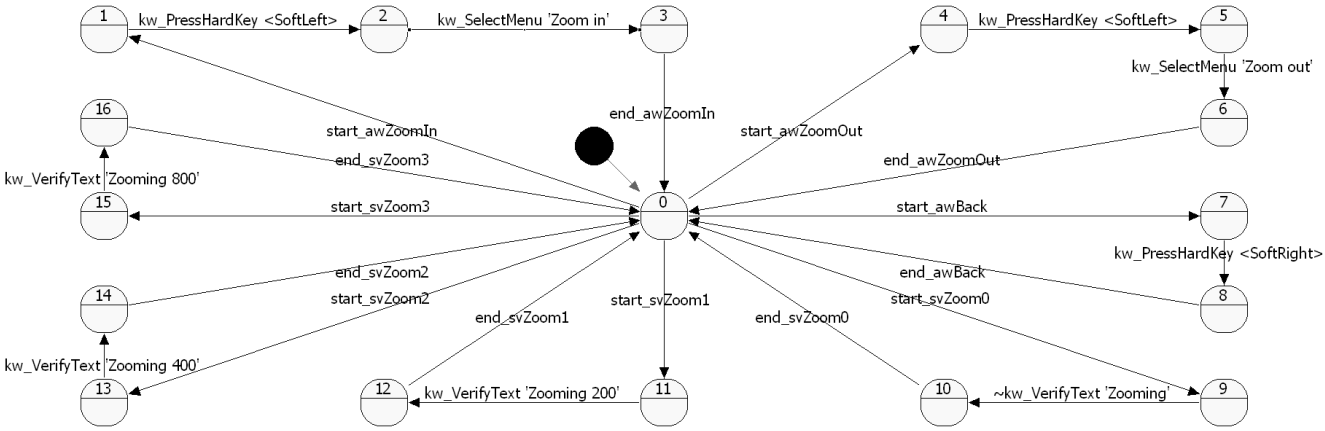

Figure 3. The refinement machine Gallery:ImagesZoom-rm

4 Conclusions

In this paper we have described how we developed a domain-specific GUI test model library for testing of Symbian S60 smartphone applications. The library is available under the MIT Open Source License. The library already covers the basic functionality of most pre-installed applications and can be easily extended to cover new applications. The interested reader can view a video of a test session using the library at `http://www.cs.tut.fi/~teams`.

Such a library could be used not only for testing individual applications but also for setting quality requirements for new third-party applications. Ensuring high quality and interoperability with the default applications is in the mutual interests of the application developers, as well as the device and platform manufacturers. While some quality assurance programs are already in effect [11], we believe that model-based approaches can significantly contribute to finding defects that are out of reach of conventional approaches.

As already suggested in [9], test modeling seems like a very effective defect finding method, especially when combined with the idea of exploratory testing. Modeling needs accuracy and makes the modeler think about different interleavings between concurrent applications. Obviously, it is also possible to start test modeling much earlier in the process than the actual test execution, since automated test execution works only with a SUT of sufficient maturity. In the future, test execution is expected to reveal more defects.

Acknowledgments

Funding from Tekes, Nokia, Conformiq Software, F-Secure, and Plenware, as well as the Academy of Finland (grant number 121012) is gratefully acknowledged.

References

[1] Python Programming Language homepage. `http://python.org/`. Cited Jul. 2007.

[2] H. Buwalda. Action figures. *STQE Magazine, March/April 2003*, pages 42–47, 2003.

[3] ETSI. Conformance test specification for SIP – part 3: Abstract test suite (TTCN-3 code). Available at `http://portal.etsi.org/docbox/EC_Files/EC_Files/ts_10202703v030101p0.zip`, 2003.

[4] M. Fewster and D. Graham. *Software Test Automation: Effective use of test execution tools*. Addison–Wesley, 1999.

[5] A. Jääskeläinen. A domain-specific tool for creation and management of test models. Master's thesis, Tampere University of Technology, Jan. 2008.

[6] A. Jääskeläinen, M. Katara, A. Kervinen, H. Heiskanen, M. Maunumaa, and T. Pääkkönen. Model-based testing service on the web. In *Proc. TESTCOM/FATES 2008*, LNCS, Tokyo, Japan, June 2008. Springer. To appear.

[7] M. Katara and A. Kervinen. Making model-based testing more agile: a use case driven approach. In *Proc. Haifa Verification Conference 2006*, number 4383 in LNCS, pages 219–234. Springer, 2007.

[8] M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Satama. Towards deploying model-based testing with a domain-specific modeling approach. In *Proc. TAIC PART 2006*, pages 81–89, Windsor, UK, Aug. 2006. IEEE CS.

[9] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. Model-based testing through a GUI. In *Proc. FATES 2005*, number 3997 in LNCS, pages 16–31. Springer, 2006.

[10] Symbian. `http://www.symbian.com/`. Cited May 2007.

[11] Symbian. Symbian Signed. `http://www.symbiansigned.com/`. Cited Dec 2006.

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

[13] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, pages 429–528, London, UK, 1996. Springer-Verlag.

[P3] With kind permission from Springer Science+Business Media: Proceedings of the 4th International Haifa Verification Conference on Hardware and Software, Verification and Testing (HVC 2008), “Synthesizing Test Models from Test Cases”, ser. Lecture Notes in Computer Science vol. 5394, 2009, pp. 179–193, A. Jääskeläinen, A. Kervinen, M. Katara, A. Valmari, and H. Virtanen,

Synthesizing Test Models from Test Cases

Antti Jääskeläinen, Antti Kervinen, Mika Katara, Antti Valmari, and Heikki Virtanen

Tampere University of Technology
Department of Software Systems
P.O.Box 553, FI-33101 Tampere, FINLAND
{antti.m.jaaskelainen,firstname.lastname}@tut.fi

Abstract. In this paper we describe a methodology for synthesizing test models from test cases. The context of our approach is model-based graphical user interface (GUI) testing of smartphone applications. To facilitate the deployment of model-based testing practices, existing assets in test automation should be utilized. While companies are interested in the benefits of new approaches, they may have already invested heavily in conventional test suites. The approach presented in this paper enables using such suites for creating complex test models that should have better defect detection capability. The synthesis is illustrated with examples from two small case studies conducted using real test cases from industry. Our approach is semi-automatic requiring user interaction. We also outline planned tool support to enable efficient synthesis process.

1 Introduction

Model-based software testing [1] has several obvious advantages over conventional test suite testing where test cases are crafted manually. For instance, on-line tests generated from state machines can reach significantly higher coverage in testing non-deterministic systems under test (SUTs) than linear and static test suites. Moreover, maintenance of large test suites is more difficult when changes occur in the SUT. Frequent changes are common especially in graphical user interface (GUI) testing that is typically used to check the functionality of the SUT from the perspective of the end users before a release is made.

The problems with conventional test automation approaches have resulted in many bad experiences, and manual testing is still widely considered as the primary quality assurance method at the system and acceptance level testing of GUI-intensive software [2]. While unit and integration level test automation can significantly improve code quality and enable efficient refactoring, system level test automation entails much more challenges. This is due to the *domain-specific nature* of system level testing; at the unit and integration levels all SUTs seem more or less similar, depending on the programming language used; the same white-box testing and static analysis techniques work across different domains. At the system level, however, the context comes into play: testing a banking system can be quite different from testing a set-top box.

The deployment of model-based system testing has been hampered in many contexts in spite of its many benefits [3, 4]. In our earlier work, we have developed a domain-specific solution to the GUI testing of S60 [5] smartphone applications that should be

easier to deploy than more generic methodologies [6, 7]. The approach consists of a domain-specific modeling language based on LSTSs (Labeled State Transition Systems) augmented with S60 specific restrictions, a model-library containing test models for the basic smartphone applications such as calendar, contacts, camera, and messaging, and tools for on-line test generation. In on-line testing, the idea is to generate tests while they are executed, thus testing can be seen as a game between the test automation system and the SUT [8].

In the course of developing our approach we have identified another problem in deployment: companies may have invested huge sums of money to craft test suites and thus can be unwilling to invest to the development of test models replacing the former way of working. Thus, in order to facilitate the deployment of our approach, we have developed a semi-automatic method for synthesizing test models from test cases. This enables utilizing the existing assets when moving from test suite testing to model-based one. The method is domain-specific to enable a higher level of automation in the synthesis and promote the usefulness of the resulting models. However, a similar method could presumably be developed for some other domain, using similar principles.

In this paper we describe the method and the case studies we have conducted. In addition, since model synthesis is quite different from the traditional way of creating models, and we compare the synthesized model to a one crafted by hand using a top-down approach [9]. A tool support for the synthesis is also outlined; its implementation will be future work. The remainder the paper is structured as follows: Section 2 describes the context of our contributions, i.e., model-based GUI testing of mobile applications. Then, we move on to present our approach for model synthesis in Section 3. Sections 4 and 5 present the case studies and discuss the results and the future work.

2 Model-Based GUI Testing of Mobile Software

Action words and keywords [10, 11] are commonly used concepts in software test automation, especially in GUI testing. The basic idea is to separate different concerns: *what* are the important actions to be tested and *how* they are implemented. Action words are high level descriptions of functionality; in the smartphone context there can be different action words for opening the messaging application, taking a photo with the camera, or adding a new contact, for instance. Keywords, on the other hand, specify the exact sequence of events that are needed to implement the functionality described by an action word. In S60 GUI, for instance, there can be multiple ways of opening a messaging application (short cut, menu, some other application). Each of the different ways can be encoded as a separate sequence of key strokes that accomplish the action. Furthermore, to receive input from the SUT, some keywords can be dedicated to verifying that a given text string is found on the display, for instance.

The main benefit of action words and keywords is in enabling non-technical testers to design action word level tests without deep knowledge of the underlying keyword implementations. Moreover, they ease the tedious maintenance tasks often hindering the use of GUI test automation; in many cases minor GUI changes can be restricted to the keyword level. Action words and keywords can be used in conventional approaches so that the keywords are implemented as a library of functions, one function for each

keyword. Action words are then specified using spread sheets, for instance, that list the sequences of keywords needed to implement the corresponding action word. Finally, test cases can be encoded as sequences of action words using spread sheets as in the previous step.

However, linear and static tests are limited in their ability to find new defects. Thus, the true power of the action words and keywords is realized when combined with automatic test generation based on behavioral models. For this purpose, we have chosen to use Labeled State Transition Systems (LSTSs) [12] for test modeling. LSTS is an extension of the more common Labeled Transition System (LTS) formalism where labels have been added to states as well as transitions. Action words and keywords are used as transition labels in the models. The formal definition for LSTS is as follows:

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.*

Notation of internal transitions makes no sense in test modeling, because our behavioral models have to be strictly deterministic for test generation. Our definition differs from the original one in that respect.

Actions can be divided into three categories according to how they deal with the SUT: *input*, *output* and *setup actions*. Input actions correspond to user input, and output actions get information from the SUT. Setup actions affect the SUT just as input actions, but in ways not accessible to an ordinary user. Setup actions might, for example, directly create or remove files in memory or alter internal settings. Action words often combine aspects of more than one category, whereas keywords usually fall neatly into one or another.

To enable modular and compositional test modeling, *parallel composition* is used for combining test model components. The parallel composition of LSTSs [12] is based on a rule set explicitly defining which actions are executed synchronously. 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 its execution. The following definition is slightly modified in two respects; internal transitions are not needed and handling of state propositions is made more straightforward:

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, with* $\forall i, j; 1 \leq i < j \leq n : \Pi_i \cap \Pi_j = \emptyset$. *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* $\|_R (L_1, \ldots, L_n) = repa((S, \Sigma, \Delta, \hat{s}, \Pi, val))$, *where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \Sigma_R$
- $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ *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* $s_i = s'_i$
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$
- $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$
- $val((s_1, \ldots, s_n)) = val_1(s_1) \cup \cdots \cup val_n(s_n)$
- *repa is function restricting LSTS to contain only the states which are reachable from the initial state* $\hat{s}$.

Parallel composition offers tools for implementing rudimentary variables, which the basic LSTS formalism lacks. A variable can be created as a single component model, whose states correspond to different values. The actions in such a *variable model* are synchronized to those of the other component models so that different values allow different actions. These synchronized actions can be used to test the value of the variable or to change it. The idea of using compositional test modeling and separate variable components is motivated by existing tools, that proof the concept [13].

To hide the complexity inherent in test models and test generation algorithms, and so to facilitate the deployment of our model-based testing methodology, we have introduced a web based testing service [7]. The idea is that test service users can order tests using a simple web interface specifying the desired coverage requirements. The coverage requirements are then used for driving on-line test generation based on an extensive model library containing test models for basic S60 applications [9].

We believe that such a service can greatly ease the adoption of model-based testing in smartphone application testing. However, companies have existing assets in conventional test suites, and they might prefer to utilize them when migrating from traditional test suite based automation to a model-based one. This led us to research an approach for synthesizing test models from test cases.

3 Synthesis of Test Models

The synthesis process we have developed allows the creation of a single test model from a number of test cases. The cases must be strictly linear to begin with; they should also be specific in detail. The resulting model will have the same level of abstraction (action word/keyword) as the original cases. Test cases which verify the state of the SUT often may be easier to handle, but the process is designed to also work with few or no verifications.

The process has five distinct phases. In the first phase the relevant actions are listed and parameterized. The second phase consists of creating variables to hold some of the state information of the SUT. The third phase takes care of the initialization sequence of the SUT. In the fourth phase recurring states within the test cases are marked and labeled. Finally, the fifth phase sees the test cases merged together with the variables and the initialization to form a new test model.

Although the phases are presented consecutively, their order is not fixed. Only the merging phase is dependent on the others and must therefore be performed last. The others may be performed in any order, and it may even be a good idea to consider them side by side. Throughout the process description we will present a running example, starting with the three imaginary action word level test cases in Figure 1. In the first the phone

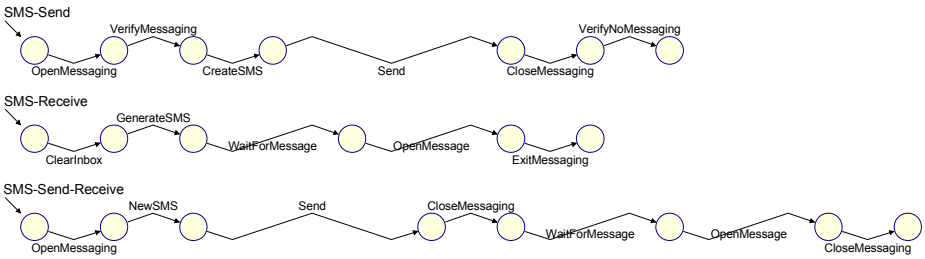

Fig. 1. The three initial example test cases.

sends an SMS to itself, in the second it receives and opens an automatically generated SMS, and in the third it first sends an SMS and then receives it. Note that the actions *CreateSMS* and *GenerateSMS* perform the same task, as do the actions *CloseMessaging* and *ExitMessaging*. They are used to demonstrate the effects of different actions sequences corresponding to the same functionality.

3.1 Action Definition

The first thing to do is to list all the actions used within the source test cases. Possible parameters should not be included. Once listed, each action is assigned two values: weight and idempotence status.

An action's weight represents its situational specificity. An action with a high weight is one whose execution with a certain parameter is likely to lead the SUT into the same state every time. This may be either because the action is only executable in very few states or because it resets parts of the SUT. An action with a low weight, on the other hand, is one which can be executed in many different situations and whose effects depend on the current situation. Weights are used in the merging of test cases. If identical action sequences taken from different test cases or different parts of the same test case have a high combined weight, it is likely that the sequences are related to the same functionality of the SUT. If this is the case, the two test cases may be merged at the points after the sequences, giving them two different ways to proceed from that point. The comparison is made with sequences instead of single actions because a long series of actions is likely to be far more situationally specific than any of its actions individually.

Actions may be marked as idempotent. The execution of an idempotent action leaves the SUT in the state it had before the execution. Most idempotent actions are used to get information out of the SUT. An idempotent action can be discarded from a test case without breaking it, although the testing value of the case may drop.

Finding the right weights is not an exact process. Action words should generally be given high weights, whereas keywords' weights vary case by case. In our running example all actions are action words. This means they have a high situational specificity, and we can give all of them maximal weights. VerifyNoMessaging and VerifyMessaging are idempotent, the rest are not.

Following are some examples with keywords: A keyword for resetting the SUT has a very high weight, since by default it always leaves the SUT in the same state. It is clearly not idempotent. A keyword which verifies that a given text is visible on the screen is idempotent and has a relatively high weight, since the same text does not very often occur in different situations. A keyword indicating that nothing should be done for a period of time has minimal weight, since waiting is always possible. It is not idempotent, because it is generally used in situations where the state of the SUT is expected to change during the wait.

3.2 Variable Definition and Integration

Embedding a part of the state of the SUT into variables is an important part of the synthesizing process. Without separate variables, the states of the test cases may contain so much information that they can never be merged together. The first, most difficult task is to identify the variables to be created. As a general rule, those properties of the SUT which are independent of the current screen of the SUT yet affect execution should be moved to variables. Having too few variables reduces the number of potential merge points and thereby limits the functionality of the final model. Too many variables mean more work in creating them and may increase the size of the final model, but should not reduce its quality.

After the variables have been determined, each is given a number of possible values. The number of values should be kept as small as possible, because they can cause exponential growth in the final model. Once the values have been chosen, each may be given one or more setup actions as *assignment actions*. In the final model, the execution of the assignment action will automatically set the variable into the designated value. A single action may act as an assignment action for multiple values, as long as they do not belong to the same variable. Finally, for each variable one of its values may be chosen as the initial value. The initial value should either have an assignment action or be otherwise guaranteed when testing begins. A variable may be left uninitialized, but then no action based on it can be taken until it has been given a value during a test run, and the size of the resulting model is also somewhat increased.

Once the variable definitions are ready, variable models are created for them. For this purpose we have made a simple Python script which reads in the variable definitions in CSV (Comma Separated Values) format and automatically produces an LSTS for each variable. The script also creates a *variable initialization model* which can set the variables to specific values before a test run by using the assignment actions.

The ready variables must be integrated into the test cases. This is performed by adding preconditions and postconditions to the actions in the test cases. Preconditions specify the values of the variables necessary for the successful execution of the action. Postconditions, conversely, define the changes of values caused by the execution of the action. Assignment actions do not require explicit postconditions, but are synchronized directly into appropriate variables. For optimal result, pre- and postconditions should be placed right around the relevant action, not around a whole action sequence.

In our running example, we create a single variable to record whether there is a message on its way to the phone, so that we will be free to merge the test cases at the main screen, regardless of whether messages have been sent or not. We use GenerateSMS

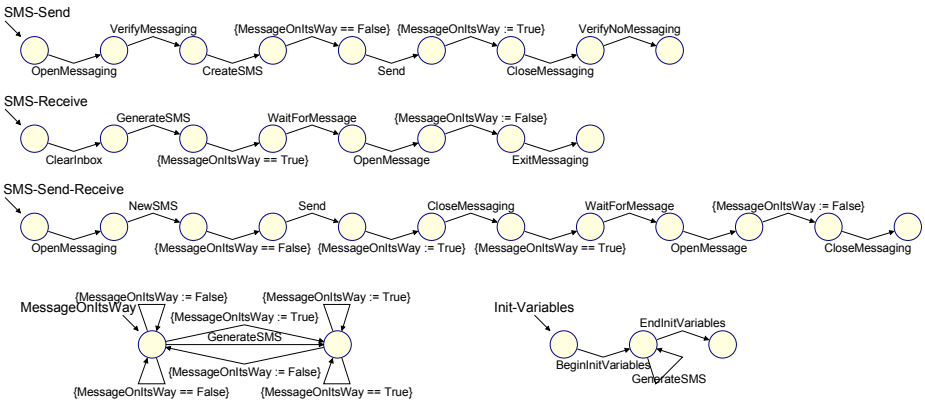

Fig. 2. The example test cases with pre- and postconditions added.

as an assignment action for the value True and pick False as the initial value, which should be safe for a new test run. Figure 2 shows the variable model and the variable initialization model, and above them the test cases with pre- and postconditions marked with braces.

3.3 Initialization Sequence Definition

In order to automatically set the SUT into its initial state before a test run, an initialization sequence is defined. The sequence contains those setup actions which should always be executed before a test run. They could, for example, reset the SUT, disable features that might interfere with testing, and create suitable data. Variable initialization should not be included here. As a rule, all setup actions should be within the initialization sequence or act as an assignment action for a variable. If a setup action belongs to neither group, more variables might be needed.

The rest of the initialization phase could be performed automatically with the information from the earlier phases, although we do not currently have tools for it. The initialization sequence is made into a *general initialization model*. All non-idempotent setup actions are removed from the beginnings of the test cases (by now they are all in the general initialization model or the variable initialization model), and synchronization is added to connect them into the initialization models.

The changes made into the test cases in the example are very minor, as Figure 3 shows. The only setup action is ClearInbox, which has been moved into a model of its own.

3.4 State Label Definition and Assignment

The existence of the variables allows the test cases to be merged with relative freedom, but there is no guarantee that suitable merging points can be automatically identified.

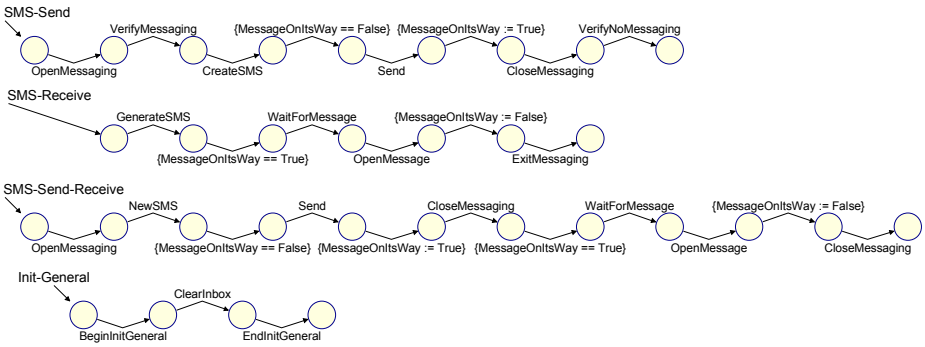

Fig. 3. The example test cases with setup actions separated.

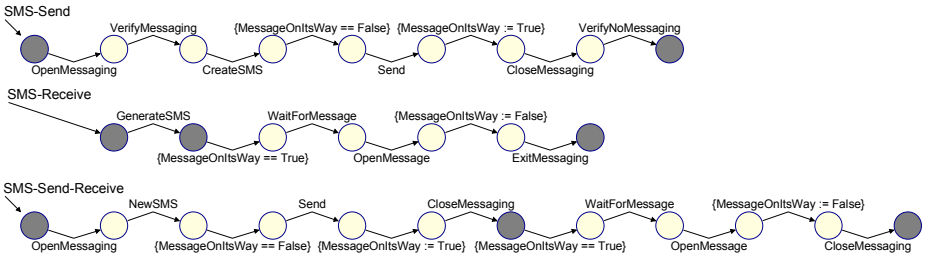

Fig. 4. The example test cases with filled states marking the main screen.

For this purpose *state labels* are added into the test cases. The important states of the SUT are identified and a name is given to each. Especially important are the starting and ending states of the test cases (ideally the same state); the basic states of other major SUT screens visited during the test cases are also good choices. Properties included in variables should be ignored.

Once the important states have been selected, state labels with suitable parameters are placed into test cases at every point in which the SUT is in a chosen state. The state labels can be handled as LSTS attributes; alternatively they can be interpreted as idempotent actions with maximal weights. Either way, merges will always be attempted at their points of execution. They can be easily removed from the final model so that they do not interfere with its execution.

In our example, we decide that the only noteworthy state is the main screen of the phone and label it, as shown in Figure 4. The states in question have been filled.

3.5 Merging of the Component Models

Now that the test cases have been prepared we can perform the actual merging. This is done with the merger program, which looks for identical sequences of sufficient weight

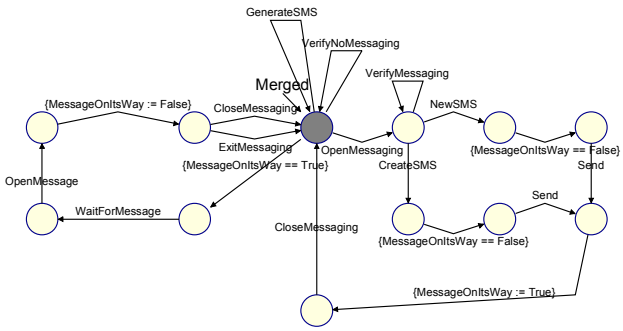

Fig. 5. The model merged from example test cases.

within the test cases and suggests merging their destination states. The program may also offer false suggestions, i.e. merges that would result in an erroneous model. Because of this, the legality of each merge must be manually checked to ensure the validity of the model. The merging results in a *control model* which contains the functionality of the original test cases, but without the information encoded into variables. The merged model, variable models and initialization models are then passed through parallel composition, which creates an executable test model.

While the test model obtained this way is usable, it may pose difficulties for test generation. That is because the model is likely to contain many paths leading to deadlocks, i.e. states with no outgoing transitions, resulting either from a denied precondition or the end of a test case that could not be merged anywhere. The model may be cleaned by removing all the dead paths, but this is not always a good idea. If the test cases could be looped back into themselves and deadlocks occur only or mostly in places where a precondition fails, the clean-up procedure should be safe to perform. Conversely, if many test cases ended in unique states and caused deadlocks at the end, the clean-up could remove relevant functionality. In this case the model may be better left as-is, and the test generation algorithm must take care not to guide the execution toward a deadlock prematurely.

Figure 5 shows the model obtained from our example test cases. Merges have been performed at matching actions and state labels. Adding the initialization models and the variable model in parallel composition results in the usable model depicted in Figure 6. It would seem that in this case the cleaned model (Figure 7) would be more useful, since the dead end on the right likely serves no practical purpose; it depicts a situation where a new message is created with one already on its way, causing the preconditions to block its sending.

4 Case Studies

We have performed two small scale case studies to test our synthesizing methodology. The original sequences were linear keyword level test cases picked from a much larger

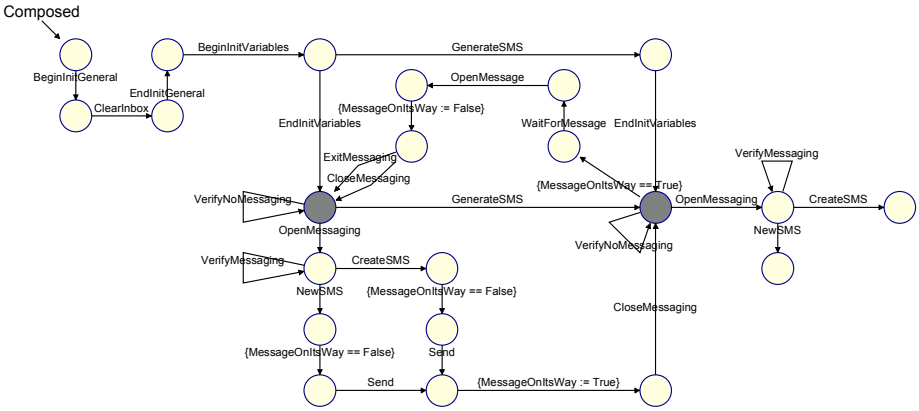

Fig. 6. The example model after parallel composition.

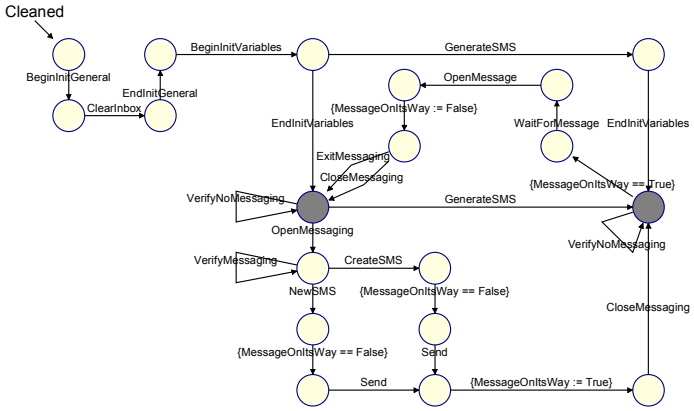

Fig. 7. The final, cleaned model.

set of test cases for S60 applications developed by one of our industrial partners. The first case study used seven test cases for the Phonebook application. The second one had nine for the Messaging application, concentrating on short and multimedia messages (SMS and MMS). Both case studies used the same set of 30 keywords. The Phonebook test cases had 193 actions altogether, the Messaging test cases 363. Three of the test cases for the Messaging case study can be seen in Figure 8, with some changes made for readability and to adapt them for a single phone.

Both case studies used the same set of keywords, which we were already familiar with from our earlier work. Giving keywords their weights was therefore easily done, though the values were somewhat arbitrary; we had yet to perform enough experiments to find the best values. The Phonebook case proved to require seven variables, six to hold information about existing contacts and groups and one for incoming messages. The Messaging case required six variables, two for the existence of messages and reports and the rest for various settings. The first case labeled the idle state and the contacts and groups screens, the latter labeled the idle state and the screens for SMS and MMS writing.

After the merge, the Phonebook model had 126 and the Messaging model 192 states. Parallel composition and cleanup brought state counts to 12523 and 2327, respectively. The Phonebook case shows the potentially exponential growth caused by variables. This happened because the variables controlled relatively small portions of the model and had little to do with each other. Conversely, the variables in the Messaging case were interconnected to some degree, and affected control to a much greater extent; for example, many individual test cases specified certain settings before sending a message. As a result, large portions of the control model were reachable only with certain variable values. Figure 9 shows an overview of the final Messaging model, illustrating its scope and complexity. Although the models are too large for human understanding, their size is not a problem for our automated test generation tools.

The quality of the final models appeared to be comparable to the test models in our test model library [9] created by hand from scratch, although not quite equal to them. The synthesized models contained less functionality, but this was a result of the original choice of test cases, not a failing of the method itself. A notable difference was the higher granularity of the synthesized models: often actions which could be performed separately in hand-made models were forcibly chained together in synthesized ones. However, this tendency did not seem to reach truly detrimental levels, and the number of possible action sequences was still magnitudes higher than in the original linear test cases. The final difference between the synthesized models and our old models was that keyword level test cases naturally became a single keyword level model, not a combination of keyword and action word level models as in our model library. The action word level might be added using the bottom-up modeling technique presented in [6]. Presumably action word level test cases could be combined into an action word level model and the action words then refined as in original test cases, though we have yet to attempt that.

In both case studies, most of the effort during the synthesizing process went into variable definition and integration. In the Phonebook case, this was mostly manual work: the variables were simple, but referenced often. With Messaging the situation

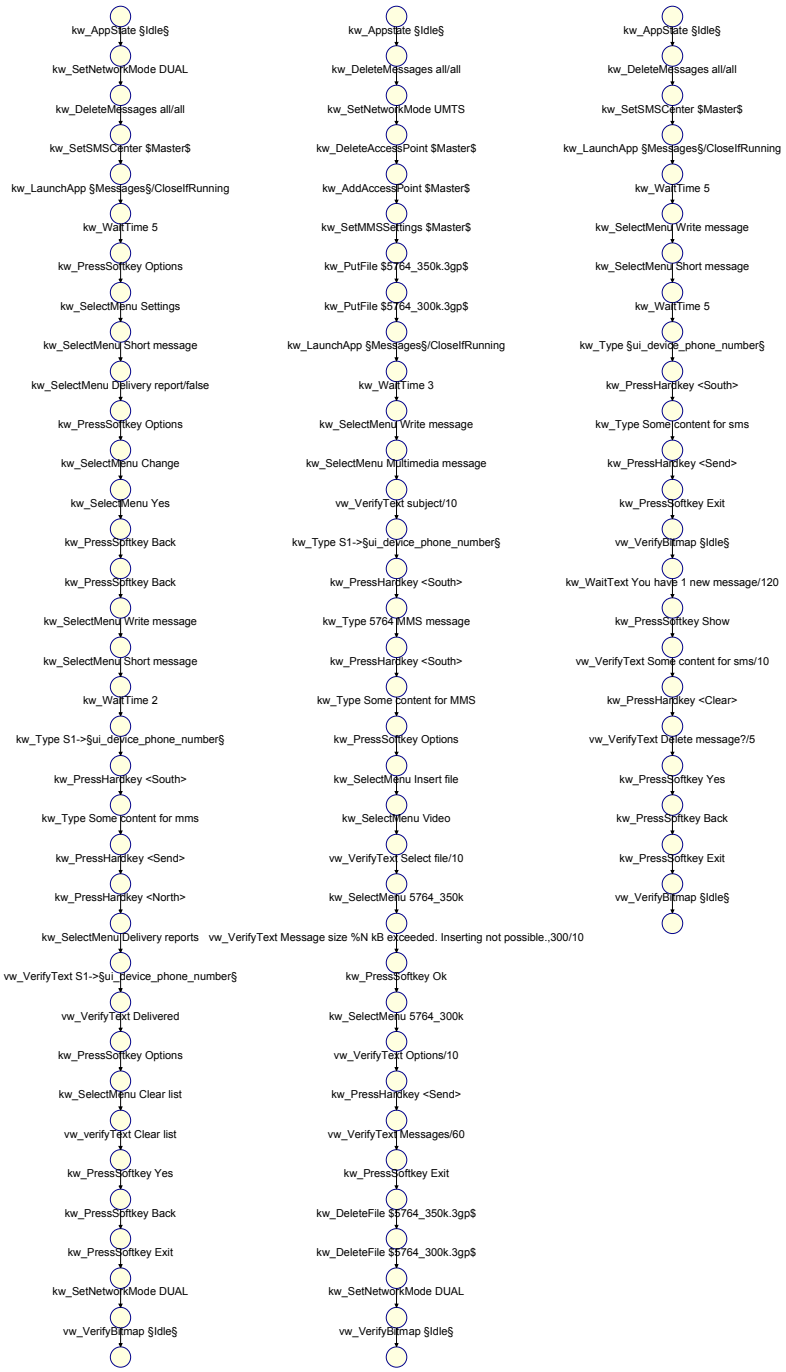

Fig. 8. Three of the nine Messaging test cases.

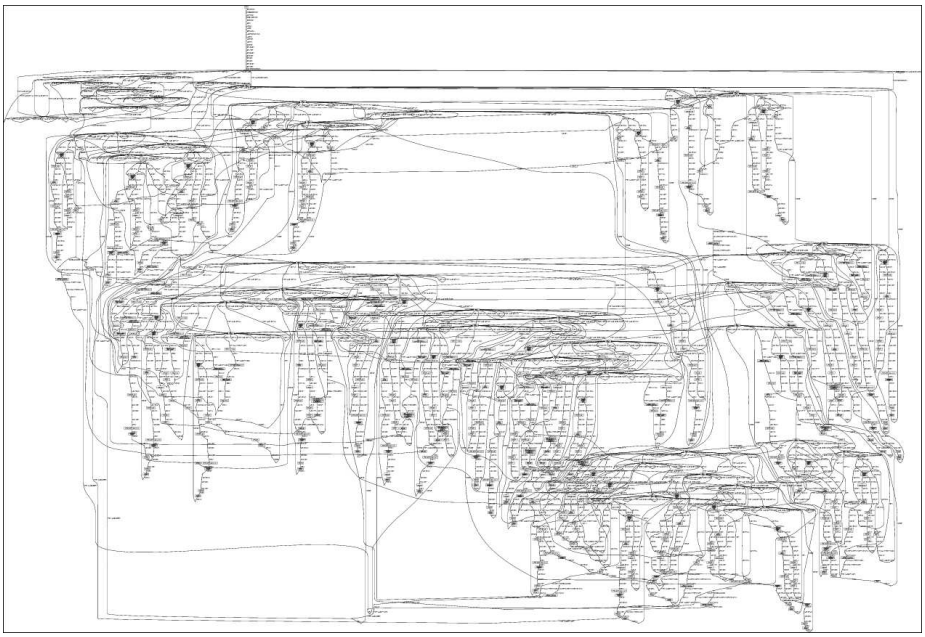

Fig. 9. The final Messaging model.

was different. There much time was spent in deciding what exactly should be modeled into variables, and how exactly would they be integrated into the test cases. Placing the pre- and postconditions also took considerable time, mostly because the complexity of the variables demanded great care in integrating them into control. We found merging to be relatively easy, but it might pose more difficulties to someone not used to test modeling. It definitely requires some understanding of the implemented variables, which implies that the whole process might be best performed by a single person.

Both of the case studies were performed by a single person and each required less than a day to complete. It seems quite reasonable to us that with good tools a person familiar with the process could synthesize a model of considerably greater size within a single day. That would be notably faster than creating a comparable test model from scratch, and would not require a similar expertise in modeling. Fortunately the most time-consuming phase, variable definition and integration, seems likely to scale reasonably well with the number of test cases (probably linear effort or less). The least scalable phase by far is merging of the component models (potentially quadratic or even exponential effort), which at least might be fully automatable.

5 Discussion

In this paper we have described an approach for synthesizing test models from test cases. In addition, we presented the results of two small case studies where the approach

was applied for creating test models from existing test cases in the domain of S60 GUI testing. The synthesis is semi-automatic and thus requires user interaction to achieve useful results. A tool supporting this interaction was also sketched.

Our approach is domain-specific in the sense that the set of keywords and the corresponding weight values must be decided based on the domain knowledge. In our case studies this was easy because the same person who had built our model library conducted the experiments. However, the other phases of the synthesis process should be applicable also in other contexts.

There exists a large body of knowledge about the synthesis process. While most, if not all, of the existing approaches have been originally developed for design, analysis and code generation purposes, they may be useful for test model synthesis also. Amyot and Eberlein have compared twenty-six solutions for constructing design models from scenarios [14]. Moreover, Liang, Dingel, and Diskin have developed comparison criteria for comparing different algorithms and applied the criteria to compare twenty-one different approaches [15]. However, it seems that domain knowledge can improve the synthesis; we first experimented with a more generic approach [16], but decided to develop our own to better fit the needs of our context. An extensive study would be needed to analyze the other existing approaches for their applicability to test model creation, but this lies outside the scope of this paper.

Some of the currently manual phases in our synthesizing process might be automated, most notably initialization and parts of modeling of variables. The action weights and state labels must be set manually. The defining and integration of variables also requires user input, but actual variable models can be created automatically. It might also be possible to automate merging totally, not just finding the potential merge points. In the two case studies, potential merge points occurring at state labels were always mergeable; this seems likely to be a general rule, as long as the labels have been placed well. The merge points based on action sequences varied, some being mergeable and others not. However, in these cases the sequence merges did nothing that could not have been replicated with well-placed state labels. Based on these observations, it might be possible to automate the merging to always merge at labels and disregard sequences altogether, but more testing is required before implementing such changes.

Although the most work-intensive part of the process, the creation and integration of variables, cannot be truly automated, it could be substantially eased by proper tools. These should offer both an easy way to define variables, preferably hiding the models altogether, and a simple method for setting pre- and postconditions. Some algorithm for suggesting potential variables would be a highly useful feature, but difficult to design.

In the future, in addition to developing tool support, there is also the need to conduct wider case studies and to compare the test coverage that can be achieved with handcrafted versus synthesized test models in actual on-line test generation.

Acknowledgements

This paper reports results of research funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq Software, F-Secure, and Plen-

ware, 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. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing: A Context-Driven Approach. Wiley (2001)
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 June 2008.
5. S60. (http://www.s60.com. Cited June 2008)
6. 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
7. Jääskeläinen, A., Katara, M., Kervinen, A., Heiskanen, H., Maunumaa, M., Pääkkönen, T.: Model-based testing service on the web. In: Proceedings of the the 20th IFIP Int. Conference on Testing of Communicating Systems and the 8th Int. Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES 2008). Number 5047 in Lecture Notes in Computer Science. Springer (2008) 38–53
8. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, USA, ACM (2004) 55–64
9. Jääskeläinen, A., Kervinen, A., Katara, M.: Creating a test model library for GUI testing of smartphone applications. In: Proceedings of the 8th International Conference on Quality Software (QSIQ 2008), IEEE Computer Society (2008) 276–282
10. Fewster, M., Graham, D.: Software Test Automation: Effective use of test execution tools. Addison–Wesley (1999)
11. Buwalda, H.: Action figures. STQE Magazine, March/April 2003 (2003) 42–47
12. Hansen, H., Virtanen, H., Valmari, A.: Merging state-based and action-based verification. In: Proceedings of ACSD 2003, the Third International Conference on Application of Concurrency to System Design, IEEE (2003) 150–156
13. Virtanen, H., Hansen, H., Valmari, A., Nieminen, J., Erkkilä, T.: Tampere verification tool. In: Proceedings of TACAS 2004, Tools and Algorithms for the Construction and Analysis of Systems, the 10th International Conference. Volume 2988 of LNCS. Springer-Verlag (2004) 153–157
14. Amyot, D., Eberlein, A.: An evaluation of scenario notations and construction approaches for telecommunication systems development. Telecommunication Systems **24** (2003) 61–94
15. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06). (2006) 5–12
16. Mäkinen, E., Systä, T.: MAS - an interactive synthesizer to support behavioral modeling in UML. In: Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society Press (2001) 15–24

Filtering Test Models to Support Incremental Testing

Antti Jääskeläinen

Tampere University of Technology, Department of Software Systems
PO Box 553, FI-33101 Tampere, Finland
antti.m.jaaskelainen@tut.fi

Abstract. Model-based testing can be hampered by the fact that a model depicting the system as designed does not necessarily correspond to the product as it is during development. Tests generated from such a model may be impossible to execute due to unimplemented features and already known errors. This paper presents a solution in which parts of the model can be filtered out and the remainder used to generate tests for the implemented portion of the product. In this way model-based testing can be used to gradually test the implementation as it becomes available. This is particularly important in incremental testing commonly used in industry.

Keywords: Model-Based Testing, Test Modeling, Model Filtering, Model Transformation, Strong Connectivity

1 Introduction

Traditionally software test automation has focused on automating the execution of tests. A newer approach, model-based testing, allows the automation of the creation of tests by generating them from a formal model which depicts the expected functionality of the system under test (SUT). An excellent approach in theory, widespread deployment of model-based testing is nonetheless hindered by a number of practical issues.

One such issue is fitting model-based testing into the product life cycle. The error-detection capability of model-based testing is based on the correspondence between the model and the SUT; a difference between the two indicates an error in one or the other. However, testing should begin before a fully functional SUT is available, which means that this correspondence is in practice broken.

The problem first appears during the early implementation of the product. The test model can be created based on the design plans, and is likely to be ready long before all the features of the SUT have been fully implemented, since modeling is a good method of static testing. In this case, the tests generated from the model may span the whole system under development, even though the SUT only contains limited functionality. Developing and updating the model alongside the product is possible but impractical; it should be possible to model the whole

system before it is fully implemented. How, then, can we use a model of the complete system to generate tests just for the current implementation?

A similar situation is encountered when the testing pays off and an error is found. Fixing the error may take some time, especially if it is particularly complicated or not very serious. Testing, of course, should be continued immediately. But how can we ensure that new generated tests do not stumble on the same, already known issue?

In these cases, the problem is that the model contains functionality that cannot be executed on the SUT, yet we need to generate actually executable tests. The magnitude of the problem depends on how the tests are generated. If the process is cheap, it may be possible to generate an overabundance of tests and discard the unfeasible ones. However, if test generation is complicated and costly, it will be necessary to ensure that as little effort as possible is wasted on unproductive tests.

This paper presents a solution based on *filtering* the test model in such a way that unimplemented or faulty functionality is effectively removed. The remainder of the model can then be used to generate tests for the implemented functionality. As new features are implemented they can be allowed into the model and test generation; as erroneous functionality is uncovered it can be filtered out until fixed. Using this method, a complete test model can be used to generate tests as soon as the product is mature enough for automatic test execution. The challenge is to ensure that the filtered model remains suitable for test generation.

The rest of the paper is structured as follows: Section 2 provides an overall presentation on our approach to model-based testing. Section 3 explains our filtering methodology in detail, and Section 4 presents a case study based on it. Finally, Section 5 concludes the paper.

2 Background

Model-based testing is a testing methodology which automates the generation of tests. This is done with the help of a *test model*, which describes the behavior desired in the tests. Depending on the approach, this may mean the behavior of the SUT or its user, or both combined.

There are two ways to execute the generated tests. In *off-line testing* the model is first used to create the test cases, which are then executed just as if they had been designed manually. In the alternate approach, *online testing*, the tests are executed as they are being generated. The latter method is especially well suited for testing nondeterministic systems, since the results of the execution can be continuously fed back into test generation, which can then adapt to the behavior of the SUT.

Our research focuses on online testing based on behavioral models. The formalism in our models is labeled state transition system (LSTS), a state machine with labeled states and transitions. LSTS is a simple formalism and other behavioral models can be easily converted into it, which allows us to create models also in other formalisms, if need be. The formal definition of LSTS is the following:

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*.*

Creating a single model to depict the whole SUT is virtually impossible for any practical system. Therefore we create several *model components*, each depicting a specific aspect of the SUT, and combine these into a test model in a process called *parallel composition*. We use a parallel composition method developed in [7], generalized from CSP (Communicating Sequential Processes) [11]. It is based on a rule set which explicitly specifies which actions are executed synchronously. The formal definition is as follows:

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*, such that* $\forall i, j; 1 \leq i < j \leq n : \Pi_i \cap \Pi_j = \emptyset$*. 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* $\|_R\ (L_1, \ldots, L_n) = repa((S, \Sigma, \Delta, \hat{s}, \Pi, val))$*, where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \Sigma_R$
- $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ *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* $s_i = s'_i$
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$
- $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$
- $val((s_1, \ldots, s_n)) = val_1(s_1) \cup \cdots \cup val_n(s_n)$
- *repa is a function restricting LSTS to contain only the states which are reachable from the initial state* $\hat{s}$*.*

The parallel composition allows us to use a relatively small number of simple model components to create a huge test model. In practice, the test model may well be too large to calculate in its entirety, so the parallel composition is usually performed *on the fly* for the needed portion of the model. The available model components comprise a *model library* [6], from which individual components can be composed into a suitable test model.

The model components are divided into two tiers corresponding to the concepts of action words and keywords [1, 4]. *Action words* define user actions, such as those commonly used in use case definitions. Accordingly, the upper tier models based on action words, called *action machines*, describe the functionality of the SUT. Action words and action machines are independent of implementation, and can often be reused in testing other similar systems.

Keywords describe UI events, such as pressing keys or a text appearing on a display. The lower tier models, *refinement machines*, use keywords to define implementations for the action words in the action machines. Refinement machines are specific to implementation, so every different type of SUT requires its own.

The execution of a keyword returns a Boolean value, which tells whether the SUT executed the keyword successfully or not. Usually a certain value is expected, and a different result indicates an error. However, in online testing of nondeterministic systems it may be reasonable to accept either value, since the exact state of the SUT may not be known. This is modeled by adding a separate transition for successful and unsuccessful execution. The actions of such transitions are *negations* of each other. These *branching keywords* allow the implementations of action words to adapt to the state of the SUT. If the nondeterminism affects the execution of the test beyond a single action word, a similar *branching action word* is needed. Such action words can be used to direct an online test into an entirely different direction depending on the state of the SUT. Branching actions do not fit well into the linear sequences of off-line testing, though, and the unpredictability especially at the action word level makes the generation of online tests somewhat more difficult.

Tests are generated with guidance algorithms based on coverage requirements. A *coverage requirement* [8] defines the goal of the test, such as executing all actions in the model or a sequence of actions corresponding to a use case. A *guidance algorithm* is a heuristics whose task is to decide how the test will proceed. A straightforward algorithm may simply seek to fulfill the coverage requirement as quickly as possible. Others may perform additional tasks on the side, such as continuously switching between different applications in order to exercise concurrency features; yet another may be completely random.

Facilitating such diverse goals and methods places some requirements for the test model. The most important of these is that the model must be *strongly connected*, that is, all states must be reachable from all other states. A test model that is not strongly connected poses great difficulties for test generation, since the execution of any transition may render portions of the model unreachable for the remainder of the test run. Coverage requirements can no longer be combined freely, since their combination may be impossible to execute even if they are individually executable. Finally, online test generation becomes effectively impossible, because the only way to ensure that the whole test can be executed is to calculate it out entirely before beginning the execution and making potentially irreversible choices.

If strong connectivity is for some reason broken, it must be restored by limiting the model to the maximal strongly connected portion of the model containing the initial state, which we will call the *initial strong component*. Unfortunately, finding the initial strong component can be difficult if the model is too large to calculate in its entirety. In particular, strong connectivity of model components does not in itself guarantee strong connectivity in the composed test model.

Ensuring the strong connectivity and general viability of the models is in the end up to the *test modeler*, who is responsible for the creation and maintenance

of the models. The *test designers*, who are responsible for the actual test runs, should be able to use the models for test generation without needing to worry about their internal structure. Such distribution of concerns relieves most of the testing personnel from the need of specialized modeling expertise [9].

3 Filtering

In this section we present our filtering method. First we go through some basic requirements for the method, and then present a solution based on those. After that, we examine implementation issues concerning the filtering process, especially regarding strong connectivity. Following is some analysis of the algorithm used in implementation, and finally an example of its use.

3.1 Basic Criteria

A method for filtering out unwanted functionality from the models should fulfill the following criteria:

1. The execution of faulty or unimplemented transitions can be prevented.
2. The model should not be restricted more than necessary.
3. The model must remain strongly connected.
4. Filtering may not require modeling expertise or familiarity with the models.
5. The manual effort involved in the process may not be excessive.
6. Filtering must be performed without modifications to the models themselves.

The first three criteria define the desired result for the filtering process. Criterion 1 is the very goal of the filtering process. Criterion 2 is likewise obviously necessary, since we want to keep testing the SUT as extensively as possible. Criterion 3 ensures that the process does not break the basic requirement placed on the test model. As a consequence, the filtering cannot be performed by just *banning* (refusing to execute) problematic transitions or actions, since such a strategy might effectively lead to deadlocks or otherwise break the strong connectivity necessary for test generation.

The next two criteria are procedural requirements. Criterion 4 requires that the filtering process can be performed with no manual involvement with the models. Ideally, the process would be carried out by test designers, who may not be familiar with the models or the formal methods involved [9]. Since the process may need to be carried out often and repeatedly, Criterion 5 states that it may not require much manual effort.

Finally, Criterion 6 is an implementation requirement. Modifying the models for filtering purposes would require extensive tool support, so that individual changes could be made and rolled back as needed, all without breaking the models. Enabling such a feature might also place additional requirements on the structure of the models.

3.2 Methodology

There are a number of potential methods by which the tester might perform the filtering of banned functionality. Most of these require additional actions in order to keep the model strongly connected, as per Criterion 3; however, with properly designed models such actions can be automated. The examined methods are:

1. Ban the execution of specific transitions of the composed test model.
2. Ban the execution of specific transitions within model components.
3. Ban the execution of specific actions.
4. Remove model components from the composition.

Actions are general labels for the events of the SUT, whereas transitions represent the SUT moving from a specific state to another through such an event; therefore, banning an individual action corresponds to banning all of the transitions labeled with it. Likewise, banning a transition from a model component may correspond to banning several transitions from the composed test model.

Method 1 fulfills all of the specified criteria except Criterion 5, where it fails spectacularly. An individual faulty transition in a model component is likely to correspond to many transitions in the test model. Even if the problem is a concurrency issue and appears only with a specific combination of applications, it is unlikely to be limited to a situation where all of the tested applications are in exactly specific states. As such, the method is thoroughly impractical.

Method 2 is more promising, since removing the faulty transition from a model component will remove all of its instances from the test model. This method is no longer minimal (Criterion 2): in case of a concurrency issue, this method may remove more functionality than is strictly necessary. However, it does not greatly limit continued testing; furthermore, a more specific method based on multiple components at once would likely require a deeper understanding of the models, violating Criterion 4. Another problem is that transitions do not have inherent identifiers, although they can be uniquely identified by their source state and action. States are only identified with numbers, whose use would at the very least require some inspection of the model components.

In practice, Method 3 works very much the same as Method 2. It may restrict the models more, but only if the model component uses the same action in multiple places, only one of which actually fails. Unlike transitions, actions are clearly labeled and test designers will work with them in any case, so they can be easily used also for this purpose.

Finally, Method 4 is also easy to use. In fact, it might well be worth implementing for other purposes such as limiting the size of the test model. However, removing whole components from the model goes against Criterion 2, since it could drastically reduce the amount of functionality available for testing. It does have one additional benefit: it is relatively easy to design the models so that the removal of a component leaves the rest of the test model strongly connected.

Of these four, Method 3, based on banning actions, appears to be the best. It does not restrict the models much more than is necessary and is quite easy to use.

It does require some additional effort in order to retain the strong connectivity of the models, though.

In contrast, Methods 1 and 2 involve serious procedural issues and in practice do not leave much more of the model available. On the other hand, Method 4 is considerably more restrictive than necessary. However, as mentioned, it may be worth implementing anyway for other reasons, in which case it can be also used to filter models where suitable.

3.3 Banning Actions

There are three implementation issues to take care of. First, we need a means to obtain a test model with individual actions removed without altering the original models, as per Method 3 and Criterion 6. Second, we must devise a method for restoring the strong connectivity of the test model (Criterion 3), since removing individual actions may break it. Third, we must take into account the branching actions, whose both branches must be retained or removed together.

The simplest way to obtain a modified test model is to create a modified copy of the rules of parallel composition such that banned actions will not show up in the test model. This method is simple to implement and limits modifications to one place. Alternatively, modified copies of the model components could be created with banned actions removed, and then composed as usual. However, such an approach would require modifications in several places, and modifying a model component is liable to be more difficult than removing rules from a list.

Ensuring the strong connectivity of the test model is more difficult. It is obviously not possible to design all models so that any actions could be removed without breaking strong connectivity. As for automation, in a general case it is not possible to determine whether a test model is strongly connected without calculating it entirely, which may be impossible due to the potential size of the model. As a solution, our filtering algorithm seeks to deduce the initial strong component from the model components and the rule set, but without calculating the parallel composition. The result is an upper bound for the initial strong component, that is, a limited portion of the original model which contains the initial strong component. The algorithm is based on the following principles:

1. an action must be banned if it labels a transition which leads away from the initial strong component of a model component
2. an action may be banned if it does not label any transition within the initial strong component of a model component
3. an action may be banned if there remain no rules which allow its execution
4. a rule may be removed if any of its component actions is banned

The first principle is the most important: leaving the initial strongly connected component of a model component cannot be allowed, since there would be no way back, and the strong connectivity of the test model would be broken. In contrast, the other three principles ban actions and remove rules which could not be executed in the test model anyway. Actions outside the initial strong

components are effectively unreachable, an action without rules does not appear in the composed test model, and a rule without all of its actions can never be applied. Therefore, these three do not limit the models needlessly. They are also not useful in themselves, but may allow greater application of the first principle.

Based on these principles, we have developed Algorithm 1 and implemented it as a part of the TEMA open source toolset [10]. The lines from 1 to 11 set the initial values for the data structures, as well as marking for handling the initially banned actions and removed rules. The loop on line 12 additionally marks for handling those actions for which there are no rules. The three main parts of the algorithm are within the loop on line 16. First, the loop on line 18 handles banned actions, removing any rule which requires them. Second, the loop on line 24 handles rules in a similar way, banning all actions for which there are no rules left. Third, the loop on line 32 calculates the initial strong components of the model components and marks for handling those actions which lead outside the component or cannot be reached within it. These three are repeated until no more actions can be banned or rules removed. The calculation of the strong components, which can be performed for example by Tarjan's algorithm [13], is the most time-consuming part of the algorithm. It is therefore only performed when no other method for progress is available.

The algorithm returns both a set of removed rules and one of banned actions; either can be used to perform the actual filtering. The list of banned actions is also useful to the modeler, since it can be used to estimate the effects of filtering. This is important because the algorithm does not necessarily yield the exact initial strong component but only an upper bound for it. The rest will be up to the modeler, who should design the models so that the bound is in fact exact, and there is no way out of the initial strong component.

The nature of the algorithm makes it easy to define not only an initial set of banned actions, but also one of removed rules. This may be occasionally useful, for example to remove some kinds of actions across the model components.

Specific model semantics may require some changes or additions to the basic algorithm. Branching actions are such a case: if one branch gets banned, the other one must, too. To take this into account, we modify the algorithm such that every time an action is marked to be handled, we check for other branches and mark them also. It might also be useful to allow the modeler to define similar dependencies on a case-by-case basis, where strong connectivity demands it; we have yet to implement such a method, however.

3.4 Analysis

Following is a brief analysis of the time requirements of Algorithm 1. For an arbitrary model component $m \in M$, we will mark $m = (S_m, \Sigma_m, \Delta_m, \hat{s}_m, \Pi_m, val_m)$. All set operations used in the algorithm (addition and removal of elements, check for membership or emptiness) can be performed in amortized constant time.

The handling of each rule requires $O(|M|)$ time: it may get marked for handling by each action it refers to, and may have to mark for handling each of

Algorithm 1 The filtering algorithm for the set of model components M composed with the rules R, with the rules $remove \in R$ initially removed and the actions $ban(m) \in \Sigma_m$ of model components $m \in M$ initially banned.

$banned\_actions, unhandled\_actions, removed\_rules := \emptyset$
$unhandled\_rules := remove$
$changed\_models := M$
for all model components $m \in M$ **do**
  for all actions $a \in ban(m)$ **do**
    add (m, a) to $unhandled\_actions$
  for all actions a of m **do**
    $remaining\_rules(m, a) := \emptyset$
for all rules $r \in R$ **do**
  for all actions a of model components m in r **do**
    add r to $remaining\_rules(m, a)$
for all model components $m \in M$ **do**
  for all actions a of m **do**
    if $remaining\_rules(m, a) = \emptyset$ **then**
      add (m, a) to $unhandled\_actions$
while $unhandled\_actions \neq \emptyset$ or $unhandled\_rules \neq \emptyset$ **do**
  while $unhandled\_actions \neq \emptyset$ or $unhandled\_rules \neq \emptyset$ **do**
    for all model-action pairs $(m, a) \in unhandled\_actions$ **do**
      for all rules $r \in remaining\_rules(m, a)$ **do**
        if $r \notin removed\_rules$ **then**
          add r to $unhandled\_rules$
        add (m, a) to $banned\_actions$
    $unhandled\_actions := \emptyset$
    for all rules $r \in unhandled\_rules$ **do**
      for all actions a of model components m in r **do**
        remove r from $remaining\_rules(m, a)$
        if $remaining\_rules(m, a) = \emptyset$ and $(m, a) \notin banned\_actions$ **then**
          add (m, a) to $unhandled\_actions$
          add m to $changed\_models$
      add r to $removed\_rules$
    $unhandled\_rules := \emptyset$
  while $changed\_models \neq \emptyset$ and $unhandled\_actions = \emptyset$ **do**
    $m :=$ any element from $changed\_models$
    remove m from $changed\_models$
    $reachables := \emptyset$
    $isc :=$ the initial strong component of m with banned actions removed
    for all transitions (s, a, s') of m **do**
      if s within isc **then**
        add a to $reachables$
        if s' not within isc and $(m, a) \notin banned\_actions$ **then**
          add (m, a) to $unhandled\_actions$
          add m to $changed\_models$
    for all actions a of m **do**
      if $a \notin reachables$ and $(m, a) \notin banned\_actions$ **then**
        add (m, a) to $unhandled\_actions$
        add m to $changed\_models$
return $removed\_rules, banned\_actions$

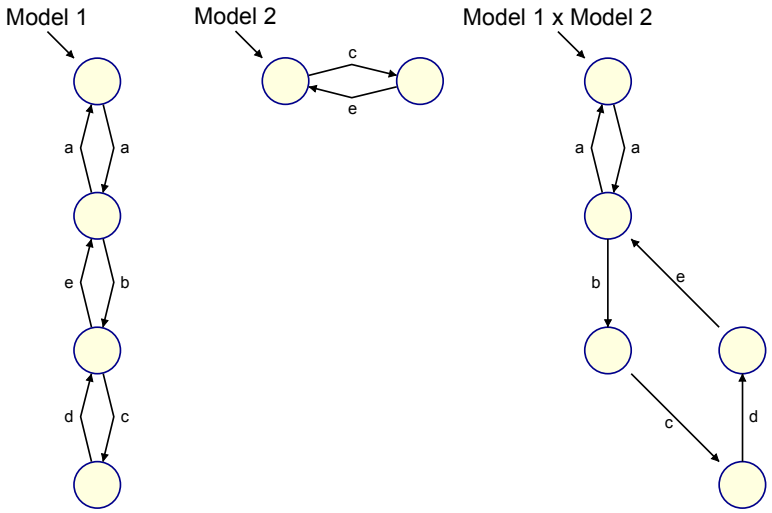

Fig. 1. Two example model components and their composition with the rules $R = \{(a, \surd, a), (b, \surd, b), (c, c, c), (d, \surd, d), (e, e, e)\}$.

those actions. For all rules, this gives $O(|R||M|)$. In addition to this, the handling of each action takes only constant time, yielding $O(\sum_{m \in M} |\Sigma_m|)$. Calculating the strong components of a single model $m \in M$ with Tarjan's algorithm takes $\Theta(|S_m| + |\Delta_m|)$ time. However, since we are only interested in the initial strong component, effectively $|S_m| \leq |\Delta_m| + 1$, resulting in $\Theta(|\Delta_m|)$. The subsequent handling requires $\Theta(|\Delta_m| + |\Sigma_m|) = \Theta(max(|\Delta_m|, |\Sigma_m|))$. The calculation is carried out for each model only after new actions have been banned; since all unreachable actions get banned on the first (compulsory) time, the calculation will be performed at most $min(|\Sigma_m|, |\Delta_m|) + 1$ times. The result is $O(\sum_{m \in M} min(|\Sigma_m|, |\Delta_m|)max(|\Delta_m|, |\Sigma_m|)) = O(\sum_{m \in M} |\Sigma_m||\Delta_m|)$.

Putting the above figures together, we get $O(|R||M| + \sum_{m \in M} |\Sigma_m||\Delta_m|)$. This means linear dependence on the number of rules times the size of a single rule, plus quadratic dependence on what is essentially the sizes of the model components. The first term is quite reasonable, since the same time is required to simply write out the rules. The second term, while not insignificant, is still perfectly manageable if individual model components are kept small enough.

3.5 Example

We will now present an example of Algorithm 1 with the models in Figure 1, combined with the rules $R = \{(a, \surd, a), (b, \surd, b), (c, c, c), (d, \surd, d), (e, e, e)\}$. Let us assume that the implementation of action d of Model 1 is faulty and initially ban $(1, d)$.

Since the action $(1, d)$ is banned, we remove the rule $(d, \surd, d)$ which refers to it. After that, we must calculate strong connectivity; we shall do it for Model 1

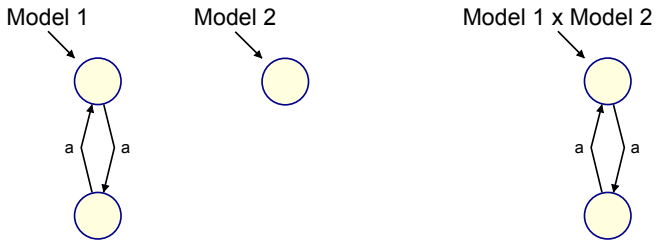

Fig. 2. Filtered versions of the example model components and their composition with the filtered rules $R = \{(a, \surd, a)\}$.

(calculating the strong connectivity for Model 2 would not yield anything new anyway). We notice that in Model 1 the action c leads out of the initial strong component and ban $(1, c)$. Consequently, we also remove (c, c, c) and then, because there are no longer any rules for it, $(2, c)$.

Again we must calculate strong connectivity. This time, we do not learn anything from calculating it for Model 1, but in Model 2 we notice that $(2, e)$ is unreachable and ban it. Following that, we remove (e, e, e) and ban $(1, e)$. We note that now the action b breaks the strong connectivity of Model 1, and ban $(1, b)$ and remove $(b, \surd, b)$. Finally, Model 2 has changed since our last connectivity calculation for it, so we perform one, but learn nothing new. At this point the algorithm returns the results and terminates.

In the end, we have banned the actions b, c, d and e from Model 1; banned the actions c and e from Model 2; and removed the rules $(b, \surd, b)$, (c, c, c), $(d, \surd, d)$ and (e, e, e). All that is left of the model components is a two-a loop in Model 1, which is also exactly what will show up in the test model composed with the single remaining rule $(a, \surd, a)$, as seen in Figure 2. Looking at the original composed model in Figure 1, it is easy to see that this is what should happen with the action d banned.

3.6 Other Composition Methods

If the algorithm is to be used with a different method of parallel composition, it will be necessary to create a rule set that implements corresponding functionality. For example, the basic parallel composition where actions of the same name are always executed synchronously would correspond to the rules

$$R = \{(\sigma_1, \ldots, \sigma_n, \sigma_R) \in (\Sigma_1 \cup \{\surd\}) \times \cdots \times (\Sigma_n \cup \{\surd\}) \times (\Sigma_1 \cup \cdots \cup \Sigma_n) \mid \\ \forall i; 1 \leq i \leq n : (\sigma_R \in \Sigma_i \rightarrow \sigma_i = \sigma_R) \wedge (\sigma_R \notin \Sigma_i \rightarrow \sigma_i = \surd)\}$$

Although the rule set is needed for the execution of the algorithm, it is not necessary to actually implement rule-based parallel composition. The list of banned actions the algorithm returns can be used to perform filtering within the model components, and these can then be combined with the original method of composition.

4 Case Study

As a case study, we will examine the process of modifying models from an existing model library to conform to the requirements of filtering. The purpose is to ensure that test models composed from the library can be relied on to remain strongly connected when arbitrary actions are filtered out; afterward, filtering can be performed automatically. First, we will present the model library and how its model components might in practice be filtered. We will then examine the actual modifications made to the models of one application in the library, and finally analyze the results.

4.1 Setup

The model library we will examine has been designed for the testing of smartphone applications [5]. The latest version contains models for eight applications such as Contacts and Messaging, over four different phone models, on different platforms such as S60 and Android. The model components in the library have been designed to yield a usable test model even if only some of them are included in the composition, as long as specified dependencies are met. However, they have not been designed to withstand the arbitrary removal of actions gracefully.

In this case study we will focus on the models of the Contacts application. It consists of six action machines and a corresponding number of refinement machines, and has about 330 states altogether. As such it is one of the smaller applications in the library, and simple enough to be a comprehensible example.

When examining the effects of filtering, we can safely limit ourselves to banning action words in the action machines, since they represent the (potentially unavailable) functionality of the SUT. The task is performed by banning action words one at a time and examining the results with the help of the filtering algorithm. From the results we can determine whether the composed test model would remain strongly connected or not.

4.2 Modifications

An initial execution of the algorithm with no actions banned yields a list of a few unimplemented actions; these appear in the action machines but have no implementation. Such actions would not appear in the test model anyway, so they can be safely banned. We then proceed to banning individual action words, and find two problematic situations.

The first problem we encounter is in the model component depicting the functionality of the list of contacts (Figure 3). The only action word in the model, *awVerifyContactsExist*, is a branching action word used to find out whether there are any contacts in the application (the negative branch is prefixed with a '∼'). This action can only be executed if we are unsure of the current situation regarding contacts; the preceding synchronization actions check from other model components whether we know anything about the existence of contacts.

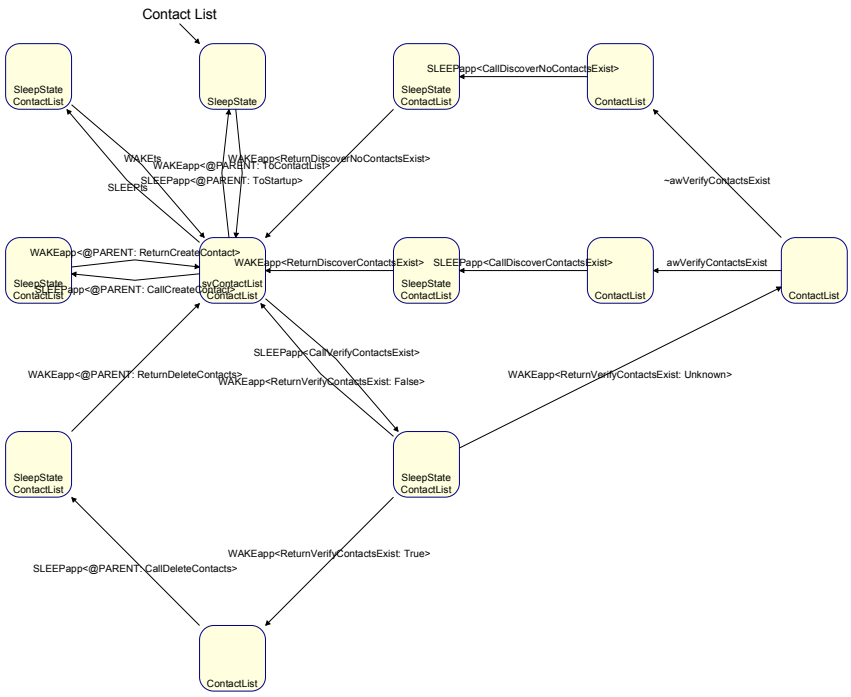

Fig. 3. The Contacts List action machine, with the action word *awVerifyContactsExist* on the right.

The filter algorithm quite intuitively suggests that if the action word is banned, the action *WAKEapp<ReturnVerifyContactsExist: Unknown>* should also be banned to preserve strong connectivity. However, that would actually cause a deadlock elsewhere in situations where the existence of contacts really is unknown. The solution here is to add a transition with a new comment action from the state on the right between the synchronization and the action word back to the central state on the middle left. A comment action can be executed with no effect to the other model components or the SUT, allowing us to bypass the verification of contacts' existence. Now the synchronizing action no longer needs to be removed with the action word, and strong connectivity is preserved.

The second problem spot is also related to the way the models keep track of the number of contacts. The existence of contacts is abstracted into three categories: contacts exist, contacts do not exist, and unknown, with unknown used as the initial value. The problem shows up in the model component responsible for the deletion of contacts (Figure 4), if we ban one of the actions *awToggleContact*, *awAttemptDelete* or *awDelete*.

The immediate result of the ban is that contacts can no longer be removed individually (or at all for *awDelete*). However, the individual removal of contacts

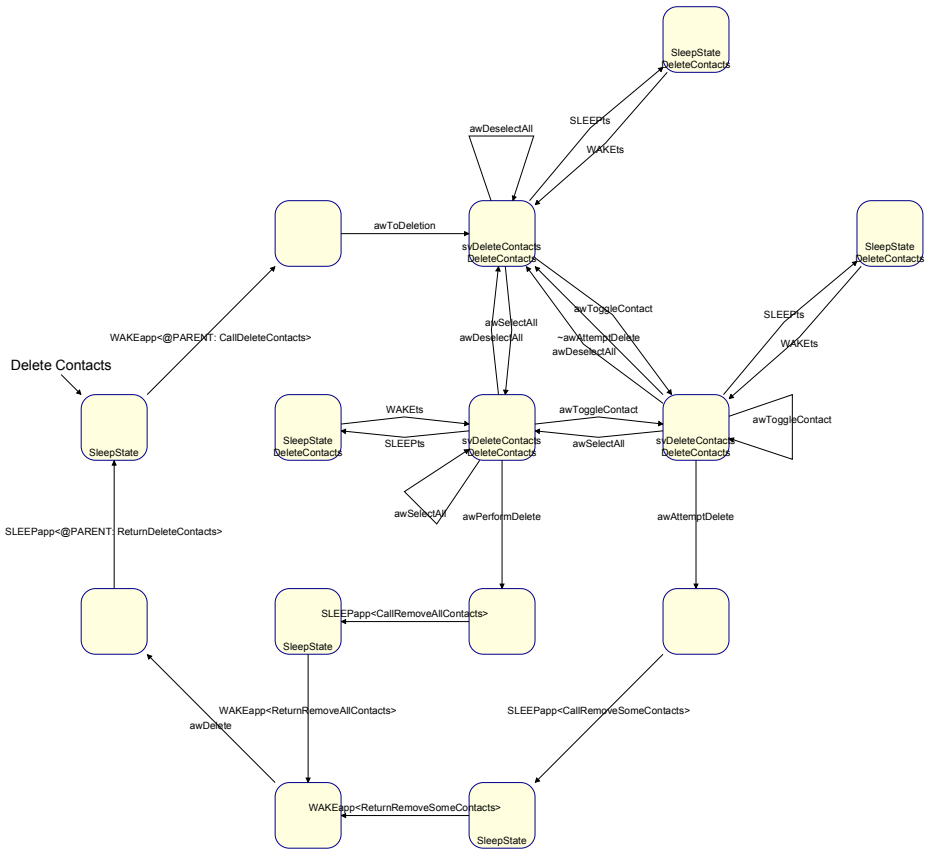

Fig. 4. The Delete Contacts action machine, with the action words *awToggleContact* and *awAttemptDelete* at the right side of the octagon, and *awDelete* at the bottom left. *awAttemptDelete* fails if no contacts are selected.

is the only way that the existence of contacts, once known, can become unknown again. This means that their existence cannot ever be allowed to become known, which results in banning every action related to their creation and handling. The test model becomes next to useless, though is does remain strongly connected. Despite the apparent complexity of the problem, the solution is simple: modify the models so that the knowledge of the existence of contacts can be 'forgotten', moving us back into the unknown state.

4.3 Results

All in all, the Contact models withstood the banning of action words fairly well. The first described problem is likely typical, with complex synchronizations between the model components resulting in a deadlock whose existence the fil-

tering algorithm cannot deduce. The second problem shows that broken strong connectivity is not the only potential issue; one should also consider whether connectivity could be preserved with lesser limitations.

The filtering algorithm was very useful in finding the problematic situations in the models. While the first problem would have been easy enough to spot in manual inspection, the second was more obscure and might have been easily missed. Using the algorithm to calculate the effects of removing actions was also much faster than manual examination would have been.

Making the necessary modifications to the models clearly requires some modeling expertise. This is not a serious issue, since they would usually be made by the original modeler, as part of the normal modeling process. In this case the whole modification process took less than an hour, and was performed manually apart from using the filtering algorithm. Thus, there should not be any significant increase in the modeling effort.

5 Discussion

Using model-based testing in the early phases of product implementation can be difficult, because the product does not yet correspond to the model depicting the entire system. The problem can be solved by altering the model so that unimplemented or faulty functionality is removed and no tests are generated for it. This way the model can be matched to the product throughout its implementation.

Model transformations [2] can be used to modify the test models as needed; their use to keep the test models up to date during development is described in [12]. The use of parallel composition to limit the model to specific scenarios is mentioned in [3, 14], although no mention is made of ensuring the viability of the resulting models. All in all, there does not appear to be much previous work on restricting the functionality of test models and the consequences thereof.

The basic method presented in our paper is very simple, based on banning the actions corresponding to unexecutable functionality in the models or removing the rules acting on them in the parallel composition. The greatest challenge is ensuring that the model remains conducive to test generation; specifically that it remains strongly connected. The algorithm presented in the paper seeks to estimate the initial strong component of the model as well as possible without actually calculating the composed test model. The rest is left up to the modeler.

Our case study showed that modifying existing models to withstand filtering without losing strong connectivity is feasible; by extension, so is designing models to match the same requirement from the first. The filtering algorithm proved very useful in the task, since it can be used to show the effects of banning specific actions and thus reveal problematic structures in the models.

The filtering algorithm takes advantage of the explicit set of synchronization rules used by our method of parallel composition. It can also be used with other parallel composition methods, if a suitable rule set is created to describe the synchronizations. The practical issues related to this are left for future work.

Likewise for the future are left the methods for filtering non-behavioral models and test data.

Acknowledgements The author wishes to thank Mika Katara, Shahar Maoz and Heikki Virtanen for their comments. Funding from Tekes, Nokia, Ixonos, Symbio, Cybercom Plenware, F-Secure, Qentinel, Prove Expertise, as well as the Academy of Finland (grant number 121012), is gratefully acknowledged.

References

1. Buwalda, H.: Action figures. STQE Magazine, March/April 2003
2. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. pp. 324–339. Springer-Verlag (2003)
3. Ernits, J., Roo, R., Jacky, J., Veanes, M.: Model-based testing of web applications using NModel. In: TestCom/FATES. pp. 211–216. Springer-Verlag (2009)
4. Fewster, M., Graham, D.: Software Test Automation: Effective use of test execution tools. Addison–Wesley (1999)
5. Jääskeläinen, A., Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Takala, T., Virtanen, H.: Automatic GUI test generation for smart phone applications - an evaluation. In: Proc. of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009). pp. 112–122, companion volume. IEEE Computer Society (2009)
6. Jääskeläinen, A., Kervinen, A., Katara, M.: Creating a test model library for GUI testing of smartphone applications. In: Proc. 8th International Conference on Quality Software (QSIC 2008) (short paper). pp. 276–282. IEEE Computer Society (Aug 2008)
7. Karsisto, K.: A new parallel composition operator for verification tools. Doctoral dissertation, Tampere University of Technology (number 420 in publications) (2003)
8. Katara, M., Kervinen, A.: Making model-based testing more agile: a use case driven approach. In: Proc. Haifa Verification Conference 2006. pp. 219–234. No. 4383 in LNCS, Springer (2007)
9. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Satama, M.: Towards deploying model-based testing with a domain-specific modeling approach. In: Proc. TAIC PART – Testing: Academic & Industrial Conference 2006. pp. 81–89. IEEE CS (Aug 2006)
10. Practise research group: TEMA project home page. Available at `http://practise.cs.tut.fi/project.php?project=tema`. Cited Apr. 2010.
11. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1998)
12. Rumpe, B.: Model-based testing of object-oriented systems. In: Formal Methods for Components and Objects, International Symposium, FMCO 2002, Leiden. LNCS 2852. pp. 380–402. Springer-Verlag (2003)
13. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
14. Veanes, M., Schulte, W.: Protocol modeling with model program composition. In: Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems. pp. 324–339. Springer-Verlag (2008)

[P5] ©2010 IEEE. Reprinted, with permission, from Proceedings of the 3rd IEEE International Conference on Software Testing, Verification, and Validation (ICST 2010), “Debug Support for Model-Based GUI Testing”, H. Heiskanen, A. Jääskeläinen, and M. Katara

Debug Support for Model-Based GUI Testing

Henri Heiskanen, Antti Jääskeläinen, and Mika Katara
Department of Software Systems, Tampere University of Technology, Finland
{henri.heiskanen, antti.m.jaaskelainen, mika.katara}@tut.fi

Abstract

The fact that model-based testing has not yet attained a high rate of adoption in industry can in part be attributed to the perceived difficulty of debugging long error traces often produced by the online version of this technology. Given the extensive manual labor commonly involved in the debugging phase, automating parts of this process could yield considerable productivity benefits. This paper presents viable debugging strategies applicable in model-based graphical user interface testing, from which two methods were refined and experimented with. The first is based on superimposing log-derived, synchronized subtitles on recorded test run footage, while the second addresses error trace shortening. The results obtained from applying these methods in real-life case studies demonstrate the practical utility of these methods.

1. Introduction

It is commonly recognized that testing and debugging consume a considerable amount of time and resources in software projects due to the great manual work needed in these tasks. Especially black-box testing and the fault analysis and localization related to this type of testing are highly context-sensitive activities, which accounts for the limited number of approaches working across different contexts [1]. However, if we could develop techniques to automate parts of the laborious fault analysis and localization process at least in some widely used context, such as graphical user interface (GUI) testing, this could potentially have significant benefits in the productivity of software production at least in that specific context.

Model-based testing (MBT) [2] offers many advantages relative to traditional script-based approaches. However, MBT is not widely spread and has as yet been adopted primarily by technological innovators. This low adoption rate of MBT is due to both technological and non-technological reasons [3], [4]. As a technology, MBT has been under active research in recent years, during which the paradigm has spawned numerous practical applications, e.g. [5], [6], [7]. It is obvious that the success of these technologies depends to a great extent on the ability to correctly analyze and localize discovered faults, i.e., the debugging of model-based test runs is vitally important and factors greatly into the degree to which these technologies provide value to their users. The work described in this paper addresses the difficulties of fault analysis and localization that are inevitable when engaging in automated test generation based on behavioral models.

Since MBT allows long-period testing where new sequences of events are constantly input to the system under test (SUT) instead of just repeating the same test cases as in conventional test automation approaches, the *execution traces*, i.e. sequences of executed actions, produced by this technology can be very long. When an execution trace has given rise to a failure, it is known as an *error trace*; the failure can be due to the SUT functioning incorrectly or some other reason. Such a trace can often be automatically obtained from a test log. The purpose of debugging, in the instance of MBT, is to establish the cause of failure by examining the error trace or by some other means. The significance of this process is emphasized by the fact that, depending on the interface in use for test execution, it might be possible to execute a sequence of millions of events in a short period of time: in a few hours, for instance. In the instance of GUI testing, the focus of this paper, test execution is usually slower than when using software application programming interfaces (APIs) or test-specific interfaces; we consider long sequences to consist of thousands of events, which can mean a tedious debugging task without any tool support.

Debugging has recently received growing interest from the research community, with techniques such as dynamic program slicing [8], execution backtracking and delta debugging [9] having been researched to a moderate degree. However, one area that has not been extensively researched so far is the debugging in the context of model-based test automation technologies.

In this paper, we explore solutions for debugging long error traces produced by model-based GUI testing. Some of these approaches have already been applied in different contexts, but in addition to these methods, two promising approaches were conceived and implemented during our research, and case studies were conducted to assess their practical utility. The results obtained from these case studies establish the usefulness of these methods in the target context.

The remainder or this paper is structured as follows: In Section 2 an overall presentation of MBT is provided. In Section 3 we introduce methods for debugging long error

traces. The results of the case studies are presented in Section 4 and conclusions drawn in Section 5.

2. Model-Based Testing

In general, MBT can be defined as a testing approach where not only the execution, but also the generation of tests is automated. The level and depth of test generation may vary between different techniques. The tests are generated, at whatever level, from a formal *test model*, such as a state machine, the contents and purpose of which also vary between different approaches. The test model may be based on system requirements or specifications, or reverse-engineered from the SUT.

There are two basic methods for using the test model to create tests. In one approach, *offline testing*, the model alone is used to create a finite sequence of actions according to some predefined criteria. This sequence is then treated as a traditional test script, and may be executed at some later point. The other approach is *online testing*, where tests are created and executed simultaneously. Every time an action is selected in the model, it is also executed in the SUT. Separate scripts are not created at all. Online testing has two advantages over offline testing. First, since selected actions are executed immediately, the results of the execution can affect the progress within the model and thereby the selection of the following actions. This allows the effective modeling and testing of nondeterministic systems. Second, an online test may be unlimited in length since there is no need to prepare a complete script beforehand.

The selection of actions in the model is performed by a *guidance algorithm*, whose task is to select the actions to be added to a script or executed, depending on the approach. Useful tests can be generated even with a totally random algorithm, but more complicated heuristics and parameters allow the generation of different kinds of tests, based on use cases, for instance. The parameters defining the objective of the test are called the *coverage requirement*. For example, a test corresponding to a use case might be generated by using a graph search heuristic with a coverage requirement representing the use case. The semantics we use for such a coverage requirement is described in [10]; essentially the requirement is an expression of model actions combined with operators AND, OR and THEN. For example, the expression A THEN (B OR C) would require the execution of action A, followed by the execution of either B or C. Other actions could be executed in between; coverage requirements do not restrict executable actions. A graph search algorithm would seek to fulfill the requirement by first searching the model for a path to a transition labeled with A. Once a path is found, it is executed in the model and on the SUT. Next, a path leading to either of B or C is searched for and executed. With strongly connected models a path to any transition can always be found.

In comparison with traditional script-based testing, MBT has many advantages. First, provided that the test model is perfected so that it contains all the necessary functionality of the SUT, it is possible to automatically generate test cases far more inventive and intricate than a human tester could design. In other words, automatically generated test cases can describe scenarios that never would have occurred to a human tester. Second, the modeling process itself can uncover many faults in the test target, possibly even more than what could be discovered as a result of running the actual tests based on the created test models. Last, maintainability is facilitated as there is no need to update test cases when the SUT changes, the changes needed can usually be limited to few component models.

Maintainability can be further enhanced by separating the models into two levels of abstraction, based on *action words* and *keywords* [11]. Action words depict the functionality of the SUT, that is, what actions the SUT can perform. Keywords, on the other hand, depict user interface (UI) events, for example, key presses and onscreen text verifications. Models based on action words describe the functionality of the system as a whole and define what kinds of tests can be generated. Models based on keywords contain the implementations for the action words. Since the functionality of the SUT is relatively stable during development, as compared to the often more volatile UI, the necessary maintenance effort can be focused on the keyword level. For a more thorough description of our approach and the associated open source toolset called TEMA, the reader is referred to [7].

Automatic execution of model-based tests is enabled by an *adapter*, a component which transforms the actions of the test sequence into actual events on the SUT. The adapter determines the set of keywords available in modeling; the set is based on what access the adapter has to the SUT.

The whole MBT and debugging process is illustrated in Figure 1. Models may be based on the requirements or specifications of the system, or reverse-engineered from a working SUT. The test model is used to generate tests, which are relayed to the SUT via the adapter for execution. When the test results (including the test log) indicate errors in the system, they are debugged in order to establish their source, so that they can be fixed.

3. Debugging Long Error Traces

Even though MBT features many advantages, it poses new challenges to the process of debugging run tests, especially when these tests have been long. This section treats the debugging issues stemming from the complexity of the MBT paradigm, focusing particularly on the aggravated problems with longer test runs. At the outset, some general considerations and possibilities in debugging MBT are discussed, continued by the introduction to two prospective debugging

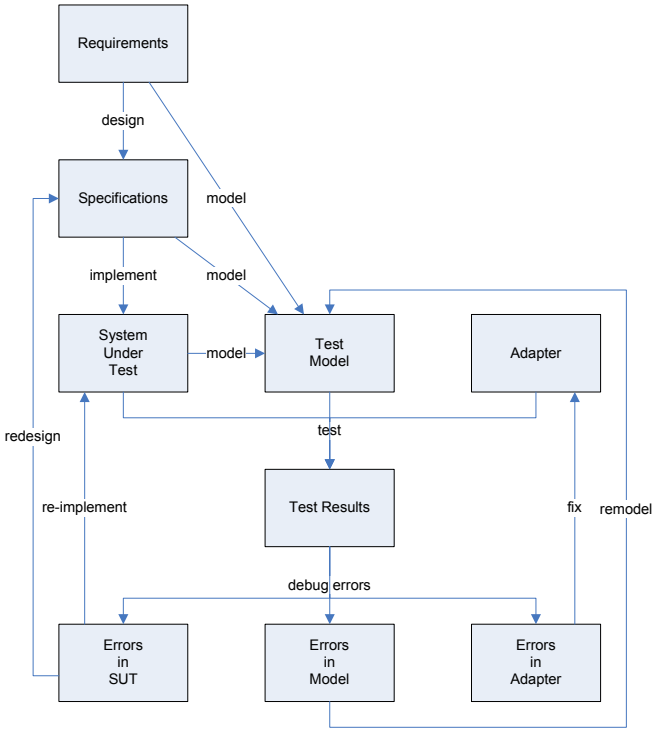

Figure 1. Testing and debugging process.

methods and a brief look at a few alternate, potential debugging methods. The implementations of the identified two methods are thereafter presented in more detail.

The first of the two implemented methods is particularly applicable in GUI testing, exploiting both video footage of a test run and the test log of that particular test run, whereas the second method draws on a simple principle of executing the original test gradually in subsequences of the original trace. This process is carried out with an ascending number of actions included in the subsequences, starting from the very last action of the original test, while the actual execution of events is still carried out with a guidance algorithm.

3.1. Debugging Model-Based Test Runs

In general, it can be stated that the most common reason for a MBT test failure is a conflict between the SUT behavior and the modeled behavior, i.e., the test model does not, to some degree, describe the behavior of the SUT as it should. This may be due to incorrect modeling or a real fault in the SUT.

In our approach, another very common cause of failure in performed test runs has been the delay between executing certain keywords on the SUT, which could be pinned on both the SUT and the test model. For example, when sending two consecutive key presses within a short enough interval to the SUT, the second one cannot be processed by the SUT while the execution of the first one is still in progress, resulting in a test run disruption. This is, on the one hand, a property of any computing device, but on the other hand, an oversight in the model design process, as it is possible to reckon with such eventualities by adjusting a proper delay in the models. This delay would then occur always after the execution of certain keywords, thus insuring the SUT recovery.

As far as debugging is concerned, the issues originating in the test model are probably those most susceptible to systematic debugging procedures. Conversely, faults originating from somewhere else are more difficult to detect by any systematic method, as the nature of these faults could vary substantially.

The two aforementioned model-related fault areas, timing issues and conflicts between the test model and the SUT, are easiest to notice when it is possible to simultaneously view the actual test run and the events recorded in the test log of that particular test run. This would make it possible to instantly notice when the events occurring in the test run no longer check with those suggested by the test log. This is a viable option especially in GUI testing, as tests based on this paradigm can easily be recorded and viewed anytime later.

Then again, some real faults in the SUT could require long action sequences in order for them to manifest themselves, which necessitates a different debugging approach that would address error trace shortening.

3.2. Prospective Methods

Two methods were conceived as tentative options in debugging test runs based on the concepts and methods of our MBT approach, as described in Section 2. These methods could be adapted to other contexts and approaches as well, and their applicability to external contexts will be contemplated as the methods are introduced in more detail. Even though both of these methods are based on simple, common-sense debugging principles, their application in the context of MBT has not been reported before.

3.2.1. Test Run Video Synchronization with Log Data.

The first and more pragmatic of these methods taps into existing video footage of a test run and the test log of that particular test in order to create a synchronized visual trace of the test run. In practice, this is achieved by gleaning the most relevant information from the test log and presenting it as a sequence of timed subtitles, superimposed on the video footage. This would afford remarkable ease for debugging as the test run could be viewed as many times as necessary at its critical points with synchronized event data being displayed simultaneously. Accordingly, this method would facilitate the debugging process by enabling the viewer to instantly distinguish when the actual events performed on the SUT no longer agree with those suggested by the test log data.

This would be especially helpful with long test runs, as the video could be either rewound or fast-forwarded toward the point where the video events no longer square with those implied by the log data, with no need to view the entire video. Furthermore, this method is relatively universal in nature, enabling fault detection in the SUT as well as all parts of the test tool architecture.

As to the application of this method, it would not fit any other testing context except GUI testing on account of its visual nature, and, for instance, debugging API tests with this method would be nonsensical. It should, however, be applicable to other MBT approaches as well, in addition to the one pursued by us, regardless of whether it is online or offline testing or based on action words and keywords, as long as there is some event data with timestamps available.

3.2.2. Trace Incrementation. The second identified method is based on the concept of gradually executing a failed test run in subsequences. More precisely, the actions of an error trace would be compiled into a new coverage requirement starting from the very last action of the error trace while gradually increasing the number of included actions until the whole trace has been included. The method requires a suitable guidance algorithm for executing the resulting coverage requirement. It places no restrictions on how the original trace was formed, however.

In the increase phase of the method the number of included actions would be either multiplied by some coefficient, or alternately increased by adding a constant number of new actions to the total. The action subsequences are formed in reverse order to their appearance in the error trace (from end to beginning), while the contents of the subsequences would retain the same order as during the original test execution. The purpose of this process is to discover as short a subsequence of the error trace as possible that still causes the same failure as the original test run did. When executing the new coverage requirement the shortest subsequences, which only contain actions from the end of the trace, will be executed first. If they fail to reproduce the error, the execution will proceed into larger and larger subsequences, which also contain actions from the beginning of the trace. Since the final subsequence is the original error trace, the failure will eventually be reproduced.

The following are the definitions for the additive and multiplicative versions of *trace incrementation*:

Def. 1 (Additive trace incrementation Inc_+).
$Inc_+(A,k)$ *is the* additive trace incrementation *of the trace* $A = a_1a_2\ldots a_n$ *with the* increment $k \in \mathbf{N}$, $k \geq 1$ *such that* $Inc_+(A,k) = (\text{THEN}_{i=1}^{\lceil n/k \rceil - 1}(a_{n-ik+1}$ THEN a_{n-ik+2} THEN $\ldots$ THEN $a_n)) + A$

Def. 2 (Multiplicative trace incrementation $Inc_\times$).
$Inc_\times(A,c)$ *is the* multiplicative trace incrementation *of the trace* $A = a_1a_2\ldots a_n$ *with the* coefficient $c \in \mathbf{R}$, $c > 1$ *such that* $Inc_\times(A,c) = (\text{THEN}_{i=0}^{\lceil \log_c(n) \rceil - 1}(a_{n-\lfloor c^i \rfloor+1}$ THEN $a_{n-\lfloor c^i \rfloor+2}$ THEN $\ldots$ THEN $a_n)) + A$

This method is especially useful when the exact cause of failure is unknown and the functionality that induces that specific failure cannot be deduced from thorough trace scrutiny. This method also lends itself to debugging long test runs.

An additional advantage of this method is that it would retain its applicability when transitioning to testing of another type. In other words, it would be possible to debug API tests with this method and MBT testing of any other kind as well, provided that there were, again, some existing event data on the test run.

3.3. Related Work

The underlying principle of the trace incrementation method has already been applied to debugging before, for example in [12], where randomized unit test cases are minimized by exploiting the dependencies between the statements involved in the failure, and in [9], which presents a general minimization algorithm to aid debugging, known as *delta debugging*. However, these two approaches are not, as such, applicable in our context. The approach described in [12] is applicable with program code, but in our case the dependencies between the abstract actions involved in a failure cannot be readily established. The delta debugging approach presented in [9] would not meet our needs either since it involves executing a great number of succeeding and failing subtraces in an effort to find a minimal error trace. In our case this would take an excessive amount of manual effort, because our SUTs cannot automatically recover from failure. Trace incrementation is the closest working equivalent: starting from the smallest subtrace, and increasing its size along the route most likely to reproduce the error until a failure manifests itself.

In addition, there are a few other known debugging methods that could be adapted to the needs of MBT and that are viable options in the debugging of model-based test runs. Of these known methods, there are two approaches that are especially applicable to MBT. These methods are next introduced briefly, after which they will be evaluated against the two methods presented earlier.

3.3.1. Alternate Methods. The first of the alternate methods could be used to detect conflicts between the SUT and the test model without having to repeat the entire error trace. In practice, this would be achieved by setting the SUT into the state where the test execution was prior to the execution of the last action of a test, followed by the execution of that particular action. Since executable actions equate to test model transitions, the last action would be performed by traversing the last transition in the test model

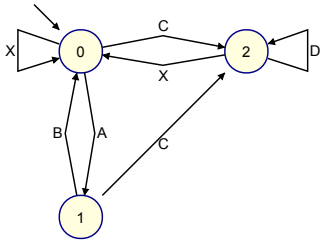

Figure 2. Simple example model.

and executing the corresponding action on the SUT. Now, if the last action could not be executed on the SUT, it would suggest a fault either in the model or in the SUT. The underlying principle of this *transition-specific search* method has been applied in debugging before, for example in [13], where the search of failure-inducing state changes is automated. The paramount advantage of this method consists in discovering circumstance-dependent failures that might otherwise be difficult to uncover. For example, some failure might manifest itself only when multiple applications are running at a time or the state of the SUT is otherwise propitious for the failure to occur.

The second prospective debugging method is also closely related to models, as it is about removing loops from the traces executed on them. Supposing that the test execution deviates from the path necessary for some failure to occur, executing some other extraneous actions before reverting to the original path, this unnecessary loop of actions would only lengthen the necessary sequence of actions for the failure to occur. If these redundant loops were removed from the trace of executed actions, the sequence of necessary actions to execute in order to reproduce the failure could be considerably shortened. Accordingly, with this *loop removal method* it is possible to shorten the error trace when it contains extraneous action loops. The principle of this method has already been successfully applied in other debugging areas, for instance in [14], and it would suit the special needs of MBT as well.

3.4. Examples and Comparison

In order to compare the transition-specific search, trace loop removal and trace incrementation methods, we will use the example model depicted in Figure 2. As an example error trace we will use $ABACDX$, where the final action X has failed. We will examine four different potential causes which might induce the failure:

1) Action X is faulty and always fails.
2) Action X fails when executed in state 2.
3) Sequence DX fails, but actions D and X work individually.
4) Action X fails if action A has been executed at some earlier point.

In all cases where relevant, we assume a perfect guidance algorithm capable of finding the optimal path fulfilling the given coverage requirement.

With the given error trace, transition-specific search will seek to bring the model into state 2, where the failed action was executed, and then execute X. This results in the simple trace CX. This is the optimal solution in the second examined case of error causes, that is, it reproduces the error with the shortest trace possible. It also reproduces the error in the first case, though with a non-optimal solution. It fails to reproduce the error in cases three and four.

Trace loop removal will directly result in an executable trace ACX. This is an optimal solution for reproducing the error in case four, along with ABX. It also reproduces the error in cases one and two, but not optimally. Just as with transition-specific search, it will not reproduce the error in case three.

Trace incrementation, used with an additive increment of 3, will produce the coverage requirement

$$(C \text{ THEN } D \text{ THEN } X) \text{ THEN}$$
$$(A \text{ THEN } B \text{ THEN } A \text{ THEN } C \text{ THEN } D \text{ THEN } X)$$

The guidance algorithm would fulfill it with the trace $CDXABACDX$, which, incidentally, contains exactly the same actions. The actual execution will naturally stop as soon as the error is reproduced. With a multiplicative coefficient of 2 we would get the coverage requirement

$$(X) \text{ THEN}$$
$$(D \text{ THEN } X) \text{ THEN}$$
$$(A \text{ THEN } C \text{ THEN } D \text{ THEN } X) \text{ THEN}$$
$$(A \text{ THEN } B \text{ THEN } A \text{ THEN } C \text{ THEN } D \text{ THEN } X)$$

and the trace $XCDXACDXABACDX$. These traces reproduce the error in all four cases. The additive version gives the optimal solution in the third case and the multiplicative in the first case. Notably, trace incrementation is the only one of the examined methods which can reproduce the error in case three.

In general, transition-specific search and trace loop removal may fail to reproduce the error in some cases, but are reasonably efficient. Both produce a path, which is necessarily bounded by the size of the model. In transition-specific search the path is (with ideal guidance) the shortest possible from the initial state to the target transition, whereas trace loop removal may produce any path up to the longest possible in the model. However, the latter should reproduce any error that the former would and more.

In contrast, trace incrementation will always reproduce the error with any error trace, but its efficiency can vary wildly. In the worst case, the very first action in the error trace is necessary to reproduce the error, which may lead to a situation where the error is reproduced only on the last iteration of the incrementation.

Execution time for such a trace will be long compared to the original error trace; longer for additive incrementation than for multiplicative. Furthermore, the error trace may already be very long compared to the size of the model since trace incrementation is most useful in debugging long and unwieldy traces. However, trace incrementation does have the unique advantage of providing a lower bound to the sequence necessary to reproduce the error, that is, it shows that the increment before the last is not sufficient to reproduce the error.

3.5. Implementation

The two identified methods, based on video synchronization and trace incrementation, were implemented in order to assess their practical utility in real-life error scenarios. These two debugging tools will next be presented and analyzed at greater length.

3.5.1. Video Synchronization Method. The first developed debugging method consolidates existing video footage on a test run and the test log of that specific test run into a synchronized whole. This method has proved very efficient in debugging test runs performed during the research.

The exact process of using this debugging method consists of three different phases. First, the test run to be debugged must be recorded. Second, the subtitle file must be generated by collecting the most interesting events from the test log of that particular test. In the instance of our approach, the most interesting events in a test log for debugging purposes are sending keywords to the adapter, executing keywords on the SUT and commencing the execution of a new action word, which is accomplished by executing a certain number of keywords, depending on the state the SUT is in when the execution of the action word commences. These events impart much value in the way of debugging the test run, as it is easy to see when a keyword is sent to the adapter and whether that keyword will be executed on the SUT. Furthermore, the progress of the test can easily be tracked as action words are displayed on the screen at the beginning of their execution. When this information is displayed in close synchronization with the video footage, it is easy to notice any discrepancies between the recorded log actions and the real actions executed on the SUT.

Apart from the aforementioned events gathered from the test log, their timestamps are also included in the subtitle file to indicate the exact moment when the event associated with the timestamp occurred during the actual test run. This is especially useful when searching for a particular event on the video, as the timestamps act as subtitle identifiers. For the process of creating the subtitle file we have developed a program named Log2Srt. This program creates a *SubRip* file for the subtitles, a format widely supported by software media players, by collecting the desired events and their timestamps from the test log.

Finally, with an existing video file and the subtitle file generated expressly for it, the only phase left in the debugging process is to play the video file on a player capable of displaying SubRip-formatted subtitles, for example MPlayer [15], which is a well-advised choice for debugging ends. This is mostly due to some features of this player that are instrumental in debugging, such as the possibility to increase or decrease the playback speed, particularly useful when the rate of successive events is so rapid that it is difficult to monitor the test progress. The importance of this is emphasized in situations where the decisive malfunction occurs at a moment when there are many events happening within a brief period of time, rendering it very difficult to discern what the cause of the ensuing test disruption is.

This approach is illustrated in Figure 3, which presents a point in one of the test runs conducted during the research where a smartphone is acting as the SUT, as viewed on MPlayer. At this particular point, the test run commences the execution of a new action word that is about writing text in a multimedia slide, as displayed in the upper subtitle. In this case, the action word is translated into a sequence of keywords that begins with a typing keyword with the desired text as its parameter, as presented in the lower subtitle. The question mark following the keyword denotes sending it to the SUT, whereas the lack of it would indicate the completed execution of the keyword. As mentioned before, the keyword subtitles are also equipped with timestamps originally generated when the data was written into the test log. The inclusion of timestamps might be helpful when attempting to locate some specific test log data on the video. In other words, the timestamps act as subtitle identifiers and linkers between the video subtitles and the original test log.

At present, the entire process of creating a synchronized, subtitled video of a test run is automated except for one part: the start of video capturing is not synchronized with the test adapter. In other words, when the adapter starts to execute keywords on the SUT, the video recording must be started manually at around the same time. Later on, when creating the subtitle file, it must be manually determined as to whether there is a delay between the subtitles and the events on the video and how long the delay is. Usually, a rough estimate in seconds is adequate for achieving a close enough synchronization. This part could, though, be automated as well, if the adapter had built-in video recording functionality or were otherwise able to control the video recording process.

3.5.2. Trace Incrementation Method. The second implemented method for debugging model-based GUI test runs is named trace incrementation method for the principle of its operation described earlier. In our experiments, we opted for multiplicative trace incrementation with the coefficient of 2.

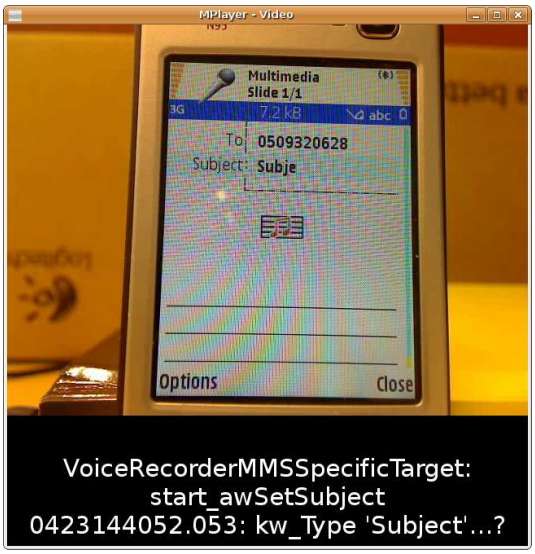

Figure 3. Video synchronization debugging approach

The actual subsequence creation process is performed by extracting a given number of actions from the end of the test log, retaining the original order of the extracted actions. When all subsequences have been derived, they will be parenthesized and concatenated by THEN operators so that all potential failure-inducing subsequences can be tested at once, with no need to repeatedly perform the same process for every single subsequence. For example, with the multiplicative approach and coefficient of 2, an error trace of eight actions would be processed into a subsequence concatenation by forming subsequences from the last action, the last two actions, the last four actions and, finally, all the eight actions of the original trace, in that specific order, followed by the concatenation of the parenthesized subsequences with THEN operators.

In practice, this whole process is carried out on a program named Sequencer, which was developed during the research. This program forms the action subsequences by extracting actions from a given test log and concatenates them in the manner described before. Once the complete concatenation of subsequences has been produced, it will be executed in a fashion similar to the execution of ordinary coverage requirements. Once the failure has been reproduced, it can be seen from the output which subsequence of the concatenation was responsible for that specific failure.

On the whole, this method serves its purpose well in condensing the failure-inducing trace, rendering it a viable option in debugging long test runs.

4. Case Studies

The two implemented methods were tested and applied to failed test runs. These experiments yielded positive results, thereby establishing the value of these methods. Through a careful study of the results of these experiments, faults were detected in test models, the adaptation component and the SUT itself. The most common cause of test failure was a conflict between the test model and the SUT, i.e., the models in use were outdated or otherwise incompatible with the SUT behavior. Nevertheless, many faults were detected in the SUT, some of which were more serious in nature, while other findings were only minor issues and inconsistencies, scarcely classifiable as faults [16].

Of all the detected faults, two interesting cases will be presented in more detail in this section as concrete case studies conducted with the implemented methods and our TEMA toolset, presented in greater detail in [7]. The first case, caused by a modeling issue, will be presented at the outset, followed by the second case, which is related to a real fault in the SUT. In conclusion, the results of these experiments will be summarized and further discussed.

4.1. Case Study I: Video Synchronization

The first case study discusses a test run that was conducted during the research in an attempt to gain experience with long test runs. In the instance of this particular test run to be presented in detail in this case study, it was decided to use the video synchronization method to help determine the real cause of failure.

The test run was conducted with a smartphone as the SUT, from whose functionality the most important multimedia properties were included in the test model that was used in the test run. In other words, the test run was intended to cover functions such as creating and sending multimedia messages, playing audio and video files, recording sounds and creating presentations. The smartphone was then connected to the adapter, which in turn executed actions on the smartphone. These actions were randomly selected by a random guidance algorithm since the test run was intended to be infinite in duration.

Eventually, the test run ended after three hours and forty-five minutes. At the beginning of this timespan there were no anomalies and the test run proceeded smoothly. Then, at around 3:45, the course of the test run was deflected, followed by an abrupt termination of the test. At the beginning of the series of events that ultimately led to the deflection, the SUT was playing an audio file on its media player, when it suddenly received a center push keyword, which corresponds to pressing the center button commonly found in many smartphones. The controls of the SUT media player are mapped so that pressing the center button either pauses or continues to play the audio file that the player was playing at the time. When the SUT received the center push keyword, it was playing the audio file and as a result of the received keyword it paused the audio file.

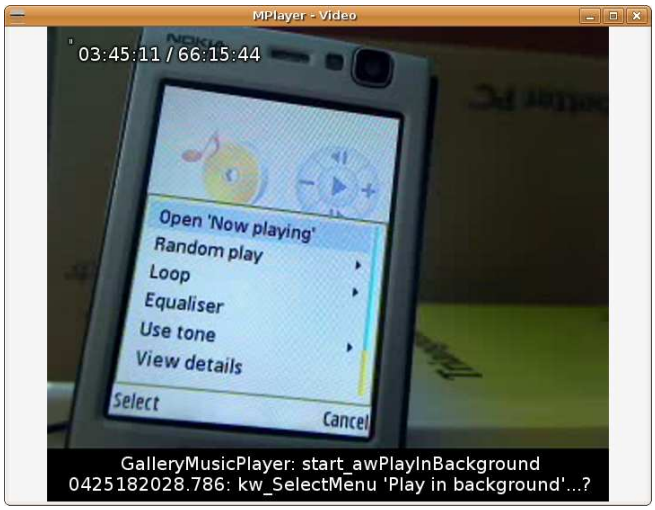

Figure 4. The SUT attempts to play the audio file in the background

Now, around one second after receiving the center push keyword, the SUT was sent another center push keyword. However, this keyword was not registered by the SUT due to its proximity to the first received center push keyword, i.e., the SUT had not recovered from the execution of the first center push by the time it was sent the second. These contiguous keywords disrupted the test run, as a result of which the test execution was eventually terminated. The disruption itself was manifested by the subsequent events of the test run, which had now paused the audio file, while it should have been playing that file had the SUT executed the second center push.

Shortly after the second center push, the SUT attempted to play the audio file in the background, which would ordinarily appear as an option in the media player menu. This is illustrated in Figure 4. This menu is accessible by pressing the key mapped to the Options text, found in the lower left corner of the screen. This time, however, the media player had paused the audio file and the contents of the menu were different from what they would have been had the file been playing. Now, there was no option of playing the audio file in the background since there was no audio file playing at the time; the SUT attempted in vain to locate an option that was not found in the menu, disrupting the test run.

This is one of those cases where the video synchronization debugging approach was indispensable, as without it, distinguishing the ultimate cause of failure would have required time-consuming log scrutiny, after which it might still have been unclear as to what actually induced the failure. In that case, the only means of further investigation would have been manually repeating the same actions on the SUT as were performed by the test run. This, however, might not reveal the real cause of failure if the person carrying out this process did not suspect a timing issue.

Fortunately, the video synchronization method dispensed with these time-consuming manual processes, and with this method it could be immediately distinguished when the video and the test log data became desynchronized. Once the exact onset of desynchronization had been localized, it was relatively easy to notice the underlying timing issue after a closer study of the events preceding that specific point. This was further aided by the capability of MPlayer to decrease the playback speed, which enables viewing of the most critical moments at a suitable pace and is thereby highly conducive to fault detection. Overall, debugging this test run was relatively easy with the help of the video synchronization method, whereas it might have been substantially more difficult if conducted manually, without any assisting tools.

4.2. Case Study II: Trace Incrementation

The second case study[1] considers a more challenging and intricate debugging scenario relative to the one presented in the previous section. There were attempts to debug this particular test failure with both the video synchronization debugging method and manual procedures. However, with neither of these approaches was it possible to determine the real cause of failure, and hence it was decided to shorten the error trace instead of determining the cause behind it, as it could not be accomplished in this case. For this purpose, the trace incrementation method was employed.

As for the test run itself, it was conducted with a similar test setting as in the first case study, with the random guidance algorithm randomly selecting actions to execute and a smartphone acting as the SUT. This time, however, the included functionality to test was limited to a mere calendar application. The purpose of this was to attain as long tests in duration as possible, without any failures whatsoever. This is easiest when there are the smallest possible number of interrelating components involved in the test run. With just the calendar application, which is arguably one of the simplest applications of most smartphones functionalitywise, it was possible to attain long test runs.

However, this test run ended abruptly less than two hours after its beginning, which can be regarded as a short time in the instance of a test run with relatively limited functionality. Moreover, the failure that terminated the test run was a system error of the SUT, which is definitely serious, especially in an application as simple as the calendar.

More precisely, the test run was around one hour and forty-nine minutes in duration, most of which consisted in creating calendar entries of various kinds, switching between different calendar views, verifying the onscreen user interface texts and closing and opening the calendar application itself. Everything was proceeding without issue,

1. Some details have been omitted due to confidentiality reasons.

until the test execution decided to create a new memo entry in the calendar, which resulted in the system error.

Before attempting to create the entry, there were several existing entries that might conceivably have clashed with the entry that could not be created. This observation was further sustained by the fact that the system error could not be reproduced when there were no existing entries in the calendar at the time of attempting the creation of the final memo entry. However, despite a number of rational conjectures as to the cause of the system error, the attempts to determine the real reason for the failure were ineffectual. On these grounds, it was decided to resort to the trace incrementation method in order to shorten the error trace, as the cause of failure remained unknown.

After creating the concatenation of action subsequences on Sequencer, it was run on the SUT. This experiment immediately yielded positive results, as the error trace leading to that specific system error could be substantially shortened from around 1850 keywords to approximately 100 keywords. This is a major improvement, as the shorter the error trace is, the easier it is for developers to determine the underlying cause of failure. In this case, the reduction of 1750 keywords is a huge amount of functionality, accounting for around 95 percent of the original error trace, which could now be ignored, as that functionality did not have any impact on the occurrence of the failure. The significance of this accomplishment is further illustrated by the fact that when running the downsized keyword sequence, this actual test run lasted only for around six minutes, which is very brief next to the near two-hour duration of the original test run.

This case study is an example of a scenario where the underlying fault of a system is so obscure that it cannot be readily accounted for. In these situations any debugging attempts to identify the cause of failure are of no avail and the only reasonable course of action is to shorten the error trace in order to facilitate the process of identifying the seemingly inexplicable fault. To this end, any functionality that can be disregarded in the original error trace can be beneficial in terms of further inquiries into the failure.

The approach based on trace incrementation is one that effectively pursues this end, and in this case study it removed approximately 95 percent of the original functionality from the error trace. It is considerably easier to determine the fault from the residual five-percent trace fragment than from the entirety of the original trace. Thus, this method had a great facilitating impact on the process of determining the underlying fault.

4.3. Discussion

The case studies presented two very common scenarios in debugging: a situation where reproducing the error would require much effort if carried out manually due to a long test log, and a scenario where the fault cannot be readily determined or located, leaving trace shortening the only viable option for debugging. These are both relatively common phenomena in online MBT, as tests based on this paradigm often tend to be long in duration, although this depends in great measure on the guidance algorithm in use for governing the test execution. The long duration of tests is, however, adverse in terms of debugging, and many faults uncovered by thorough testing can be difficult to account for, especially those related to concurrency and timing issues.

On the probability that either one or the other of the aforementioned scenarios occurs, it could require a considerable amount of resources to better appreciate what the underlying cause is without any supportive debugging methods. The debugging methods that were used in the case studies to address the described difficulties considerably facilitated the debugging process, enabling rapid fault detection and better insight into the functionality and sequence of events responsible for the failure.

The debugging problems that emerged were in themselves, however, not specific to MBT, as the difficulties that the developed debugging methods were created to negotiate are not caused by any MBT-specific feature. Instead, they are the product of the long test runs that the online MBT paradigm often produces due to the automatic and dynamic test generation. Thus, it can be argued that MBT was not the reason for the difficulties encountered during the research, and those same difficulties could have emerged with any automated testing methodology in use.

5. Conclusion

MBT features many advantages relative to the former generations of software testing. Regardless of these benefits, MBT has not been assimilated by the software testing industry to any large degree, and it is still relatively unknown as a paradigm. One reason for the low adoption rate of MBT may well be the drawbacks involved, one of which is the difficulty of debugging tests, which is especially true of online MBT.

As far as debugging is concerned, MBT as such does not, however, essentially differ from any other automated testing approach. Model-based tests are not intrinsically difficult to debug, but the difficulties stem from the long duration of the tests enabled by online MBT, and from the potentially complex combinations of functionality under test, especially those related to concurrency.

However, even though these issues may add to the perceived difficulty of debugging model-based tests, they can be effectively addressed by debugging methods expressly designed to counteract these difficulties in the MBT context, although based on common principles. The debugging methods developed during our research provide an efficient means to debug model-based GUI tests, which might prove difficult to accomplish by ordinary means. This is often

the case when the test run has been long and the failure has occurred far before its end, or the cause of failure cannot be readily understood, leaving trace shortening the best available course of action.

Of these methods, the video synchronization approach has proven very efficient when debugging long test runs where the failure and the sequence of events leading to it cannot be discerned firsthand, and a more thorough retrospect of the test run is needed. With this method it has been substantially easier and faster to discover the failure in a test run, particularly when the actual failure has occurred sometime before the immediate end of the test run. This finding was further sustained by the first case study.

The trace incrementation method was created to address scenarios where the underlying cause of failure is so difficult to comprehend that the only remaining option is to shorten the error trace that produces the failure. This method can considerably fast-track the process of determining the minimal sequence of actions necessary for the failure to recur, which was substantiated in the second case study.

These two methods outperform the two other debugging methods based on loop removal and transition-specific search in the context of GUI testing, as in this context the video synchronization method is especially applicable and the trace incrementation method can always reproduce the error, whereas this is not the case with the other methods.

The results obtained from the case studies act as an argument for the fact that debugging is not a bigger issue in MBT than in any automated testing approach. As this notion was reinforced by the results of the case studies, it can be argued that the difficulty of debugging is not an impediment to the prospective wider utilization of MBT. Furthermore, with the developed debugging methods the debugging process itself can be facilitated, even in relatively challenging error scenarios. While the usefulness of the video synchronization method is limited to GUI testing, future work includes more case studies to find the limitations of the trace incrementation method.

Acknowledgements

Partial funding from Tekes, Nokia, Symbio, Ixonos, Cybercom Plenware, F-Secure, Qentinel, Prove Expertise, as well as the Academy of Finland (grant number 121012), is gratefully acknowledged.

References

[1] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley, 2001.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing – A Tools Approach*. Morgan Kaufmann, 2007.

[3] H. Robinson, "Obstacles and opportunities for model-based testing in an industrial software environment," in *Proc. 1st European Conference on Model-Driven Software Engineering (2003)*, Nuremberg, Germany, Dec. 2003, pp. 118–127.

[4] A. Hartman, "AGEDIS project final report, 2004," Available at http://www.agedis.de/documents/FinalPublicReport(D1.6).PDF. Cited Jan 2010.

[5] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," in *Formal Methods and Testing*, ser. LNCS, 2008, no. 4949, pp. 39–76.

[6] Conformiq Software, "Conformiq Qtronic homepage," At URL http://www.conformiq.com. Cited Jan 2010.

[7] A. Jääskeläinen, M. Katara, A. Kervinen, H. Heiskanen, M. Maunumaa, and T. Pääkkönen, "Model-based testing service on the web," in *Proc. TESTCOM/FATES 2008*, ser. LNCS. Springer, Jun. 2008, no. 5047, pp. 38–53.

[8] H. Agrawal, "Towards automatic debugging of computer programs," Ph.D. dissertation, Purdue University, West Lafayette, IN, USA, 1992.

[9] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, pp. 183–200, 2002.

[10] M. Katara and A. Kervinen, "Making model-based testing more agile: a use case driven approach," in *Proc. Haifa Verification Conference 2006*, ser. LNCS, no. 4383. Springer, 2007.

[11] H. Buwalda, "Action figures," STQE Magazine, March/April 2003, pp. 42–47.

[12] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, USA, November 2007, pp. 417–420.

[13] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: finding the needle in the haystack," in *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 6–6.

[14] K.-H. Chang, V. Bertacco, and I. L. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *ICCAD '05: Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1045–1051.

[15] MPlayer homepage, http://www.mplayerhq.hu/. Cited Jan 2010.

[16] A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen, "Automatic GUI test generation for smart phone applications - an evaluation," in *Proc. of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society, 2009, pp. 112–122, companion volume.

[P6] ©2010 IEEE. Reprinted, with permission, from Proceedings of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009), “Automatic GUI Test Generation for Smartphone Applications – An Evaluation”, A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen

Automatic GUI Test Generation for Smartphone Applications – an Evaluation

Antti Jääskeläinen,
Mika Katara,
Antti Kervinen,
& Mika Maunumaa
Dept. of Software Systems
Tampere Univ. of Technology
P.O.Box 553
FI-33101 Tampere, Finland
{antti.m.jaaskelainen,
firstname.lastname}@tut.fi

Tuula Pääkkönen
Nokia Devices
P.O.Box 1000
FI-33721 Tampere
Finland

Tommi Takala & Heikki Virtanen
Dept. of Software Systems
Tampere Univ. of Technology
P.O.Box 553
FI-33101 Tampere, Finland
firstname.lastname@tut.fi

Abstract

We present the results of an evaluation where we studied the effectiveness of automatic test generation for graphical user interface (GUI) testing of smartphone applications. To describe the context of our evaluation, the tools and the test model library we have developed for the evaluation are also presented. The library contains test models for basic S60 applications, such as camera, contacts, etc. The tools include an on-line test generator that produces sequences of so called keywords to be executed on the test targets. In our evaluation, we managed to find over 20 defects from applications that had been on the market for several months. We also describe the problems we faced during the evaluation.

1. Introduction

Software test automation systems offer benefits over manual testing and are found useful in regression testing, especially at unit and integration testing levels. While conventional tools have automated the test execution phase, newer ones are also able to automate the test generation (i.e. the test design) phase. Deployed successfully, automatic test generation could bring benefits in lower testing costs not only in the form of reduced test design but also in maintenance, since the updated tests can be re-generated easily.

However, the industrial deployment of such tools has so far been hampered mainly by non-technical issues, such as poor usability of the tools and organizational obstacles [11, 30]. Moreover, studies supporting the transition from manual to automatic test design are few [24]. Thus, there is a need to develop easy-to-use tools, techniques, processes, etc. to support the deployment of the technology and run case studies to evaluate their effectiveness.

In this paper we concentrate on tools and techniques and evaluate an automatic test generation solution for graphical user interface (GUI) testing of smartphone applications. The basic approach is well known: generating tests from state machines modeling expected behavior. Smartphone applications, such as calendar, contacts, calculator, and camera differ, however, somewhat from standard desktop applications when considering GUI testing. Since embedded devices running such applications are limited in the physical size of their display and keyboard, as well as the processing power, the GUIs are usually simpler than their desktop counterparts. Moreover, since the tested products conform to the concepts of software product line, the reusability of the test artifacts and results is very important.

Unfortunately, testing through a GUI is usually much harder than using some test-specific interface of even an application programming interface (API). In order to run a test, we need to be able to input the parameters, data values etc. and check the outputs. However, not all operating systems provide direct access to the GUI resources, meaning that bitmap comparisons or optical character recognition (OCR) may be needed. In addition, GUIs are often volatile, i.e. they often reflect the changes in the requirements, thus increasing the test maintenance effort [17]. Nevertheless, many organizations developing smartphone applications for mass customer markets prefer to test the application behavior through GUIs from the point of view of the end-user experience. Manually testing all the language versions, for instance, could also be very expensive and time consuming.

The background of our approach has been introduced al-

ready in [14, 15, 19, 20, 21]. The contributions of this paper are in presenting the results of the evaluation and the experiences we gained in long-period testing. However, we first review the earlier results in order to explain our context. The remainder of this paper is structured as follows: In Section 2 the background of our study is presented in detail. Sections 3 and 4 introduce the goals of the study and describe the modeling approach, as well as the associated toolset. Sections 5 and 6 discuss running long-period tests and the results of our evaluation. Finally, Sections 7 and 8 review the related work and summarize the lessons learned.

2. Background

In this section we present the background of our evaluation. The system under test (SUT) is an application running on S60 [32], the most widely spread smartphone platform, with almost 200 million installations. An S60 phone resembles more a desktop computer than a regular phone, since a user can install his/her own applications.

2.1. Automatic test generation

While traditional test automation tools automate the execution of tests, new ones also automate the generation of tests. The basic idea in most of these tools is to derive tests based on high-level descriptions, or *test models*, of the SUT. The anticipated benefits of automatic test generation include better test coverage and reduced maintenance compared to the traditional test suites. Moreover, many defects can be found already in the modeling phase before executing any tests [21].

There are many types of approaches to automatic test generation (see [37] for a taxonomy). Our test-specific models specify the behavior of the GUI from the end-used perspective. These transition-based models are deterministic, untimed, and discrete. Moreover, the tests can be generated *on-line* concurrently with the test execution. This is regarded as beneficial, especially in testing non-deterministic reactive systems like smartphones, whose response to a given input may be hard to predict due to concurrency and an uncontrollable environment like network. There are no test cases in the conventional sense and testing can be seen as a game between the tool and the SUT [23].

Alternatively, in *off-line* testing the test suites are first generated from the models and executed in a separate phase. This solution is more compatible with conventional thinking as well as existing development processes. In addition, it provides a means to generate conformance testing suites to be executed by 3rd parties using pre-existing tools, for instance. Spec Explorer [6], reportedly being used by several Microsoft product groups on a daily basis, is an example of a tool supporting both of these approaches.

In either case, the generated tests are usually too abstract to be executed directly. Hence, a transformation is needed that converts the tests to a form understood by the SUT. In the off-line case, a separate transformation phase is needed after generating the abstract test suite. However, in the on-line case, a special SUT adapter translates messages between the test generation tool and the SUT (or a test tool accessing the SUT).

2.2. Choosing the right test interface

System level software test automation needs to access the SUT using some interface enabling communication between the two. In practice, such communication includes setting up the state of the SUT prior to a test run, inputting the events and data values as specified in the test, and checking the actual results against the expected ones.

There are basically three options when considering which interface to use for this kind of communication. A test-specific interface can be designed just for the purposes of supporting test automation. Unfortunately, test automation engineers seldom have the luxury of testing a system with a built-in test interface.

On the other hand, system level test automation can be implemented through a high-level application programming interface (API). The benefits of using such an interface include stability and performance. APIs that have been published for third party developers are usually stable enough. They also provide efficient implementation of functionality to access systems. However, compared to the test-specific interfaces, extra work is required to implement the functionality needed by the test automation tool.

The third alternative is the user interface, which is usually a graphical one in most modern applications. However, GUIs are much more volatile than APIs, since changes in the requirements are often reflected in the user interface. In addition, the performance and accuracy can be poor if, for instance, comparison of the test results involves low-level techniques such as bitmap comparisons or OCR. On the other hand, automated GUI testing does not always require separate middleware for adaptation, at least in a standard UI environment such as MS Windows. Moreover, since the end users interact with the system through the GUI, using the same interface for testing is likely to focus on relevant behavior. It can also reveal problems related to GUI-specific issues that would be hard to detect otherwise [21].

Different types of testing complement each other. Combined with automated unit and integration tests, as well as advanced manual techniques such as exploratory GUI testing [17], the choice between the three alternative interfaces needs to be done on a case-by-case basis. This decision can be affected by the availability of appropriate tools and expertise as well as non-obvious organizational issues.

2.3. GUI test automation in S60

Automating GUI testing is often not considered an optimal solution, mainly due to many bad experiences with so called capture/replay tools [8, 17]. These first generation tools captured GUI events and produced low-level scripts that were hard to understand and maintain. Even the slightest change in the GUI forced a recapturing of the script. These tools seldom provided a positive return on the investment and often ended up as shelfware.

It was soon realized that scripts need to be structured and modularized like any other code of sufficient size [8]. Moreover, with the introduction of the so-called data-driven approach, it was possible to reuse the same test execution engine with different data-values that were commonly separated on a spreadsheet. This facilitated localization testing for different languages, for instance.

The state of the art in automated GUI testing is represented by so called *keywords* and *action words* [5, 8]. They help in separating concerns by abstracting from the concrete GUI. The idea is to map the user requirements, captured for instance in use cases, to high-level events called action words. In the smartphone context, such action words can include events for opening a Calendar application, adding a contact to the list of contacts, or sending an SMS. On the other hand, keywords corresponding to key presses and GUI navigation provide the lower level of abstraction. For instance, a keyword kwPressKey<Center> corresponds to pushing the centre button that usually chooses the current selection in the menu. Checking the results is done using keywords such as kwVerifyText<'string'>, which verifies that a given string argument is visible on the display.

The separation of concerns provided by action words and keywords enables non-technical testers to develop tests by creating action word sequences based on requirements. Test automation engineers, on the other hand, can concentrate on implementing the keywords in the particular SUT. Keywords and action words and similar abstractions are commonly found in commercial GUI testing software.

2.4. Software product line testing

According to [34] a software product line is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Software product lines enable the introduction of new software products at a pace much faster than traditional approaches. This poses a challenge to testing, since tests should also be seen as reusable assets [36]. However, it can be very hard to determine what tests to skip just because they were executed for the previous product. Moreover, re-testing using the same artifacts may not be directly possible because of differences in features and GUIs between the products.

In our context, it can be assumed that the basic functionality provided by the GUI stays the same across the product line of smartphones. Some devices have larger displays or extended keyboards, but still provide the same look and feel as well as the basic set of applications. Due to the openness of S60, there are plenty of third party applications available. To enforce quality guidelines for such applications and to support network operator requirements, there are some common test requirements [35].

From the GUI testing perspective, the reusability and maintainability of test artifacts is seen as extremely important. There is a need to reflect in the test artifacts the separation of concerns between the things that stay the same and those that change across the product line. On the one hand, it should be possible to reuse the existing tests as much as possible when testing a new product. On the other hand, things that tend to change, such as keyboard related test automation code, should be modularized in a way that enables easy adaptation.

A solution to this problem is provided by action and keyword techniques. The implementation of keyboard related issues can be separated at the keyword level and the basic functionality can be encoded as action words. For each key in the keyboard, there could be a separate keyword. However, the number of possible action words can be much greater, which can pose new maintainability problems if not managed properly.

3. Goals of the evaluation

Our approach to evaluation was more qualitative than quantitative. On the one hand, it would have been more convincing to be able to provide exact data and to compare different approaches (see, for instance, [27]). On the other hand, since testing is a context-sensitive activity with no best practices working across different contexts [17], we rather concentrated on providing the kind of evidence our industrial partners requested, i.e. a proof-of-concept. Hence, we had several goals and some of the more general ones were simply related to trying out automatic test generation in the domain of smartphone applications and disseminating experiences to the partners. Towards this end, there was a need to have a prototype version of the toolset running as soon as possible, in order to be able show demos and promote discussion with technical experts using conventional test automation tools in this domain.

We also had more concrete goals. The first requirement was to be able to run tests that use two phones instead of just one (sender and receiver of a SMS message, for instance). The second requirement was to include test data in the test runs in an easy-to-use way. The third requirement was to

run tests on different products of the same product line in order to assess the reusability of the test models in a product line setting. Finally, the fourth requirement was to try to derive test models from design models. All in all, during the actual test runs, we wanted to find real defects that would be out of reach of conventional testing tools.

In the beginning we also anticipated that we could train test engineers to build test models. However, this requirement was later abandoned, as will be explained.

4. Model library and the toolset

Since there were no suitable tools available when the evaluation started, we had to develop our own. We did not want to invent new theories; instead, we wanted to apply existing ones in the particular domain at hand. This section describes the tools and techniques that were developed towards this end. The solutions are domain-specific, i.e., they are tailored for smartphone application testing. However, most solutions should be adaptable to other domains also. Based on [14, 20], we begin by presenting our approach to test modeling and then describe the tools we have built.

4.1. Modeling

Domain-specific modeling languages (DSMLs) are gaining popularity in the area of model-driven development. Instead of using standard generic languages such as UML, the idea is to specify a new modeling language for a certain application area. In principle, this enables domain experts without programming skills to create high quality models. Moreover, custom-made code generators mapping the models to code can produce more efficient implementation on the target platform than generic ones. There are some industrial success stories in deploying DSMLs that report huge improvements in productivity [7]. Since building custom-made tools requires expertise, time and effort, the domain needs to be stable in order to obtain the return for the investment in the long run. Moreover, an organization lacking the expertise to create a customized language and the associated tools may become too dependent on a specific tool vendor. Nevertheless, the growing tool support for DSMLs will enable easier customization in the future.

Taking into consideration the trade-offs involved, the use of DSMLs is regarded as beneficial also in testing [12]. System level testers are usually not familiar with generic modeling languages such as UML or even testing languages such as TTCN-3. Hence, a DSML built on the concepts and abstractions of the problem domain, using keywords and action words, for instance, can be seen as a better option.

Our domain-specific approach for testing S60 applications combines two very different techniques. First, the idea of keywords and action words is adopted from GUI testing. The set of keywords is fixed and chosen to fit in the S60 domain. Action words, on the other hand, can be chosen freely by the modeler. Second, LSTSs (Labeled State Transition Systems, that is, digraphs with labeled edges and nodes and one of the nodes marked as a special "initial state") and their parallel composition are chosen as the underlying modeling formalism. Synchronizations in the parallel composition, which is generalized [18] from CSP [31] parallel composition, are defined to support modeling S60 applications and their interactions. However, the parallel composition and some of the LSTSs are hidden from the users of the language. The formal definition for LSTS is as follows:

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.*

The parallel composition of LSTSs [10] is based on a rule set explicitly defining which actions are executed synchronously. 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 its execution. The following definition is slightly modified in two respects: internal transitions are not needed and handling of state propositions is more straightforward:

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* $\|_R (L_1, \ldots, L_n) = repa((S, \Sigma, \Delta, \hat{s}, \Pi, val))$, *where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \Sigma_R$
- $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ *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* $s_i = s'_i$
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$
- $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$
- $val((s_1, \ldots, s_n)) = val_1(s_1) \cup \cdots \cup val_n(s_n)$
- *repa is function restricting LSTS to contain only the states which are reachable from the initial state* $\hat{s}$.

Action and keyword tiers consist of test model components (LSTSs). The components in these tiers are called *action machines* and *refinement machines*, respectively. They are used for building test models. Action machines model the user actions at the high level using action words. Refinement machines transform the action words into sequences of keywords, i.e., executable events in the user interface. Next, the tiers will be discussed in detail.

4.2. Action tier

Action machines in the action tier model applications of the SUT at a high level of abstraction. The machines contain action words, the executions of which can be interleaved to the extent defined in this tier. This enables testing joint behaviors of different applications running concurrently.

As indicated above, interleaving the executions of the action machines is an important part of our domain-specific modeling approach. Applications running on S60 should always be interruptible. User actions, such as received phone calls and messages, may stop the ordinary execution of the application at any time. However, implementing an application that behaves properly in every situation is very hard due to the inherent complexity imposed by concurrency. Moreover, the number of event interleavings that could be tested is far beyond the capabilities of ordinary linear and static test cases. To allow the easy creation of test models with automatically interleaved action machines, the concepts of *sleeping* and *running* action machines were introduced.

In Figure 1 $\text{Camera}_{\text{AM}}$ is a simple action machine for testing the Camera application. The states of action machines can be divided into running (in the foreground in the figure) and sleeping (in the background) states. The semantics for the running and sleeping action machines has been adopted from the platform, a multi-tasking operating system sharing one processor for several processes. Exactly one action machine at a time is in a running state. The running action machine can be changed only if it is in a running state from which there is a Sleep transition to a sleeping state. During the test run, it depends on the test generation algorithm if this transition is executed. Action words can be executed only between the running states.

The initial state of the $\text{Camera}_{\text{AM}}$ action machine in Figure 1 is the filled node in the background. When it is switched to a running state (Wake_{TS}), the test generation algorithm can choose between starting the Camera application (awStartCam) or immediately switching back to the sleep mode (Sleep_{TS}). In the former case it has to be verified that the application seems to be running correctly (awVerifyCam), after which there are three possibilities: taking a picture (awTakePhoto), quitting the application (awQuit), or leaving the application running in the background and switching to another application (Sleep_{TS}).

In addition to the generic Sleep_{TS}-Wake_{TS} primitives, after which any action machine able to execute Wake_{TS} can be taken to a running state, it is possible to use Sleep_{App}-Wake_{App} primitives, which wake up explicitly specified action machines instead of just any. Both sleeping and waking primitives were inspired by the two possible ways the user can activate an application in S60. The former corresponds to the situation where a task switching application, modeled by an automatically generated task switching action machine, is used for activating some application running in the background. On the other hand, the latter is used for modeling the user activating a specific application directly from another application. For instance, it is possible to activate the Gallery application just by choosing "Go to Gallery" from the menu of the Camera application.

There is also a communication mechanism defined for exchanging information on shared resources between action machines. For this purpose, primitives for requesting (Req) and giving permissions (Allow) are used. The former can be executed in running states and the latter in sleeping states. However, these primitives cannot wake up a sleeping action machine or put a running one to sleep. For example, $\text{Camera}_{\text{AM}}$ in Figure 1 allows other action machines the use of the image it just took by executing Allow<UseImage>.

It should be noted that the machine could have been drawn using the UML state machine notation, if so desired. In fact, a modeling tool could be used for customizing the visual appearance of the state machine according to which notation is the most familiar one to the test modeler. Thus, the exact visual notation is not an important part of this DSML. Instead, the core is in conventions for naming the labels of the model components and in the associated semantics, enabling simple maintenance and rapid development of new test model components when necessary.

4.3. Keyword tier

The executable test model is obtained using action and refinement machines. The purpose of the latter is to refine the action words in action machines to sequences of executable events in the GUI of the SUT.

As discussed above, in a product line context reusability is paramount. To support the testing of different products of the same product line, the functionality must be separated from the GUI events. This allows reusing action machines with SUTs supporting the same operations but with a different GUI. For example, the Camera application can have exactly the same functionality in two devices, one having a regular keyboard and the other an extended one. However, designing an action machine is far from trivial: it requires much effort and insight into what is worth testing. Nevertheless, after designing an action machine, defining the corresponding refinement machines should be much easier.

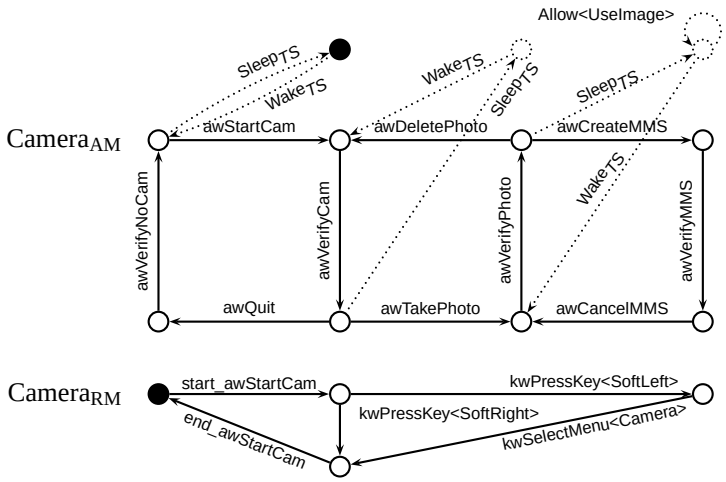

Figure 1. Camera action machine (Camera$_{AM}$) and one of its refinement machines.

In Figure 1, Camera$_{RM}$ is a refinement machine for the Camera application. In its initial state (the filled circle) the machine is able to refine starting the Camera application (awStartCam). The application can be started by two different keyword sequences: by selecting it from the menu (kwPressKey<SoftLeft> opens the menu and kwSelectMenu<Camera> chooses the menu item), or by using a shortcut (kwPressKey<SoftRight>, "SoftRight" key is the shortcut).

Keywords serve two purposes: generating input events and making observations. In a test run, keyword execution always either succeeds or fails. For example, kwVerifyText<'Camera'> fails if the text "Camera" cannot be found on the display. Sometimes the failure is allowed or is even required behavior. The allowed results are expressed using the labels of the transitions. If the keyword starts with (without) a tilde, then the failure (successful execution) is allowed in the starting state of the transition. The next state after execution depends on the execution result.

The keyword tier consists of several refinement machines, each of which interacts with a single action machine on the action tier. Usually, the refinement is as simple as a macro expansion: every transition labeled with the same action word is replaced with the same sequences of keywords. On the other hand, sometimes the sequences may vary, depending on the action words executed earlier. For example, the keyword sequences implementing "activate the Camera application" are different, depending on whether the application is already running in the background or not.

The refinement is not allowed to change the behavior (safety and liveness properties) of the action machine. To be more exact, a valid refinement machine contains neither deadlocks (and should not cause them when composed in parallel with its action machine) nor infinite sequences of keywords, i.e., directed loops of keyword transitions. Verifying this automatically is straightforward.

Many keywords require one or more parameters. Sometimes these parameters are fixed to the GUI, such as a parameter defining which key to press; sometimes they represent real-world data: a date or a phone number, for instance. Embedding this information directly into the models would be problematic; they would be limited to a fixed set of data values and possibly tied to a specific test configuration. Another major problem with data is that storing it in state machines means duplicating states for each possible value of data, quickly resulting in a state space explosion [38]. To solve these problems, we have developed two methods for varying the data in models: so called *localization data* and *data statements*. The basic function of the former is to store the text strings of the GUI in different languages. This way the models need not be dependent on any specific language variant of the SUT. In practice, the data is incorporated into the model by placing a special identifier in a keyword. When that keyword is executed, the identifier is replaced with the corresponding element from the localization tables. Even more complicated use of data can be accomplished by placing *data statements* (Python [28] code) in actions. Such statements may be used in any actions, not just keywords. Moreover, data provided by external *data tables* can be used in these data statements.

4.4. Example

As a concrete example, consider testing a Camera application. Besides testing its features alone, its interoperability with some other applications should be tested. There are several modes in the Camera application, like timer, video capturing and file renaming, all of which should correctly recover from interrupts caused by incoming messages and phone calls, for example.

When a new version of the Camera application arrives in a test lab, it may not be wise to start a full interoperability test right away. It may take hundreds of testing steps to detect a simple error, because the number of possibilities from which the test generation algorithm can choose is very large. Instead, it may be better to test first all the features of the application separately. After passing the first test, it is time to start a new and longer test that interrupts the Camera application with incoming calls and messages in all modes.

In an interoperability test case a test model could include model components for testing Camera, Messaging, and Telephone applications. This means that the action machines corresponding to the applications are composed in parallel with the refinement machines whose purpose is to refine the action words in the action machines for the particular device that is being tested. In addition to those LSTSs, a bogus application model could be included in the parallel composition causing incoming calls and messages. Moreover, an automatically generated task switcher action machine switches between the four possible applications. In the initial state of the test model, the task switching action machine is the running action machine while the other action machines are asleep. It is up to test generation algorithms to decide which action machine is woken up first.

4.5. Test generation and modeling tools

Figure 2 illustrates the toolset architecture that consists of four parts and a database. The first one is the model design part, which is used for creating the component models and data tables. The second is the test control, where tests are launched and observed. The third one is the test generation part responsible for assembling the tests and controlling test execution. The fourth is the keyword execution part that communicates with the SUT through its GUI.

The model design part consists of two primary design tools: Model Designer [13] and Recorder [33]. The latter is an event capturing tool that has been designed to create keyword sequences. The sequences can then be formed into refinement machines. Model Designer is the tool for creating action machines and data tables. In addition, it is responsible for assembling the models into a working set ready for testing. The model repository is used for storing the elements of this working set.

After the models have been prepared, the focus moves to the test control part that contains a web GUI used for launching test sessions. Once a test session has been set up using the web GUI, the Test Control tool (in the test generation part) takes over. First, the tool checks the *coverage requirement* (a formal test objective, see [19] for details) that it received and determines which model components are needed for the test run. These are then given to Model Composer that combines them into a single model on the fly. Test Engine manages the model and determines what to do next. For this purpose, it receives parameters from Test Control. Both Test Control and Test Engine write into a test log, which may be used for observing, or repeating the test for debugging purposes.

Moving on to the keyword execution part, as keywords are executed in the model, Test Engine relays them to this part, whose purpose is to handle their execution in the SUT. The SUT responds with the success status of the keyword, i.e. true or false, which is then relayed back to Test Engine. First, a specific adapter tool translates the keywords into a form understood by the receiver. Moreover, it manages the gradual execution of some more complex keywords. The next part in the communication chain is the test tool, which directly interacts with the SUT and is thus SUT specific. Hence, it is not provided alongside the toolset, but the users of the toolset must provide their own test tool.

The architecture has been designed to support the plugging-in of different test generation heuristics. Initially, we implemented three heuristics, which allowed us to experiment with the tools. The first one is a purely random heuristics that can be used in bug hunting. The two other heuristics are based on game-theory and are to be used in use case driven testing [19]; we have implemented a single thread and a two thread version. Moreover, the duration of a test run can be limited, allowing, for instance, smoke testing in a continuous integration cycle [9]. The difference between the two game heuristics is that the latter continues searching an optimal path to a state that fulfills the coverage requirement, while the other thread waits for a return value from keyword execution. Unfortunately, we ran into performance problems even with the two thread version.

Towards solving these performance problems, we developed yet another test generation algorithm that searches for the shortest (loopless) path, whose execution changes the coverage value. The algorithm outperforms the game-theoretic ones in many cases. However, having implemented several test generation algorithms, it seems that none of them is better than others in every case. Their performance, when measured as the growth of the coverage in the function of time, depends on the size of the model, the coverage requirement and the speed of keyword execution, i.e., the speed of the adapter layer and the SUT. Hence, an optimal algorithm should be able to change the strategy automatically.

5. Running long-period tests

Already at the beginning of the evaluation it was realized that we needed to run long-period tests to find new defects. The applications under test were already thoroughly tested using conventional methods. Some of them had been on the market for several months before we obtained them. Short

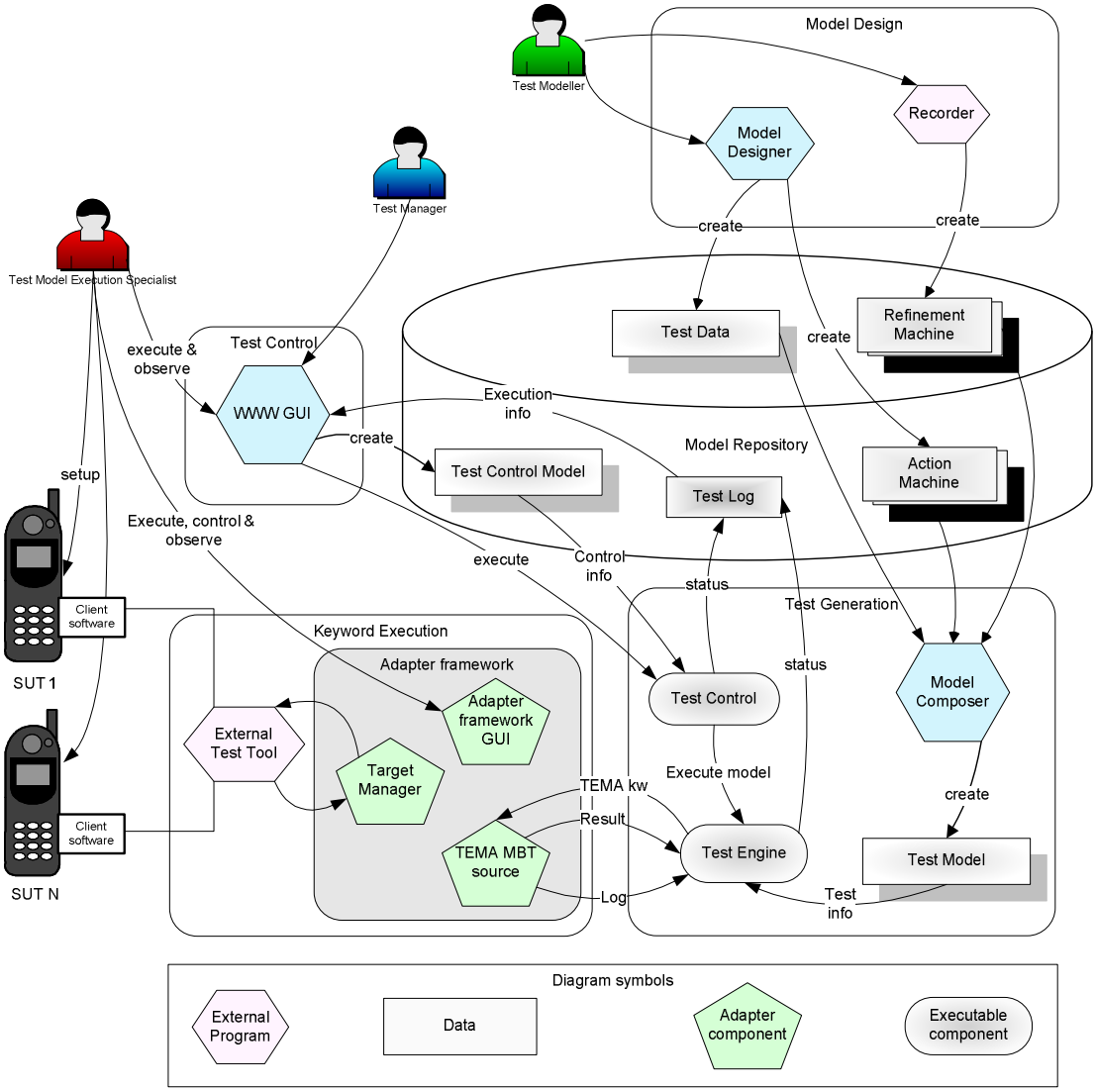

Figure 2. Test tool architecture (adapted from [14]).

tests were just unlikely to find any defects.

As already mentioned, we had performance problems with the test generation heuristics that we were able to solve. However, the most difficult part was the development of a suitable tool adaptation for the keyword execution part. The problem lay in finding a suitable third party test tool that would be reliable enough for running long-period tests. A lot of time was spent on trying to get a certain test tool to work with our tools, until this solution was abandoned. Since the tool was not open source and we had no access to the source code, we were depending on the vendor to fix certain problems. We also considered developing our own tool for accessing the SUTs, but this idea was dropped due to the great complexity of the task. Before we started to use another tool, i.e. ASTE [25], we were only able to run tests that lasted a few minutes.

After solving the adapter problems, another unanticipated problem arose: the Bluetooth connection used for accessing the smartphones from a PC running the keyword execution part was cut after about 50 hours of test execution. A solution was to replace Bluetooth with an USB connection that proved to be more reliable.

6. Results of the evaluation

As already mentioned, we had several goals in our evaluation. One of the general goals was to have a prototype

version of the toolset running as soon as possible. This succeeded in spite of the problems in test generation performance and the third party test tool: we were able to show short demos to the technical experts. The results regarding the more concrete goals are described below.

Ability to run tests that use two phones instead of just one: We identified problems in our modeling methodology that made it quite difficult to compose arbitrary test model components to be run on two test targets. The reason was that the keywords used for switching between the target devices were not modeled initially. Moreover, we needed to copy some model components in the case of testing the same application in both targets. These issues were solved and this requirement was fulfilled.

Inclusion of test data to the test runs in an easy-to-use way: this was achieved with localization data and data statements, as discussed above.

Running tests on different products of the same product line: The test model library was developed originally for versions 2 and 3.0 of the S60 platform. However, by the time long-period testing become possible, our primary test target conformed to version 3.1 of the platform. We were able to use most of the model library as such, and the maintenance efforts were restricted mainly to the keyword models, as anticipated. Thus, the requirement was satisfied. However, we identified a certain problem related to the variability between versions 3.1 and 3.2, which needs special attention in the next generation model library.

Derivation of test models from design models: This requirement was not fulfilled. In principle, we could tag UML design models with stereotypes in order to identify keywords and action words. However, finding suitable design models for the purposes of such a transformation is not easy; generally, they do not lend themselves to be used as input for our test modeling. The design models do not model the behavior of the applications in concurrency setting and the models are either too generic or too detailed. Thus, instead of specifications, our models were based on observations and, to some extent, common sense. The lack of specifications gave us a rather realistic setting; nowadays popular agile methods do not encourage detailed specification, instead the implementation is seen as the most important artifact. This also led us to use some exploratory testing practices to develop the models, and we were also able to find some real defects while modeling.

During the evaluation, we also identified a need to develop new metrics and testing processes for test management that could be used in deployment of this technology. However, these requirements were not included in this evaluation, but left as future work.

We also wanted to find real defects that would be out of reach of conventional tools, thus proving the effectiveness of automatic test generation. In total, 21 defects of different severities and priorities were found from built-in applications in S60 smartphones, such as Gallery, Music Player, Flash Player, Messaging, Notes, and Voice Recorder. Some of these defects existed in more than one smartphone model. The most severe of the defects caused the phone to hang with "System error" message on the display. To reproduce this particular case, a test run of around 110 minutes and 1850 keywords was needed. Later, a much shorter run of around 6 minutes and 100 keywords was found.

About two thirds of the defects were discovered while modeling (exploratory testing), and the remaining third by executing the tests. Most of the defects had already been previously found in traditional testing (both manual and automatic test execution), but they had not been fixed for some reason. However, there were also some that were totally new. Many of the defects were related to concurrency issues: performing some multimedia-related functionality in one application and then switching to another application causes unexpected behaviors in some circumstances. This conforms to our earlier findings [21]. In addition to defects found in applications, some were found in both proprietary and commercial test tools, which was considered rather surprising, as these tools were quite mature.

The most surprising results of this study were that the modeling was easier than anticipated but the adapter development took much more time than planned. The model library (see [15] for details) contains 11 different applications that were modeled in some 110 action machines, with a corresponding number of refinement machines. Separately, the action machines contain about 1300 states, 1700 actions (perhaps 40% of them action words) and 3200 transitions. Refinement machines add roughly 3000 states, 3000 actions and 4100 transitions to the totals. The first version of the model library took about two months to build by a talented student with no prior experience in modeling. Another month was spent on debugging and maintaining the library. We envisioned in the beginning that testers would be able to build high quality models using our DSML. However, it was realized that it is much better to have a separate expert role dedicated to modeling. Some testers may want to learn modeling skills, but another new role of test model execution specialist may be an easier option.

7. Related work

Concerning related work, the idea of using general purpose GUI test automation tools for automatic test generation originates from Robinson [29]. Ostrand et al. [26] proposed a visual test design environment to create, edit, and maintain test scripts. They used a commercial test tool to capture GUI information and replay that information back to the SUT. Memon [22] proposed a framework for testing GUI applications. The framework is based on 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 run-time execution information. Belli [2, 3] extended state machines to show not only correct GUI actions, but also incorrect transitions.

Use cases (or sequence diagrams) as well as more expressive formalisms, such as state machines [1, 16], have been previously suggested to drive test generation. Traceability between requirements and model-based tests has also been studied before. For instance, Bouquet et al. [4] present an approach where the idea is to annotate the model used for test generation with requirement information. The formal model is tagged with identifiers of the requirements, allowing model coverage to be stated in terms of requirements. This allows automatic generation of a traceability matrix showing relations between the requirements and the generated test suite.

As already mentioned, our primary objective was not to invent new theories on GUI testing. Instead, existing ones were adapted to facilitate the deployment of automated test generation in the particular testing context. This was the reason for extending the keyword and action word techniques with model-based practices. Compared to Buwalda's approach [5, 8], the main methodological differences are in using LSTSs and their parallel composition to enable automatic creation of keyword sequences; in [5] state machines and decision tables expressed in spreadsheets are recommended for test generation. LSTSs offer a visual formalism that should be quite easy to grasp and parallel composition enables automatic generation of concurrency related tests provided the test models have been created with the domain-specific synchronization mechanisms discussed earlier. Moreover, it seems that use-case driven test guidance using sequences of action words has not been considered before, at least in this context. There are also some minor differences in the terminology: in our approach, low-level action words are referred to as keywords. In practice, the most generic keywords can be considered as action words in the sense of functionality. Thus, the main difference is in the purpose of use and the level of abstraction.

8. Conclusions

To summarize the results of our evaluation, we managed to reach most of the original goals. Most importantly, there is now some evidence that automatic test generation can find defects effectively from smartphone applications: we were able to find issues from applications after they had been on the market for several months and heavily tested using conventional methods before release. This provides a good basis for future studies, since more technical experts are becoming interested in this new technology. Moreover, we have set up a web-based test service, where the inherent complexity of the test models and algorithms is hidden from the end user (test model execution specialist) [14]. The idea is to use an expert test modeler for maintaining and extending the model library. Using such a service, if a new application is introduced and modeled, it is fairly easy to test its interworking with the built-in S60 applications. Obviously, in an open development environment such as S60, this is important both from the point of view of the platform and individual applications. The tools described in this paper as well as the model library are available under the MIT open source license.

Modeling was easier than anticipated, but the adapter development took much more time than originally planned. The first version of the model library took about two months to build by a student with no prior experience in modeling; another month was spend on debugging and maintaining the library. The negative result from our study was that design models seem to be very difficult to use as input for test modeling, at least in this context.

In spite of the easy-to-use test generation service, there are still severe obstacles in the deployment of this technology in the smartphone applications domain. Many of these issues relate to organizational changes, the need to develop new metrics and processes, etc. that remain as future work. However, management support for pursuing these new goals is easier to obtain now that there exists a proof-of-concept showing the effectiveness of the technology.

Acknowledgments

Partial funding from Tekes, Nokia, Conformiq Software, F-Secure, and Plenware, as well as the Academy of Finland (grant number 121012), is gratefully acknowledged.

References

[1] AGEDIS Consortium. AGEDIS project homepage. Available at `http://www.agedis.de/`. Cited Oct 2008.

[2] F. Belli. Finite-state testing of graphical user interfaces. In *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, pages 34–43. IEEE CS.

[3] F. Belli, C. J. Budnik, and L. White. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification and Reliability*, 16(1):3–32, 2006.

[4] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting. Requirements traceability in automated test generation – application to smart card software validation. In *Proc. ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST)*. ACM.

[5] H. Buwalda. Action figures. *STQE Magazine, March/April 2003*, pages 42–47, 2003.

[6] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with Spec Explorer. In *Proc. Formal Methods 2005*, number 3582 in Lecture Notes in Computer Science, pages 542–547. Springer.

[7] Domain-Specific Modelling Forum. DSM case studies and examples. Available at `http://www.dsmforum.org/cases.html`. Cited Oct 2008.

[8] M. Fewster and D. Graham. *Software Test Automation: Effective use of test execution tools*. Addison–Wesley, 1999.

[9] M. Fowler. Continuous integration. Available at `http://www.martinfowler.com/articles/continuousIntegration.html`. Cited Oct 2008.

[10] H. Hansen, H. Virtanen, and A. Valmari. Merging state-based and action-based verification. In *Proceedings of ACSD 2003, the Third International Conference on Application of Concurrency to System Design*, pages 150–156. IEEE CS, 2003.

[11] A. Hartman. AGEDIS project final report. Available at `http://www.agedis.de/documents/FinalPublicReport%28D1.6%29.PDF`, 2004. Cited Oct 2008.

[12] A. Hartman, M. Katara, and S. Olvovsky. Choosing a test modeling language: a survey. In *Proc. Haifa Verification Conference 2006*, number 4383 in Lecture Notes in Computer Science, pages 204–218. Springer, 2007.

[13] A. Jääskeläinen. A domain-specific tool for creation and management of test models. Master's thesis, Tampere University of Technology, Jan. 2008.

[14] A. Jääskeläinen, M. Katara, A. Kervinen, H. Heiskanen, M. Maunumaa, and T. Pääkkönen. Model-based testing service on the web. In *Proc. TESTCOM/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 38–53. Springer.

[15] A. Jääskeläinen, A. Kervinen, and M. Katara. Creating a test model library for GUI testing of smartphone applications. In *Proc. 8th International Conference on Quality Software (QSIQ 2008) (short paper)*. IEEE CS.

[16] C. Jard and T. Jéron. TGV: theory, principles and algorithms – a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *STTT*, 7(4):297–315, 2005.

[17] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley, 2001.

[18] K. Karsisto. A new parallel composition operator for verification tools. Doctoral dissertation, Tampere University of Technology (number 420 in publications), 2003.

[19] M. Katara and A. Kervinen. Making model-based testing more agile: a use case driven approach. In *Proc. Haifa Verification Conference 2006*, number 4383 in Lecture Notes in Computer Science, pages 219–234. Springer, 2007.

[20] M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Satama. Towards deploying model-based testing with a domain-specific modeling approach. In *Proc. TAIC PART – Testing: Academic & Industrial Conference*, pages 81–89, Windsor, UK, Aug. 2006. IEEE CS.

[21] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. Model-based testing through a GUI. In *Proc. 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005)*, number 3997 in Lecture Notes in Computer Science, pages 16–31. Springer, 2006.

[22] A. M. Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, University of Pittsburgh, 2001.

[23] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 55–64. ACM.

[24] A. D. Neto, R. Subramanyan, M. Vieira, G. H. Travassos, and F. Shull. Improving evidence about software technologies: A look at model-based testing. *IEEE Software*, May/June:10–13, 2008.

[25] M. Nikkanen. Use case based automatic user interface testing in mobile devices. Master's thesis, Helsinki University of Technology, May 2005.

[26] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *ISSTA '98: Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 82–92. ACM.

[27] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proc. 27th international conference on Software engineering*, pages 392–401. ACM.

[28] Python. Python Programming Language homepage. `http://python.org/`. Cited Oct 2008.

[29] H. Robinson. 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`. Cited Oct 2008.

[30] H. Robinson. Obstacles and opportunities for model-based testing in an industrial software environment. In *Proc. 1st European Conference on Model-Driven Software Engineering*, pages 118–127, Nuremberg, Germany, Dec. 2003.

[31] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[32] S60. `http://www.s60.com`. Cited Oct 2008.

[33] M. Satama. Event capturing tool for model-based GUI test automation. Master's thesis, Tampere University of Technology, Sept. 2006.

[34] Software Engineering Institute. Software product lines. `http://www.sei.cmu.edu/productlines/`. Cited Oct 2008.

[35] Symbian. Symbian Signed. `http://www.symbiansigned.com/`. Cited Oct 2008.

[36] A. Tevanlinna, J. Taina, and R. Kauppinen. Product family testing: a survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12–12, 2004.

[37] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Working paper series 4, Department of Computer Science, University of Waikato, New Zealand, 2006.

[38] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, number 1491 in Lecture Notes in Computer Science, pages 429–528. Springer-Verlag, 1998.

[P7] With permission from Taylor & Francis Group: Model-Based Testing for Embedded Systems, “Model-based GUI Testing of Smartphone Applications: Case S60 and Linux”, ser. Computational Analysis, Synthesis, and Design of Dynamic Systems, A. Jääskeläinen, T. Takala, and M. Katara, to appear

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-2513-1
ISSN 1459-2045