2011 Fourth IEEE International Conference on Software Testing, Verification and Validation

Experiences of System-Level Model-Based GUI Testing of an Android Application

Tommi Takala and Mika Katara
Department of Software Systems
Tampere University of Technology
Finland
{*tommi.takala, mika.katara*}*@tut.fi*

Julian Harty
eBay
jharty@ebay.com

***Abstract*—This paper presents experiences in model-based graphical user interface testing of Android applications. We present how model-based testing and test automation was implemented with Android, including how applications were modeled, how tests were designed and executed, and what kind of problems were found in the tested application during the whole process. The main focus is on a case study that was performed with an Android application, the BBC News Widget. Our goal is to present actual data on the experiences and to discuss if advantages can be gained using model-based testing when compared with traditional graphical user interface testing. Another contribution of this paper is a description of a keyword-based test automation tool that was implemented for the Android emulator during the case study. All the models and the tools created or used in this case study are available as open source.**

***Keywords*-Model-Based Testing; Automatic GUI Testing; Keyword-Driven Testing; Automatic Test Generation; Android**

I. Introduction

Model-based testing (MBT) is a testing methodology where the SUT is described with a formal model at such a level of detail that it can be used to automatically generate tests. The test generation is based on algorithms that process the model and produce tests in a desired manner. A very common way to describe a model is to use state machines, and thus graph algorithms can be used in test generation. Some of the advantages of MBT are the reduced amount of test maintenance (maintaining models instead of large sets of test scripts), and a basically unlimited number of different tests that can be generated [1].

Although discussed and researched actively for some years, MBT has not been widely used in the industry. Major obstacles in the adoption of MBT include organizational difficulties and the lack of easy-to-use tools [2], [3]. Some of the organizational difficulties are the new expertise needed in the modeling effort and the absence of formal specifications that could be used as a basis for models. We aim to lessen the tool issue in one domain with the tools presented in this paper.

Android is a popular open source operating system for mobile devices, especially smartphones. In the second quarter of 2010, Android climbed to be the leading operating system in smartphones sold in the United States [4]. The Android platform is based on the Linux operating system kernel and is being developed by the Open Handset Alliance, led by Google. The Android framework consists of four layers. The bottom layer of the framework is the Linux kernel working as the hardware abstraction layer (HAL). On top of the kernel there are a set of native C/C++ libraries and the Android runtime including the Dalvik virtual machine, which is the Android specific implementation of the Java virtual machine. The properties of the native libraries are used in the Java core libraries that form the Application Framework. The final layer contains the Java-based applications that are created using the Application Framework layer. The Android platform also includes some key applications such as Phone, Messaging, Gallery and Camera [5].

We have experimented using MBT with Android earlier, as presented in [6]. The differences here are that in the earlier work the test automation was based on a commercial tool, and because of confidentiality restrictions concerning the SUT, we could not present details about the models or the actual results. Compared to the existing body of knowledge in MBT, the new contributions of this paper are the following: we present an Android test automation solution developed by ourselves and we are able to openly discuss about the bugs found and other results of the case. Since all the models, tools and the BBC News Widget are freely available, the reader can reproduce the results. The model-based testing tools used in this case (TEMA tools) can be obtained from [7].

The rest of the paper is structured as follows. Section II discusses the background, including description on graphical user interface (GUI) test automation in general and particularly in Android. The section also describes our keyword-based test automation tool. Section III introduces how MBT can be performed with the TEMA tools. In Section IV, we present the case study. Section V discusses related work and Section VI concludes the paper with final remarks.

II. GUI Test Automation With Android

To be able to execute automatically generated tests on Android devices, we need a test automation solution. This section discusses some of the common challenges of automating GUI testing, how Android supports testing of its

978-0-7695-4342-0/11 $26.00 © 2011 IEEE
DOI 10.1109/ICST.2011.11

GUI applications and what techniques we used in our tool.

A. GUI Test Automation

In general, graphical user interface test automation is a complex task. Automation needs to mimic how humans interact with the GUI by using the widgets and by understanding the data presented in the GUI. Injecting user inputs, such as pressing hardware keys or tapping touch screen coordinates can often be achieved relatively easily, but interpreting information from complex graphical representations can sometimes be problematic [8], [9].

Verifying a GUI can be divided into two approaches: using bitmap comparison, or using an Application Programming Interface (API). If interpreting textual content from the GUI, a third approach, optical character recognition (OCR) can be used. While bitmap comparison is easy to implement, it causes excessive maintenance problems: a small change in the GUI means that all bitmaps used in the comparison need to be replaced. Verifying the GUI through an API can be much more maintainable, but unfortunately such APIs are not always available. Another problem with API access is that the information gathered from the API might be slightly different than when seen through the GUI (e.g. more data is available from the API) and problems with UI rendering are not detected. Using OCR falls between the two techniques: while being more maintainable than direct bitmap comparison, OCR can be somewhat inaccurate and slow to execute.

A convenient approach in GUI test design is the keyword-driven testing method [10], [11]. In keyword-driven testing low-level UI operations are abstracted using keywords. Keywords typically depict basic user events, such as pressing hardware keys, tapping or dragging objects on the screen, or selecting an item from a menu. Some keywords verify the state of the SUT, such as examining if a desired text or GUI object is found from the screen. When tests are executed, the keywords are transformed into concrete UI operations in the system under test (SUT) through API calls or other methods with a suitable test automation tool. The benefit of keywords is that they separate the test automation logic from the test design and thus make the tests more maintainable. Moreover, if designed at a suitable abstraction level, keywords can allow small changes in the GUI, such as rearrangement (or switching between landscape and portrait modes on a phone) without causing changes to the tests. Writing and reading a test described with keywords is also easier in comparison with low-level scripts. Keywords can be divided into different hierarchical levels so that the implementation of high-level keywords is described with low level keywords, etc.

B. Structure of Android Applications

The main building blocks of Android applications are activities, services, content providers and broadcast receivers. Activities define individual GUI views, services can be used to handle background processing, broadcast receivers listen to messages initiated by the system or other applications (e.g. battery low, call) and content providers are used for sharing application data between multiple applications. One of the important ideas in Android is that the application elements can be shared between applications. For example, an activity presented by a browser can be used in other applications to present some data in a browser view. This approach blurs the boundaries between individual applications and could possibly introduce new concurrency related problems that are hard to find with conventional testing tools.

The Android GUI is primarily controlled using a touch screen, similar to most modern high-end smartphone devices. In addition, some devices include trackballs, navigational keys and full keyboards. The main view of the Android OS is the customizable home screen, which can contain widgets (e.g. clock, calendar, and search bar) and shortcuts to applications. A part of many views is the status bar, which appears at the top of the screen and shows icons that indicate various events and status, such as new mail in the inbox, battery charge level and signal strength.

C. GUI Testing in Android

The Android software development kit (SDK) contains several GUI testing tools, but due to the security features of Android, creating system level automation is currently limited. The main GUI testing utility in Android is the Android instrumentation framework. With instrumentation, separate testing applications that have access to the application context can be launched in the same process as the application under test. However, instrumentation requires access to the source code of the application under test. Moreover, instrumentation can only control applications started in the same process with the testing application. Therefore, instrumentation is not suited to system-wide GUI testing that includes testing of multiple applications. The instrumentation framework is further abstracted in the Robotium [12] tool which allows black-box testing an APK-packaged application without the source. However, the limitation to one application at a time still exists.

In addition to instrumentation, Android SDK contains a monkey testing tool (Monkey) [13] for sending pseudo-random user events, such as key presses and screen taps, to the device. The Monkey tool is useful in testing the stability of applications and thus finding, e.g. bugs that cause crashes. However, dumb monkey testing tools, such as the Monkey, do not detect other kinds of defective behavior, in contrast to smart monkey testing tools [14]. In later versions of the platform (since 1.6) The Monkey included a network interface that can be used to receive individual GUI events from clients.

In the platform version 1.6, Android introduced new accessibility features. The goal of accessibility technologies

is to help users with disabilities, such as poor eyesight, in using GUI applications. Accessibility services can be used, for example, in creating applications that read the contents of the GUI aloud. In addition to its main use, accessibility APIs can fit well to test automation [15]. For example in Linux desktops, accessibility technologies are used in multiple test automation solutions (LDTP, Dogtail, Strongwind). Accessibility has also been used in our previous case studies with Mobile Linux to enable test automation [16].

The accessibility features of Android allow developers to create accessibility services that work in the background of the device and receive notifications of various GUI events, such as when the state of the window changes or some GUI widget is focused. These events provide some information about the widget where the event originated, such as the type of widget and its text content. However, Android accessibility currently does not allow users to list the full contents of the screen, as is the case, for example, in the Linux GTK environment. This omission clearly limits the usefulness of accessibility, because now the only way to get a picture of the current view is to navigate through it using, e.g. the trackball. A GUI view might also contain certain widgets or labels that are not focusable, and are thus inaccessible.

Hierarchy Viewer is an application in the Android SDK that enables application developers to inspect the layout of the Android GUI. Hierarchy Viewer communicates with an Android device or emulator using the Android Debug Bridge (adb) that allows users to manage devices from a PC. Hierarchy Viewer lists the layout of the GUI in a tree representation and provides some information, such as the ID, type, text content, location and size of all the GUI widgets in the screen. The downside of the Hierarchy Viewer is that due to security reasons it works only with the emulator and other devices that have platform builds with security disabled and thus cannot be used with most devices on the market.

D. Keyword-Driven Test Automation for Android

As stated earlier, to enable GUI test automation, we need to be able to inject GUI events and to verify the state of the GUI. In the Android case, the network interface of the Monkey application solved the first objective. Finding a suitable way to verify the GUI was harder, though. We used OCR capabilities from Microsoft Office in some earlier cases, but we wanted to avoid that, because its accuracy is not flawless [6] and it would limit the test automation to Windows environments. The second option was the accessibility feature, but its limitations in inspecting the GUI context made its use impractical. Finally, we examined the Hierarchy Viewer application.

Internally, Hierarchy Viewer connects to a service called the Window service running in the device to receive a raw data dump of the on-screen GUI-objects. The Window service can be communicated with a socket connection using the port forwarding features of adb. The Window service was selected as the method of verifying the GUI contents in our test automation. We created a parser that reads the raw data from the service and creates an object hierarchy from that.

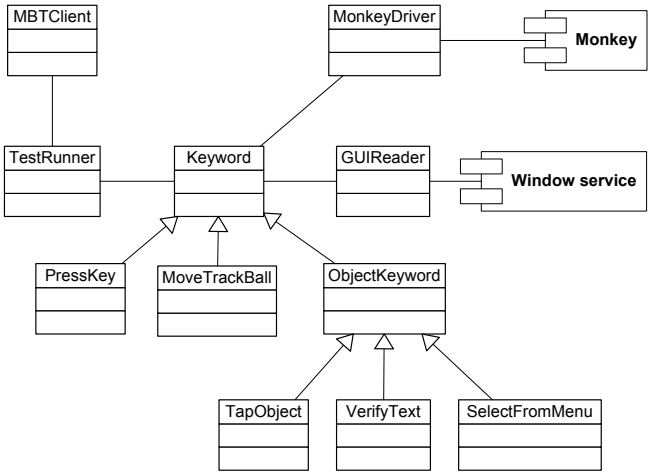


Figure 1: Architecture of the keyword-based test automation tool for Android.

The approach using the Monkey and the Window service in a keyword-driven test automation was implemented with the Python programming language. The architecture of the tool is shown in Figure 1. The main components are the *GUIReader* that communicates with the Window service and the *MonkeyDriver* that connects to the Monkey network interface. The abstract superclass, *Keyword*, introduces some common methods for all keywords. *ObjectKeyword* is a superclass for all keywords that are related to UI objects, and it contains methods for searching specific objects from the UI hierarchy. Actual executable keywords are extended either from *Keyword* or *ObjectKeyword*. The Figure presents only a small sample of 33 different keywords that were defined. The *MBTClient* class implements the client side of the *TEMA MBT test engine*, and runs tests generated from the models. In addition to running model-based tests, the test automation enables keywords to be executed from an interactive prompt or from a file.

The keywords used in the tool are formed from the keyword name and a list of parameters. Many keywords are directed to a specific GUI object, so the object must be defined in the parameters. To define the target object, the keywords accept references that use either the object ID or the text content of the object. The hierarchical structure of the GUI can also be used in the references, by defining the parents of the searched object. Sometimes, using the hierarchy can be the only way to uniquely identify an object. As an example of keywords, Figure 3 shows the headline view of the BBC News Widget application, where

the 'Feeds' tab could be tapped with any of the following keywords:

```
1. TapObject 'Feeds'
2. TapObject id/title
3. TapObject id/tabs:::'Feeds'
4. TapCoordinate 80,75
```

The first keyword searches for an object that contains the text 'Feeds' and taps it. The second keyword tries to find an object with an ID 'id/title'. In this case using the ID is not very practical, because the 'Most Popular' tab in the same view has the same ID. If multiple objects that match the reference are found, the first one found is selected. The third keyword searches for an object with a text 'Feeds' that is found under the object with ID 'id/tabs' (a layout containing the tabs) in the object hierarchy. The advantage in the third reference is that if the other parts of the GUI (e.g. the news items) contain the same text, the correct object is still safely selected. The final keyword simply taps the coordinates where the object appears on the screen. Although quick to execute (no need to either read or search the GUI), the fourth keyword has problems related to maintenance. If the tab order, location or screen size changes, or if new tabs are added (which occurred with the latest version of the application), the coordinate might be incorrect. In addition, the keywords from one to three all verify the GUI. If the application is in the wrong state and the object is not found, the conflict is noticed and the test fails. The fourth keyword will always execute successfully unless the device has not crashed completely and will continue the test in a wrong state, potentially doing something erroneous.

The use of the Window service limits the use of our test automation tool to the emulator and devices where security is disabled, as was discussed earlier. Platform builds with the security features disabled can only be installed in devices that are hardware-unlocked, such as Android development phones. In our testing, we only used the emulator, which is adequate for testing functionality. However, some features, such as performance and user experience must be checked on an actual device. Another problem with the tool is that retrieving the GUI information from the emulator/device takes some time (about one to five seconds in our experience) and thus slows down keywords that frequently need updated GUI information. This problem also makes it very hard to verify GUIs that change rapidly.

III. Model-Based Testing with TEMA Tools

TEMA Tools is a set of model-based testing tools that was originally developed for the testing of S60 GUI applications, but has also been experimented with in Linux (Gnome), Android, Java Swing and Qt (running in a Maemo device) environments. The toolset contains tools for different phases of MBT: test modeling, test design, test generation and test debugging. The architecture of the TEMA tools is shown in Figure 2. The test modeling part contains a tool for model design and utilities e.g. for model debugging. The test design includes a Web GUI for designing objectives for tests. The test generation presents a number of algorithms that use the models and the test objectives to produce the actual tests. Test debugging contains tools for interpreting unsuccessful test runs. Finally, the keyword execution performs the test execution on SUTs and is the only platform-dependent part of the toolset. In the case study, the keyword execution is covered with the Android test automation tool presented in this paper. The model library is the only part that needs updates when new application models are created.

TEMA models are state machines, specifically labeled state transition systems (LSTS). LSTS contain states, transitions between states, actions that are attached to transitions and labels that are attached to states. Modeling a real-size application in a single state machine would result in models that are hard to understand and maintain. Therefore, TEMA models can be divided into smaller model components, and the full test model is automatically combined from the components in a process called *the parallel composition*. The parallel composition links certain actions of the models together, so that they are executed synchronously. The parallel composition method TEMA uses is based on a rule set which explicitly defines which action names are synchronized. The parallel composition method was developed in [17], as a generalization of CSP [18]. We skip the formal definition of LSTS and parallel composition here due to space restrictions; interested readers are referred to [19], [20].

Each model component is divided into two levels that are described by two separate state machines: an action machine and a refinement machine. The action machine describes the high-level functionality using action words and state verifications. An action word that is modeled as an action in a transition, describes a small use case in an application, such as saving a file. A state verification, modeled as a label in a state, describes the state of the SUT and is used to verify that it corresponds to the state of the model. The implementation of the action words and state verifications are described in refinement machines using keywords. The key advantage in the two-layer model architecture is that it makes the action machine level models reusable. Normally, only the refinement machines must be reimplemented when the model is used in a new domain (e.g. in another smartphone model).

TEMA models support two types of test data: localization tables and data tables. Localization tables are used to separate localized strings from the refinement machines, where the string is referred to with a symbolic name. The correct string is chosen during test generation based on the selected locale. Data tables can be used to store complex structured data, such as inputs given to the application under test. In state machines, data tables are accessed using data

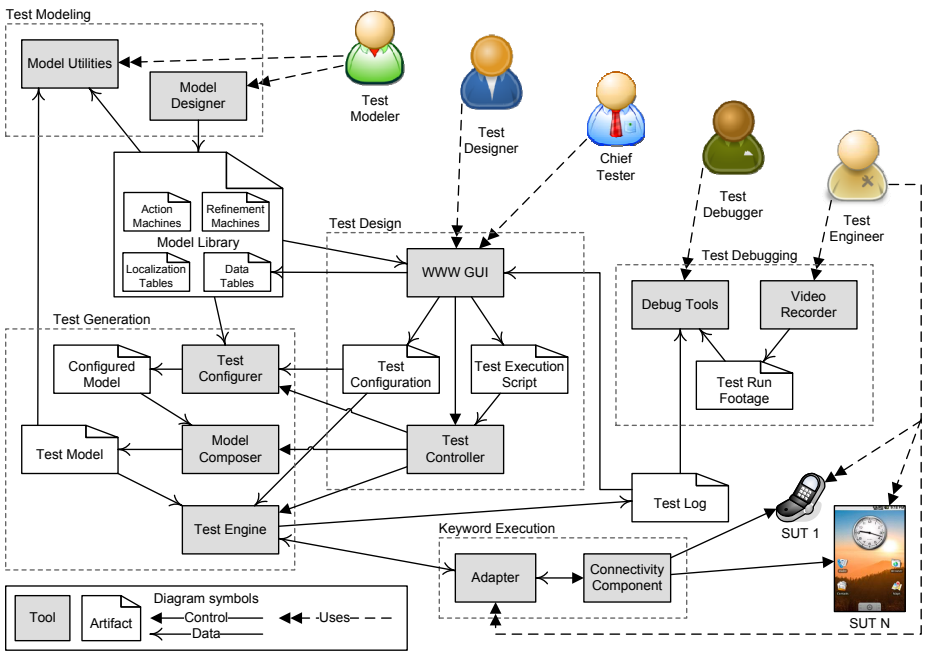


Figure 2: Architecture of TEMA tools.

statements, which are actually Python code embedded to the transitions. The data statements allow modelers to use the full Python functionality.

There are two ways of performing test generation: online and offline. In the online approach, the test generator and SUT work together in small steps. After a test step (a keyword) has been generated from the model, the step is executed in the SUT, and a Boolean value describing the result of the step is returned for the test generator. Therefore, the online approach allows test generation to adapt to different responses of SUT and makes the testing of non-deterministic features possible. The online approach also enables long-period random tests that have no other specific goals but to find faults from the system. In contrast, the offline approach generates full test cases from the model first and executes them later as a whole. The TEMA test generation uses the online approach.

IV. Case Study: BBC News Widget

In this case study, we tested a popular BBC news reader application, named BBC News Widget [21], which is available free of charge from the Android Market. The application is an RSS reader designed specifically for the BBC news feeds. At the time of writing, the application had about 849,000 downloads and 488,000 active installations. The application had been previously manually tested.

BBC News Widget contains the main application for listing different news feeds and reading news items. In addition, the application includes a widget that can be added to the home screen for showing the latest headlines from a selected feed, as shown in Figure 3. The application can be launched either by tapping a widget, or by selecting the main application from e.g. the application menu. Depending on how the application was launched, either a default feed or the widget's feed is listed in the main view.

The main view of the application lists the news titles and descriptions of a selected feed. The feed can be configured for each widget added to the home screen and to the default feed which is shown if the main application is launched. From the main view, the user can select a headline for

Figure 3: BBC News widgets on the home screen and the main headline view.

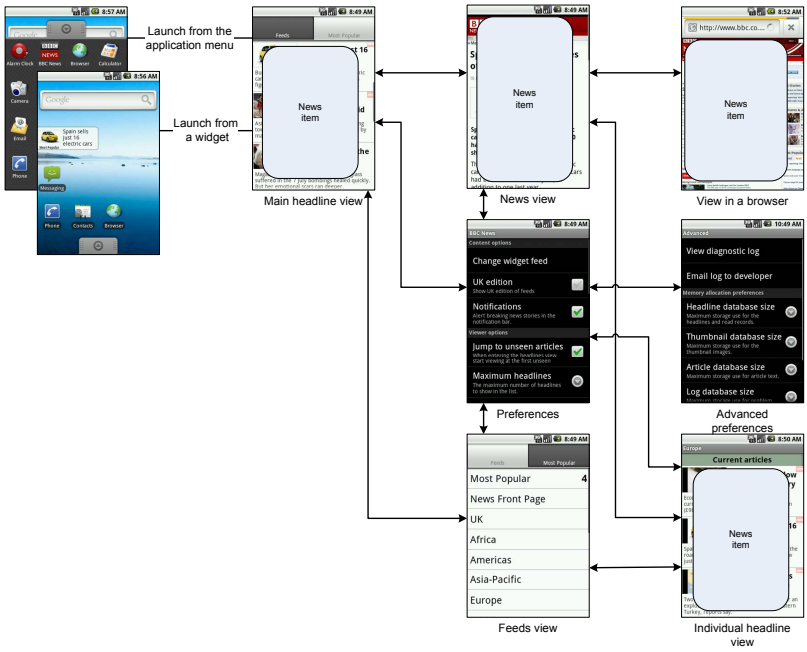


Figure 4: BBC News Widget main views.

reading the full news item, check other BBC feeds or go to the preferences view for changing the application settings. The main views and the transitions between the views can be seen in Figure 4.

A. *Modeling BBC News Widget*

The BBC News Widget contains seven individual views, as shown in Figure 4. A logical way to model a GUI application with TEMA is to create a separate model component for each individual view. The resulting model therefore contains seven model components for the views. As explained earlier, one model component in TEMA contains an action machine and a refinement machine. In addition to the views, the model contains one model component that manages the widgets in the home screen (addition and removal) and the startup functionality of the application (launching and closing the application) and one model component that stores the information about the existence of the widgets. In total, the model of BBC News Widget is composed from 16 separate state machines.

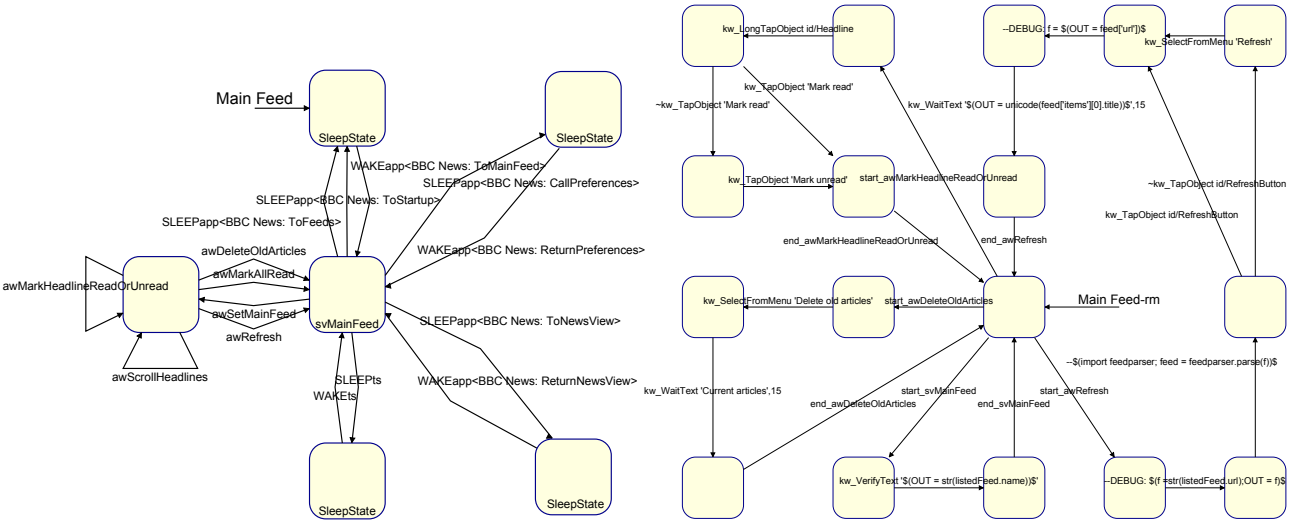
Certain information, such as whether the application was started from a widget or from the main application and what the current feed is, is stored to Python variables using the Python statements in the transitions. In some situations using this approach to store state information is convenient, as storing the same information to states would result in massive models. The Python data can be evaluated and used to branch the execution in the models.

The views *Main headline view* and *Individual headline view* are very similar in their appearance and operations that can be performed. The only difference is that the former view contains tabs for moving between the feeds and the headline view. This situation was modeled by creating a copy of the the *Main headline view* action machine for the *Individual headline view* and removing the actions that are not available. The same refinement machine was linked for both action machines, removing the need to copy. Although the refinement contains some extra functionality for the other view, they will never be accessed when there is no correspondent action in the action machine.

As an example of the models, the action machine and the refinement machine for the *Main headline view* are shown in Figure 5[1]. The actions beginning with ”SLEEPapp” are used with the parallel composition to switch between action machines by executing the action synchronously with an action that begins with ”WAKEapp”. For example, the ”SLEEPapp <BBC News: ToFeeds >” action is synchronized with the corresponding ”WAKEapp” that is found from the action machine of the feeds view. ”SLEEPts” and ”WAKEts” transitions mark states where the foreground application can be changed to another one. Actions beginning with ”aw” are the action words, and their implementations are given in the refinement machine between the ”start_aw” and ”end_aw” actions. The action machine also contains a state verification ”svMainFeed”, which is implemented similarly to action words.

The fact that the application requires an Internet con-

[1]Some actions have been left out from the refinement machine for reasons of space.



(a) Action machine: main headline view.

(b) Refinement machine: main headline view

Figure 5: State machines for the main headline view model component. The initial state is marked with a transition without a starting state.

nection made some things a somewhat hard to model. For example, how should we verify that the news items the application shows are correct? Another factor is how the models can adapt to the varying network delays. To address the former problem, we utilized python statements in the models. A Python RSS library was used to retrieve news headlines that were then compared with the news items in the application to verify that the news items were correctly updated. For the latter issue, we simply defined long enough time limits in the models where a network operation was executed. A more advanced model could, of course, even consider situations where the connection is lost, and adapt.

B. Running Tests

To make sure the models did not contain errors, we began testing by running tests that covered all the actions of the models. Once completed, we started running random tests that wander through the model choosing transitions arbitrarily. The best thing in the random tests is that their design with the TEMA Web GUI takes practically no time at all. Random tests can also cause very complex sequences that strictly designed tests would seldom cover. In addition to random testing, we experimented with interaction tests using the Contacts, Gallery, Camera and BBC News Widget applications. In the interaction tests the test generator changes the foreground application often [22].

TEMA supports other test generation algorithms as well. Use cases can be described using the action words and logical operations AND, THEN and OR. In these cases an algorithm that attempts to find the shortest path that covers the actions defined in the use case can be used. In addition to use cases, tests whose goal is to cover all actions, transitions, states or their combinations can be defined. In a real-life testing situation passing such tests could be used as a requirement before accepting a software version to be released.

A practical problem when running tests with the emulator was that the network connection was always lost after the emulator had been used for a couple of hours. After the connection is lost, the emulator must be restarted to get the connection back. Another issue in running MBT tests in general is to avoid bugs that have already been discovered. The topic has been discussed in detail in [23].

C. Testing Results

All in all, 14 bugs or instances of inconsistent behavior were found in the application. The bugs are listed in Table I. Eight of the bugs found were discovered during the modeling process and six during test execution. In addition, a few other smaller inconsistencies concerning e.g. the GUI appearance were discovered during modeling and were reported. Two of the found bugs caused the application to crash, although one of them (#5) is somewhat hard to reproduce. The other (#13) occurs when the user changes the update frequency of the news feeds to zero, which causes so much load to the device that it becomes unresponsive and eventually crashes the application. The rest of the bugs were not major, but rather small annoyances. Some of the bugs were related to where the application was launched (e.g. the main application options were shown although the application was started from a widget). The detailed bug reports are available for inspection in [7] (see the Android case study). At the time

Table I: Bugs found in the case study (* caused a crash)

#	Description	Found
1	A button gets stuck on a disabled state	modeling
2	When the main feed is changed, the feed header is not updated	
3	OK and Cancel buttons do not save/restore values in preferences	
4	When the widget feed is changed, the feed view is not updated	
5*	Application sometimes crashes when refreshing feeds after modifying the preferences.	
6	Max headline value zero not handled properly in preferences	
7	Inconsistency when switching between multiple widget feeds	
8	News item count not updated properly in the feeds view	
9	Preferences shows "Select default feed" although the application was started from a widget.	execution
10	Extra whitespaces in news titles	
11	Refresh inconsistency when marking news read	
12	Widget forgets its state when set to the foreground	
13*	Update frequency value zero not handled properly	
14	Widget launches the main app if the widget feed is unavailable	

of writing, the version number of the BBC News Widget is 2.5.4, and at least two of the listed bugs are fixed, including one that caused a crash (#5).

In our earlier case studies with S60, model-based testing was quite effective in finding concurrency related bugs from applications that share common resources [20]. In this case study BBC News Widget was executed concurrently with some of the base Android applications (Contacts, Gallery, Camera). However, we did not find any concurrency related bugs in this case, possibly because these applications mainly do not use any shared resources with BBC News Widget. However, these tests did find one inconsistency when switching BBC News Widget to background and back to foreground (#12). When the application was launched using the application menu, switching the application to the foreground always returned the application to the exact state where it was when set to background. However, when the application was started from a widget, switching it back to foreground always set it in the main headlines view regardless of what its state had been before.

When modeling an existing application, the modeling process is an effective way to test manually as it requires a careful inspection of the application. In a sense, modeling can be seen as a way of performing exploratory testing. Similar to our previous case studies, most of the discovered bugs were found during modeling. Any of these bugs could have been found with regular manual testing. These results also correspond to the experiences of Chip Groder [9, pp. 517-536], where an existing GUI was used as a reference when designing test automation scripts. Concerning the bugs found during automatic test generation and execution, none required especially long sequences to reproduce. Therefore, most of these bugs could also have been found with conventional manual testing methods. However, the model accurately found even quite small inconsistencies, which manual testers might have missed.

One of the often mentioned advantages of MBT is its maintainability when new versions of the SUT are released. Instead of updating a set of test cases, only the models need to be updated. During the case study, three updates to the application and one update to the platform were released. Updates to the application did not introduce any changes to the existing features, but some new ones were added. After the models were updated, we were immediately able to run tests that used both the new features and the old ones.

The Android platform update from version 2.1 to 2.2 introduced some changes to the GUI e.g. in the home screen. In some parts of the model, this required small updates. Eventually, the platform dependency was moved as much as possible to the test automation tool by using suitable keyword abstraction to hide the details of the platform version. In practice, the tool checks the version of the platform and then chooses the correct actions. This makes models easier to maintain, but obviously complicates the tool. However, it is usually easier to maintain the implementation of keywords than making changes to multiple models. The new platform also introduced some updates to the base applications. For example, in the Contacts application, the update removed some of the unique IDs of the GUI objects, which forced us to make such updates to the models that actually made them less maintainable. After the update, the Camera application stopped working entirely in the emulator, and could not be tested.

The modeling of BBC News Widget took a few days (including subsequent updates) from a modeler that was still learning the modeling details used in TEMA and had only limited experience of modeling. The work effort needed is largely in line with our previous experiences with the S60 applications [20].

Based on 240 separate test runs where the BBC News Widget was included, in total 27 000 action words and 50 000 keywords were executed. Altogether, the test runs lasted about 115 hours with the longest runs lasting over three hours and the average duration being about 30 minutes. The average execution time of one keyword in these runs was about eight seconds including the delays between keywords.

V. Related Work

Although there are some previous studies in model-based testing of GUI applications, using the online approach in this domain or MBT in general seems to be rare. In [24] the authors use NModel from Microsoft to model the C#

based GUI of a mobile medical assistant device used by the military. The resulting model program created with NModel can be visually represented as a Finite State Machine (FSM) similar to the ones created with TEMA. However, the models do not contain the two-level architecture of TEMA models. Moreover, test generation with NModel follows the offline approach.

In [25], Memon summarizes an approach that uses event-flow models for testing GUIs. The model can be created semi-automatically using a reverse-engineering tool that automatically executes the GUI and extracts all widgets and their properties from the GUI. The reverse engineering tool works on Java Swing and Microsoft Windows GUIs. While such a tool may significantly reduce the amount of manual work in the model creation, it removes the valuable inspection that the modeler performs during the manual modeling process. Nevertheless, the approach seems practical when creating a regression testing suite for applications that have been previously tested using other methods.

The above Memon's approach is used in [26] to test open source software with GUIs that is developed in a distributed community working over the Internet. The paper presents three different loops that are used during different development cycles to produce MBT tests for the GUI. The first loop, executed during each new commit, runs a fast set of tests that can detect crashes in the SUT automatically. The second loop, executed daily/nightly, will use the previous working version of the GUI as a baseline to generate smoke tests that search also for other GUI problems than crashes, by comparing state information gathered from the older version. The final loop is executed before any major release, and contains manually designed test cases. The two first loops of the approach are automatic: reverse-engineering is used to create the model. The last approach requires manual effort in designing the test cases and defining test oracles for the SUT. A respectable number of crashes (uncaught exceptions in Java GUIs) were found using the approach in a number of case studies, but the studies do not show any results of finding other functional errors.

TEMA does not contain tools for reverse-engineering models automatically, but there should be no technical limitations (other than caused by the SUT GUI environment) why such could not be achieved. This could make it easier to adopt MBT to an ongoing project where models have not been previously used. However, we believe that manual modeling has many benefits that are lost by trying to automate the process.

VI. Discussion

From the reasearch perspective the results of MBT in this case study were quite satisfying. The modeling proved to be an effective way to find problems, which is consistent with previous studies. Some new issues in the application under test were also found during execution, but more importantly, we now have an easily maintainable way to do regression testing for new versions released. Setting up a test that, for example, executes all actions in the model can be done in a few minutes.

TEMA toolset worked reasonably well in the case study. The set of keywords was specifically designed for the needs of the modeled applications, but the set should cover the basic needs of any Android application. When needed, new keywords are easy to add to the test automation tool by extending the base keyword classes. One notable disadvantage in the tool is its reliance on sections of the Android platform that are not part of the public API. For example, it is possible that the data dump format received from the Window service will change in future platform releases, which would cause maintenance problems. As a future work, the tool could include a scripting language used with the keywords or it could be converted to a Robot Framework [27] library.

The models of BBC News Widget cover most of its basic functionality. However, there are some extra features that could be added to the models. These could include handling breaking news alerts in the status bar, multiple widgets in the home screen and creating mutant feeds with uncommon contents (long headlines, etc). Another interesting feature to model in the application is its integration with a notes application (3banana Notes), that can be used, e.g. to save interesting news items.

One major motivation for setting up this case study was to identify obstacles to the industrial deployment of the TEMA toolset. At the end of the case study, Julian Harty provided following feedback from industrial perspective: *TEMA for Android has a strong pedigree and has the potential to find useful issues, unlike many test automation projects I have used. As the industrial partner I (Julian Harty) picked the BBC News Widget to assess TEMA's potential because the Widget was relatively simple, popular, and had been developed by a competent and experienced Android software engineer (Jim Blackler) who was willing to work with the project. The bugs found by the modeling and test execution were new and I don't expect they would have been found through manual or other automated testing. However, because they were in rarely exercised paths of the code they were not critical to fix. The TEMA tools currently require significant effort to learn how to use them, more effort than many commercial developers may be willing to invest. I recommend further investment in the project to make the tool easier to use, and would like to see some of the current claims tested, e.g. reuse of high-level models across various phone platforms. I would also like to see it used to test a variety of commercial software to ascertain the quality and utility of the bugs found by the tools.*

This sets targets for future research to make the test modeling easier. We should also set up new case studies to assess test model reusability.

ACKNOWLEDGMENT

Partial 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. We also appreciate help from the rest of the TEMA team and Jim Blackler for making the case study possible.

REFERENCES

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

[2] 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.

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

[4] The NPD Group, "Motorola, HTC drive Android to Smartphone OS lead in the U.S." http://www.npd.com/press/releases/press_100804.html. Cited Oct. 2010.

[5] Android developers homepage, http://developer.android.com/guide/basics/what-is-android.html. Cited Oct. 2010.

[6] 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., to appear.

[7] TEMA Toolset homepage, http://tema.cs.tut.fi. Cited Dec. 2010.

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

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

[10] Hans Buwalda, "Key Success Factors for Keyword Driven Testing," Available at http://www.logigear.com/resource-center/software-testing-articles-by-logigear-staff/389--key-success-factors-for-keyword-driven-testing.html. Cited Oct. 2010.

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

[12] Robotium homepage, http://code.google.com/p/robotium/. Cited Oct. 2010.

[13] UI/Application Exerciser Monkey, http://developer.android.com/guide/developing/tools/monkey.html. Cited Oct. 2010.

[14] N. Nyman, "Using monkey test tools," *Software Testing and Quality Engineering magazine*, vol. 29, no. 2, pp. 18–21, 2000.

[15] M. Grechanik, Q. Xie, and C. Fu, "Creating GUI testing tools using accessibility technologies," in *Proc. IEEE International Conference on Software Testing, Verification, and Validation Workshops*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 243–250.

[16] 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*, J. Zander, I. Schieferdecker, and P. J. Mosterman, Eds. CRC Press, May 2011, to appear.

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

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

[19] A. Jääskeläinen, A. Kervinen, and M. Katara, "Creating a test model library for GUI testing of smartphone applications," in *Proc. QSIC 2008 (short paper)*. IEEE CS, 2008, pp. 276–282.

[20] 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. Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society, 2009, pp. 112–122, companion volume.

[21] BBC News Widget, http://jimblackler.net/blog/?p=124. Cited Oct. 2010.

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

[23] A. Jääskeläinen, "Filtering test models to support incremental testing," in *Proc. TAIC PART 2010*, ser. LNCS, vol. 6303. Springer, 2010, pp. 72–87.

[24] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, and P. L. Jones, "Model-Based Testing of GUI-Driven Applications," in *SEUS*, ser. Lecture Notes in Computer Science, S. Lee and P. Narasimhan, Eds., vol. 5860. Springer, 2009, pp. 203–214.

[25] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Softw. Test., Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, 2007.

[26] Q. Xie and A. M. Memon, "Model-based testing of community-driven open-source GUI applications," in *Proc. 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 145–154.

[27] Robot Framework homepage, http://code.google.com/p/robotframework/. Cited Oct. 2010.