TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Mikko Satama

Event Capturing Tool for Model-Based GUI Test Automation

Master of Science Thesis

Subject Approved by Department Council
8th February 2006

Supervisors: Professor Ilkka Haikala
Senior Researcher Mika Katara

Acknowledgements

This thesis work is conducted at Tampere University of Technology (TUT) commissioned by the Institute of Software Systems as a part of TEMA research project. TEMA is a joint project consisting of the contributions of the Institute of Software Systems at TUT and five business partners: Nokia, Conformiq Software, F-Secure, Plenware Group and Mercury Interactive. TEMA is funded by the Finnish Funding Agency for Technology and Innovation (TEKES). I wish to thank these participants for financing the TEMA project and thus making this thesis possible.

My supervisors and colleagues at the Institute of Software Systems have been indispensable to me. Consequently, I wish to give them special appreciation. I would like to thank my supervisor Ilkka Haikala for expert criticism and for the interest he has shown in my work. Similarly, I wish to thank my instructor Mika Katara *very warmly* for all the encouragement, support, advice and numerous valuable comments that have made this thesis possible. I also wish to thank my colleagues Mika Maunumaa, Antti Kervinen and Antti Jääskeläinen for all the expert advice and useful tips for my software development. Very special thanks go to my parents Riitta and Jorma Satama for all the love, support and patient attitude that they have shown me.

Finally, I would like to give the greatest thanks of all to my Lord and Saviour, **Jesus Christ**, who gave everything for me by dying in my place. He answered my prayers, helped me enormously in my work – sometimes supernaturally – and remained on my side in the loneliest moments when all others were gone.

"Praise the Lord, my soul, and do not forget all the good things he has done for you!" (Psalm 103:2)

Mikko Satama

Tampere, Finland
28th August 2006

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology
Institute of Software Systems

SATAMA, MIKKO: Event Capturing Tool for Model-Based GUI Test Automation
Master of Science Thesis
51 pages, 4 enclosure pages
September 2006

Supervisors: Professor Ilkka Haikala (TUT)
Senior Researcher Mika Katara (TUT)
Funding: Tampere University of Technology (TUT)
Keywords: Software testing, test automation, model-based testing

Model-based testing (MBT) provides a notable improvement to conventional scripted testing by automating the creation of test cases. However, model-based methods are generally not well-embraced in large-scale industrial environments. The lack of well-developed and easy-to-use tools is a major problem that hinders the utilisation of MBT. Especially the tools for creating test models have been insufficient in many efforts of deploying the model-based methodology in industrial contexts. A crucial question is how an average tester without expert knowledge could construct a test model capable of finding defects effectively.

The aim of this thesis is to provide an answer to this question by conducting constructive research. By developing an event capturing tool for graphical user interface (GUI) test automation, a proposal is introduced for easing the model construction performed by average testers that do not necessarily possess programming or software developing skills. Likewise, further knowledge is sought by conducting a test modelling case study with the tool in an industrial context.

The work is based on a domain-specific approach to the model-based GUI testing that should be easier to adopt than more generic solutions. The method is premised on the basis of *keywords* and *action words* that are considered as best practice in conventional GUI test automation. Action words model user behaviour at a high level of abstraction while keywords correspond to GUI navigation. The idea is to capture GUI events directly and produce keywords from them automatically. These keyword sequences are then developed into action words and constructed as labelled transition system models.

The results of this study suggest that test modelling without expert knowledge is conceivable and can be eased by the development of a proper tool. In industrial use it is especially significant to bring theoretical knowledge into productive action. The event capturing tool appears to function well for this aim and should ease the adoption of MBT.

Tiivistelmä

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan osasto
Ohjelmistotekniikan laitos

SATAMA, MIKKO: Käyttöliittymätapahtumien kaappaustyökalu mallipohjaiseen graafisen käyttöliittymän testausautomaatioon
Diplomityö, 51 sivua, 4 sivua liitteitä
Syyskuu 2006

Ohjaajat: Professori Ilkka Haikala (TTY)
Vanhempi tutkija Mika Katara (TTY)
Rahoitus: Tampereen teknillinen yliopisto (TTY)
Avainsanat: Ohjelmistojen testaus, testausautomaatio, mallipohjainen testaus

Mallipohjainen testaus merkitsee huomattavaa parannusta tavanomaiseen skripti-testaukseen automatisoimalla testitapausten luonnin. Mallipohjaisia menetelmiä ei ole kuitenkaan otettu kovin hyvin vastaan teollisuudessa. Kehittyneiden ja helppokäyttöisten työkalujen puute on merkittävä ongelma, joka estää mallipohjaisen testauksen käyttöönottoa. Varsinkin testimallien luontiin käytettävät työkalut ovat olleet riittämättömiä monissa tapauksissa, joissa on yritetty ottaa mallipohjaista testausta teolliseen käyttöön. Olennainen kysymys on, kuinka tavanomainen testaaja, jolla ei ole erityisosaamista, voi rakentaa virheiden tehokkaaseen löytämiseen kykenevän testimallin.

Diplomityön tarkoituksena on pyrkiä löytämään vastaus tähän kysymykseen konstruktiivisen tutkimuksen muodossa. Työssä kehitetään käyttöliittymätapahtumien kaappaustyökalu graafisen käyttöliittymän testausautomaatioon. Tällä tavoin pyritään helpottamaan mallien rakentamista erityisesti tavallisten testaajien kannalta, joilla ei välttämättä ole ohjelmointitaitoja. Samoin pyritään hankkimaan lisätietoa suorittamalla konkreettinen testimallinnustapaus työkalulla teollisessa ympäristössä.

Työ perustuu sovellusaluekohtaiseen lähestymistapaan graafisen käyttöliittymän malli-pohjaiseen testaukseen. Sen pitäisi olla helpompi omaksua kuin yleisemmät ratkaisut. Menetelmä perustuu *avainsanoille* ja *toimisanoille*, joita pidetään parhaana käytäntönä perinteisessä testausautomaatiossa. Toimisanat mallintavat käyttäjän toimia korkealla abstraktiotasolla, ja avainsanat vastaavat käyttöliittymässä navigointia. Ideana on kaapata käyttöliittymätapahtumia suoraan ja tulkita ne avainsanoiksi automaattisesti. Näin syntyvät avainsanajonot kehitetään edelleen toimisanoiksi, joista koostetaan testimallit.

Diplomityön tulokset osoittavat, että testimallinnus ilman erityisosaamista on mahdollista ja helpottuu oikeanlaisen työkalun kehittämisen kautta. Teollisessa käytössä on erityisen tärkeää muuttaa teoreettista tietoa tuottavaksi toiminnaksi. Kehitetty työkalu näyttää toimivan tässä mielessä hyvin helpottaen mallipohjaisen testauksen käyttöönottoa.

Table of Contents

Definitions and Abbreviations

API	Application Programming Interface
CSV	Comma Separated Values
DLL	Dynamic Link Library
GUI	Graphical User Interface
IDE	Integrated Development Environment
LSTS	Labelled State Transition System
LTS	Labelled Transition System
MBT	Model-Based Testing
MDD	Model-Driven Development
MFC	Microsoft Foundation Classes
MFTW	Mercury Functional Testing for Wireless
MSLU	Microsoft Layer for Unicode
MVC	Model-View-Controller architecture
QTP	Mercury Quick Test Pro
SUT	System Under Test
TEKES	Teknologian Kehittämiskeskus (Finnish Funding Agency for Technology and Innovation)
TEMA	Test Modelling using Action Words
TTCN	Testing and Test Control Notation
TUT	Tampere University of Technology
TVT	Tampere Verification Tool
UML	Unified Modelling Language

List of Figures

1. Introduction

Model-based testing (MBT) is technologically superior to conventional scripted testing by automating the creation of test cases and has obvious advantages [1]. Nevertheless, there have been problems deploying this methodology in industrial environments. According to Robinson [2] the most common problems in introducing formal testing methods are 1) managerial problems, 2) problems of making easy-to-use tools, and 3) problems in reorganising the work with the tools. In this thesis the focus is on the second problem.

In MBT the test suites are derived from a high-level model that describes the functionality of the system under test (SUT), and not from test scripts as in conventional test automation. Accordingly, MBT requires a radically new way of thinking in testing. This paradigm shift becomes a serious hindrance in industrial deployment unless special attention is given to the ease of introduction [3].

Moreover, introductory approaches of MBT have been used mostly for testing through various kinds of application programming interfaces (API). However, this can be considered a somewhat constricted practice. Another equally important area of application is testing through a graphical user interface (GUI). This, nonetheless, has challenges of its own.

Testing a system through a GUI is definitely one of the most complicated testing manners. Usually this kind of testing is performed by domain experts who are easily able to confirm the client requirements. However, they do not necessarily possess programming skills and they need especially easy-to-use tools to support their work. Compared to testing through an API, testing through a GUI is complex due to the numerous user interface issues that must be taken into account such as the input of user commands and the interpretation of output results.

Difficulties in the utilisation of MBT in GUI test automation can be eased by developing proper conventions and tools. Katara & al. [4] have suggested that a domain-specific approach to model-based GUI testing is easier to adopt than more generic solutions. This approach, which is followed in this thesis, also carries a promise to find defects more efficiently than conventional GUI testing methods.

Although promising proposals for introducing MBT into GUI test automation have been constructed by e.g. Rosaria & Robinson [5] and Katara & al. [4] there are still open questions that need to be answered. Firstly, how could an average tester construct a productive test model that is capable of detecting defects efficiently? Secondly, how to build an easy-to-use tool for model creation purposes in model-based GUI test automation?

The aim of this thesis is to provide an answer to these questions by conducting constructive research. By developing an event capturing tool for GUI test automation, a proposal is introduced for easing the model construction performed by average testers. Moreover, further knowledge is sought by conducting a test modelling case study with the tool in an industrial context.

The event capturing tool, whose development is described in this thesis, is part of a larger model-based test automation tool being developed within the TEMA research project [6]. The larger tool is called *TEMA Tool* and the event capturing tool is called *Recorder* within *TEMA Tool*.

The TEMA research project aims at studying MBT thoroughly in GUI test automation. The project concentrates on developing a domain-specific test modelling language, developing and describing the required tool set, re-defining testing personnel roles for the new model-based approach, and performing the actual testing on-line in the Symbian OS [7] environment. The testing is based on keywords and action words which are seen as best practice in conventional GUI test automation [8].

It has been suggested that the introduction of MBT tools is eased by developing a domain-specific modelling language [4]. In an ideal case this could be done by obtaining test models from design models. However, the problem in this option is that usually the design models do not exist and there are no tools available for such transformation. In practice it is often necessary to implement the test models by designing them by hand from scratch. Test modelling can be eased by capturing GUI events directly and

producing keywords from them automatically. The keyword sequences are then easy to process further into action words. The event capturing tool is intended for these purposes.

A concrete aim of the event capturing tool is test modelling with keywords and action words, and the formation of a test model by adding state information in order to enable loops and branching in the test flow. Accordingly, this tool must not be associated with the first generation capture/replay tools which have proven to be ineffective [9, pp. 103-104]. Replay is not conducted at all and the capture functionality is very different. The event capturing tool records GUI events just like the old capture/replay tools, but instead of producing scripts that are difficult and laborious to maintain, it produces sequences of keywords. These sequences are transformed into higher level abstractions called *Labelled Transition Systems* (LTS) [10] where action words are used as labels of transitions. Action words model user behaviour at a high level of abstraction while the keywords correspond to the GUI navigation. The event capturing tool can be used to define action word implementations by recording keyword sequences.

The results of this thesis include the event capturing tool for GUI test automation, which is part of the larger MBT tool being developed within the TEMA project. The event capturing tool is a proposal for easing the model construction performed by average testers. Furthermore, the results of the test modelling case study show that modelling without expert software knowledge is conceivable and can be eased by the development of a proper tool. In industrial use it is particularly important to bring theoretical knowledge into productive action. The event capturing tool seems to function well for this purpose. Likewise, the results of the thesis show that a properly chosen and developed methodology, and a suitable tool that implements it, improve the ease of use in an industrial case study and thus make the introduction of MBT easier.

The rest of the thesis is structured as follows. In *Chapter 2* the background of the TEMA research project is discussed to provide essential information on the context of the research. *Chapter 3* describes the theoretical background that is necessary for understanding and following this thesis. The software development process of the event capturing tool is presented in *Chapter 4*. The focus is on the tool design and implementation issues. *Chapter 5* describes the test modelling case study for utilising the tool in a real environment. The solutions to the research questions are presented as well. Finally, *Chapter 6* draws conclusions on the work performed.

2. Background

This section describes the essential background information that is necessary for understanding the work of the thesis. The TEMA research project and TEMA Tool are presented to describe the overall environment to which the event capturing tool is adopted. The event capturing tool as part of TEMA Tool is discussed as well. In addition, the testing environment is described briefly.

2.1 TEMA Research Project

This thesis is implemented as part of the TEMA research project (Test Modelling using Action Words) which is funded by the Finnish Funding Agency for Technology and Innovation (TEKES). TEMA is a joint project consisting of the contributions of the Institute of Software Systems at Tampere University of Technology (TUT) and five business partners: Nokia, Conformiq Software, F-Secure, Plenware Group and Mercury Interactive.

The aim of the TEMA research project is to study software test automation in the scope of testing through graphical user interfaces (GUI). The approach is based on the development of necessary methods and tools. By developing a proper method and a tool platform for GUI test automation, the testing process in this context is made easier. The applied methodology is model-based testing (MBT) which denotes that in addition to automating the execution of test cases the generation of test cases is automated as well. The method being developed in the TEMA project will be based on test models which specify the functionality of the system under test (SUT). The test models contain logical and reusable components that are fabricated with the tools developed in earlier projects.

Accordingly, model-based testing in the scope of GUI test automation is studied thoroughly within the project. There are a few areas which are considered especially important to explore. Firstly, the attention is focussed on developing and describing the required tool set which is necessary for further research. Secondly, a domain-specific test modelling language is developed as well for test modelling purposes. Thirdly, testing personnel roles are re-defined for the new model-based approach. The actual testing is performed in a Symbian OS environment by employing keywords and action words [8].

The underlying general methodology of the TEMA project is based on model-driven development (MDD). This denotes not only raising the level of abstraction but also the automatic creation of test cases, and behavioural and functional models of the SUT. A promising approach to MDD is the utilisation of a domain-specific modelling language. In theory UML could also be used. However, it is considered too generic and it requires knowledge that is not necessary in the utilisation of MDD in testing. UML is originally developed for coding purposes, not for testing purposes, and system testers are not necessarily fluent with it or with other generic modelling languages. Nevertheless, UML has been found suitable in MBT introductions as a simulating and structuring language providing a framework to follow and giving ideas on organising the work [11].

2.2 TEMA Tool

An important contribution of the Institute of Software Systems to the TEMA project is a large MBT-utilising test automation tool called TEMA Tool. The idea of the tool is to pilot MBT in *on-line testing* (*Chapter 3.7*) through a GUI by performing case studies in real industrial contexts. The following description is based on reference [4].

TEMA Tool is founded on three components that are being developed. Firstly, the tool includes a model-based test generator whose tests are run on-line through a GUI. Secondly, it utilises a commercial GUI test automation system which is extended with MBT capabilities. Thirdly, its models are developed by a proper design tool set, whose concepts rely on a particular *test model architecture* (*Chapter 3.5*). The model architecture consists of three tiers that separate important concerns in GUI testing: navigation in the GUI, high-level actions, and test control related issues.

The architecture of TEMA Tool (*Figure 1*) consists of three parts: the test tool part, the model execution part and the design tool part. The test tool part provides an adaptation to the SUT. The model execution part sees the test tool part as a high-level interface through

which it can execute keywords and follow the execution results. The design tool part operates as a model creation and design entity.

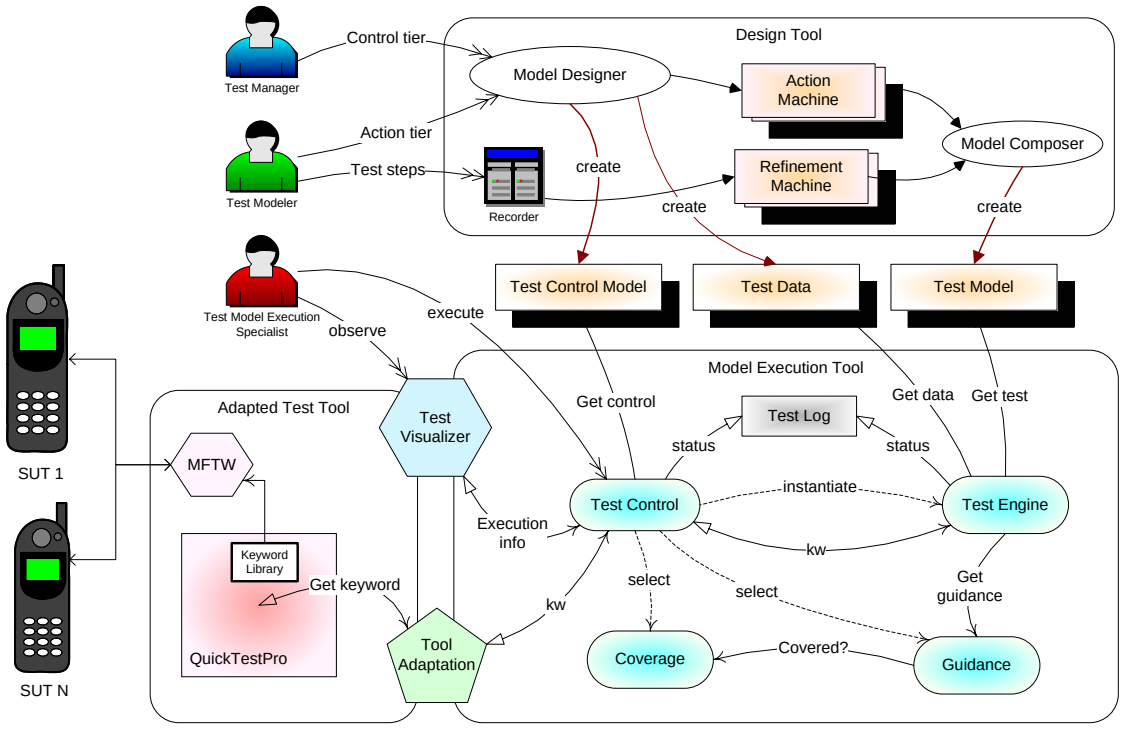

Figure 1: TEMA Tool Architecture Diagram [4]

There are two commercial components in the test tool part with which TEMA Tool interacts: Mercury Functional Testing for Wireless (MFTW) and Quick Test Pro (QTP). They provide the necessary interaction with the SUT. MFTW creates a connection to a mobile phone (our SUT) and transfers the SUT's GUI (display and physical controls) to a Windows application window (as an image and buttons). MFTW refreshes the screenshot of the SUT constantly in the application window (*Figure 2*). It is also able to recognise text in the screenshot. MFTW can connect several mobile devices at the same time.

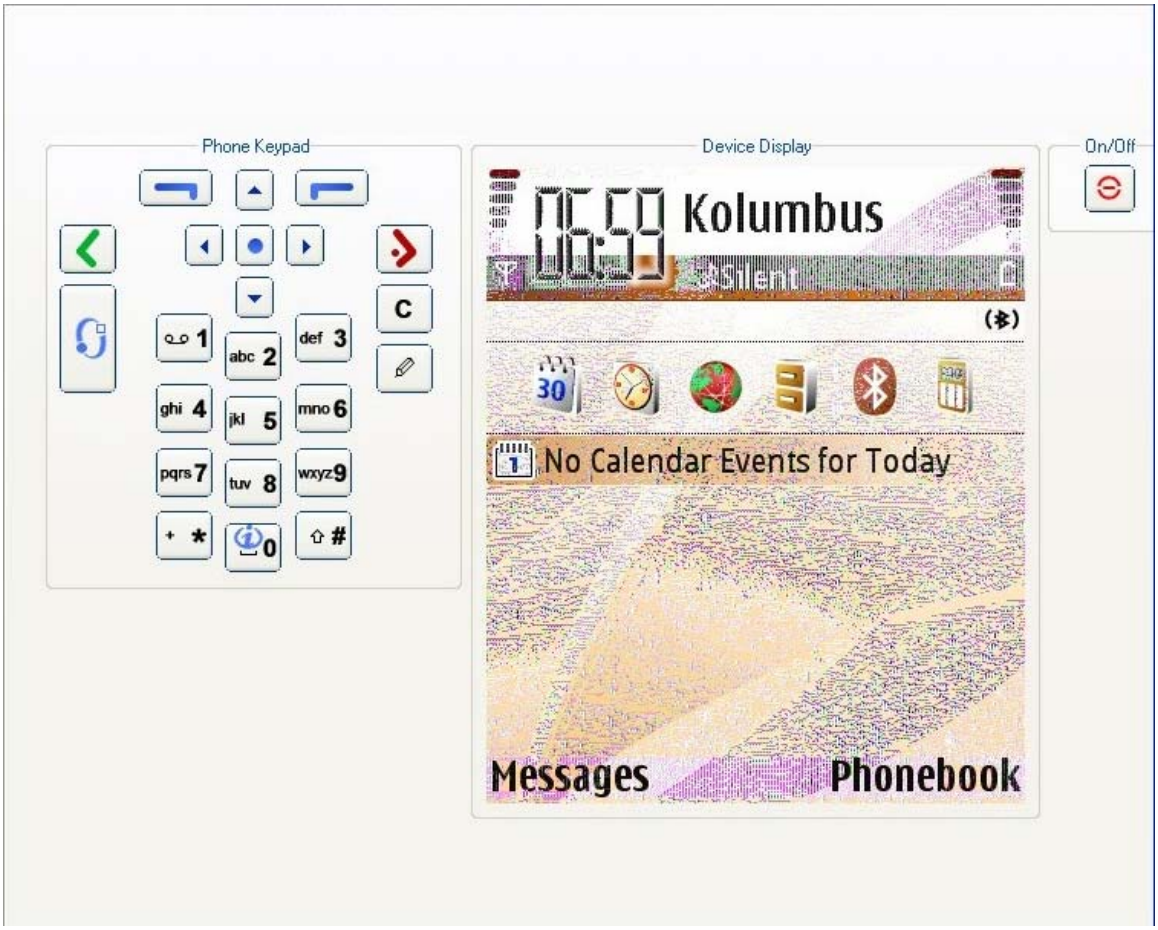

Figure 2: Mobile Phone Screenshot as seen from MFTW

MFTW receives events from QTP (which is a general GUI testing tool for Windows applications) and transfers the events directly to the SUT. The keywords are implemented with VBScript (the scripting language provided by QTP). This is conducted by converting the keywords to GUI events in the window of MFTW. There is also a small communication module (Test Tool Adapter) which connects to the model execution part of TEMA Tool. The model execution part is a separate application that could be running in another computer.

As seen in the architecture diagram in *Figure 1*, the first active component in the model execution part is the *Test Control*. When the system is started a *Test Control* component is initialised and new *Guidance* and *Coverage* components are created to guide the test run. A visual tool (*Model Designer*) is used for designing and developing test models. A test model that can be run is built using the *Model Composer*.

When a new test run is arranged the *Test Model* component is initialised and new *Guidance* and *Coverage* components are created. *Test Control* starts the test run by passing these three components (*Guidance, Coverage, Test Model*) to the *Test Engine*. During the test run information on the test is sent to the *Test Log* component. The contents of *Test Log* can then be visualised with the *Test Visualizer*, which can be utilised both in on-line and off-line modes to observe how the current test is progressing.

The roles connected to the architecture in *Figure 1* reflect a re-definition compared to conventional testing personnel roles. The *Test Manager* decides the coverage criteria, defines the entry and exit criteria to the test model execution, and specifies which metrics are collected. Additionally, the *Test Manager* focuses on communicating the testing technology aspects. For example, he explains how MBT differs from conventional testing and provides recommendations on proceeding with the new approach.

The *Test Modeler* is essentially a novel role compared to conventional definitions of testing personnel roles. The central aim of this role is the construction of test models by utilising the *Recorder* and *Model Designer* tools. This can be conducted according to the product specifications by utilising either the bottom-up or the top-down approach (*Chapter 4.3*). Furthermore, the *Test Modeler* may be responsible for the execution design along with the model.

The *Test Model Execution Specialist* investigates the test execution within the test model to confirm that the model is utilised in accordance with the principles specified by the *Test Manager*. The central aim of this role is to report the results and failures onwards. Additionally, the *Test Model Execution Specialist* documents the utilisation of test models and testing software. He should ensure that test models can be reused.

2.3 Tampere Verification Tool

The *Model Designer* tool is still under construction within the TEMA project and therefore cannot be utilised yet. In the meanwhile, TEMA Tool and the test modelling case study of this thesis (*Chapter 5*) exploit another visualisation tool. This temporary tool is part of a larger tool set called *Tampere Verification Tool* (TVT) [12].

TVT is a tool set for error detection and automatic verification of concurrent and reactive systems. It supports both action and state based verification which are enabled by the labelled state transition system (LSTS) (*Chapter 3.4*). LSTSs are utilised for representing the behaviour of a complete system, or alternatively the behaviour of a component of a

system. TVT comprises tools for advanced state space methods, and a framework for further tool development. [13]

The visualisation tool (*TVT Drawing Tool*) is utilised for some modelling tasks within the TEMA project. It is basically a graphical viewing and editing tool for LSTSs which calculates layouts for states and transitions and opens a window for a visual presentation. The dash patterns of the transitions are drawn as coloured arrows and the states are painted as coloured circles. An illustration of the tool is presented in *Chapter 5*.

Different line styles are used to differentiate actions. For instance, those actions that are related to each other could be drawn with shades of the same colour and with different dash patterns. The dash patterns can be utilised to denote a specific property of the action. For example a dash pattern consisting of a long line and a short line could indicate "message 1" and a pattern consisting of a long line and two short lines could indicate "message 2". [13]

2.4 Event Capturing Tool as Part of TEMA Tool

In the utilisation of TEMA Tool the *Design Tool* part (*Figure 1*) has to be well-developed. It is essential to design and implement tools that ease the model development and make it possible for people who might not possess any programming skills, etc. The event capturing tool (*Recorder*) functions as a model construction tool in the overall design process.

One goal of the TEMA project is the development of domain-specific languages for model-based GUI testing with action words and keywords. This can be done with the help of the *Recorder* which converts GUI events into sequences of keywords. These keyword sequences are associated with action words that correspond to concepts of the problem domain. The test modelling can be performed with the defined domain-specific language that includes the action words implemented with the *Recorder*.

2.5 Testing Target

The piloting and case studies of the TEMA project are performed in a Symbian OS environment. As Symbian OS is an operating environment designed specifically for mobile devices the actual testing is implemented on cellular phones (in this case S60-

based). However, there is nothing in the methodology of the TEMA project that prevents conducting the piloting work in other environments as well. From a technological point of view, MBT piloting is not restricted to Symbian OS alone.

In Symbian OS, S60 is a mobile phone platform that provides applications a compatible and common look-and-feel. It is basically a user interface library with some standard applications. S60 controls essential phone functionality as well as more advanced applications. Consequently, it makes mobile phones increasingly similar to small computers. [14]

In the TEMA project, S60 is a common architecture for the product family in which the case studies are performed. A domain-specific language is developed for S60 enabling the common utilisation and testing of several products in the S60 product family. The domain-specific approach is exploited in the TEMA project mainly due to the product family approach to Symbian OS piloting.

The term “product family” denotes an underlying product platform architecture which is based on similarity and commonality. It enables developers to reuse and differentiate products. That is, varying products can be derived from the architecture. Product family engineering aims at reusing components and structures as much as possible.

S60 is one example of how smart phones and other modern mobile devices have become increasingly computer-like and increasingly different from embedded systems. However, they still possess a large number of features that are machine dependent. Therefore, MBT is seen as an especially suitable approach for testing mobile devices since it eases the product family testing by raising the level of abstraction while remaining independent from test script creation and maintenance. Due to the non-deterministic nature of mobile devices, on-line testing (*Chapter 3.7*) is a preferred choice in MBT utilisation.

3. Theory

In this section the technological and theoretical background of the work is presented. Firstly, the principles of model-based testing and the differences between MBT and conventional test automation are described. Secondly, the special nature of GUI test automation is taken into account and it is compared to conventional API testing.

3.1 Principles of Model-Based Testing

Model-based testing (MBT) is an advanced software testing methodology in which the test suites are derived from a high-level model that describes (partly) the functionality of the SUT and not from test scripts as in conventional test automation. The models can be developed in parallel with the software design or created afterwards by analysing the SUT functionality.

MBT is occasionally considered a sort of black-box testing because the test cases are obtained from a behavioural model and not from the code itself. However, this is not a correct point of view. Although MBT is usually black-box testing, it can be interconnected with code-level mechanisms, and the models can be based on source code.

In conventional test automation, test execution is based on some forms of test scripts that run automatically without human interaction. However, these test scripts are laborious to manage. When the SUT is changed every test script relating to the change must be changed. Additionally, test scripts are mostly linear by their nature which hinders their ability to find defects. The test flow is tied to the script and follows the script execution. There is a need for looping and branching in the test flow.

Model-based testing carries a promise to solve these problems. It automates not only the execution of tests but also the creation of test suites. In conventional test automation, testing denotes executing the test code while in MBT the tests are generated from higher-level models describing the behaviour of the SUT.

However, it should be noted that MBT is no silver bullet. It is not a solution to all testing problems. The benefits of MBT are in control variation (enabling looping and branching in the test flow) and in data variation. Additionally, MBT is suggested to be effective especially in detecting concurrency related defects [15]. A specific limitation of model-based GUI test automation is that it can only be utilised relatively late in the software development process. Hence, a considerable amount of testing has already been performed prior to the utilisation of MBT through a GUI.

3.2 GUI Test Automation

In most cases MBT has been used for testing through various kinds of APIs. However, this kind of approach is a limited one. An equally important area of application is system testing through a GUI. Software developers are usually unwilling to design system-level APIs solely for testing purposes. In addition, the general-purpose testing tools must be tailored in order to adapt and use such APIs in an effective way.

There are many general-purpose GUI testing tools available that can be easily utilised. However, GUI testing tools are usually not greeted without reservations among the test automation community. This scepticism is often a result of bad experiences in utilising capture/replay tools that capture mouse movements and key presses, and replay those in the regression tests.

Capture/replay tools were the first generation test automation tools and had several problems [9, pp. 103-104]. For example, defects, if any, were found during the capture phase. Replaying did not provide additional benefit. There were also high maintenance costs with such tools because the GUI is usually changed more frequently than the other system and changes to it create a need for change in the GUI test automation scripts.

The times have changed since the introduction of capture/replay tools. As Fewster & Graham [16] have described, the evolution of test automation moved on to structured test scripts and later on to data-driven scripts. However, the key problem of scripted testing, the laborious maintenance, still remained. No real ease was found until the introduction of keywords and action words.

3.3 Testing with Action Words and Keywords

According to Buwalda [8] the most recommended practice in the test design process is that the test designers concentrate on high-level concepts, i.e. business process modelling. This way they can pick the chains of events that are interesting for discovering potential defects. These concepts (high-level events) are called “action words”. Action words, of course, require concrete implementations in order to be useful in the test automation. These simpler implementing events are called keywords.

Action words reflect the actions of users at a high level of abstraction. In the case of a mobile phone, for example, action words can be such as “add a new contact”, “send an SMS”, “answer a call”, “browse the calendar” etc. Keywords, instead, map every action word to a sequence of key strokes, e.g. menu navigation, text inputting etc. Action words may involve several alternative keyword sequences that implement them.

An example of a keyword in Symbian OS could be *kwPressKey* which models a key press. This keyword could be used, for example, in an action word that starts the calendar application, *awStartCalendar*. The keywords may include parameters that specify their functionality. Pressing the key “5” in the keypad could be described as *kwPressKey <5>*.

Sometimes the difference between action words and keywords is not clear. The most complicated or the most general keywords could be described as action words as well. Therefore the conceptual difference must be kept in mind when keywords and action words are defined. The level of abstraction and the purpose of use are the key factors that define which events are keywords and which are action words. Keywords are usually derived from a GUI while action words correspond to the operations of applications. [15]

3.4 Labelled Transition Systems

Labelled transition systems (LTS) are state machines in theoretical computer science, especially in computational studies. An LTS comprises a set of states and labelled transitions between the states. LTSs are, however, different from finite state automata since the group of states in an LTS is not compulsorily finite. This applies to the transitions as well. If an LTS contains a finite number of states and transitions, it can be exemplified as a directed graph.

We use basically two variations of labelled transition systems: conventional labelled transition systems (LTS) and labelled state transition systems (LSTS). The latter differ from the former in one specific way: the states of LSTSs may contain some information while the states of LTSs do not. The state information can make the states increasingly different from each other, thus enabling the expression of different functionality, for instance. LTSs and LSTSs are used in TEMA Tool and in the event capturing tool for modelling purposes.

3.5 Three-Tier Test Model Architecture

In order to conduct MBT successfully in GUI test automation, a proper test model architecture should be designed. Kervinen & al. [17] have developed a three-tier test model architecture with the intention of performing a case study with it in Symbian OS environment. This architecture (*Figure 3*) is utilised in TEMA Tool and it is also the conceptual basis for the event capturing tool development in this thesis.

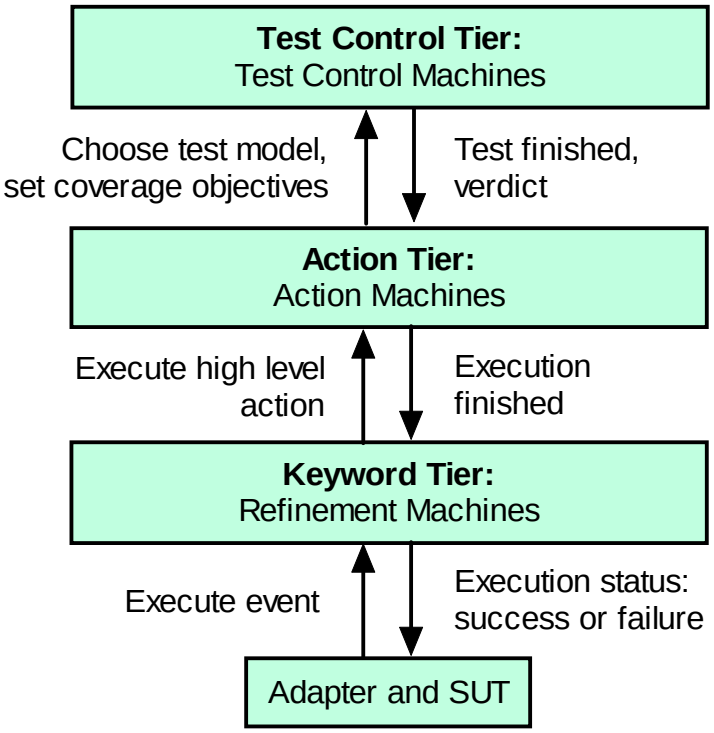

Figure 3: Three-Tier Test Model Architecture [17]

The architecture consists of three tiers that distinguish the utilised concepts in GUI testing:

1. *Keywords* for navigating in the GUI
2. *Action words* describing the high-level functional concepts
3. *Control words* which define the test control related matters

As seen in *Figure 3*, the lowest-level tier is the *Keyword Tier* which defines how to navigate in the GUI. In this tier the LTSs are called *refinement machines*. They describe the means of execution for GUI actions. *Figure 4* offers an example of a refinement machine for a S60 cellular phone. The hollow circles represent the states where execution is led while the black dot denotes the initial state. The arrows define the taken actions.

Figure 4 illustrates implementations of two action words: *awVerifyCam* and *awStartCam*. The latter, which is on the right hand side of the initial state, contains two alternative keyword implementations. The action of starting the camera is executed by either pushing the soft right key or making a selection in a menu. The implementation of *awVerifyCam* verifies that the application is really running. This verification could consist of checking whether a certain text appears on the screen as illustrated in *Figure 4*.

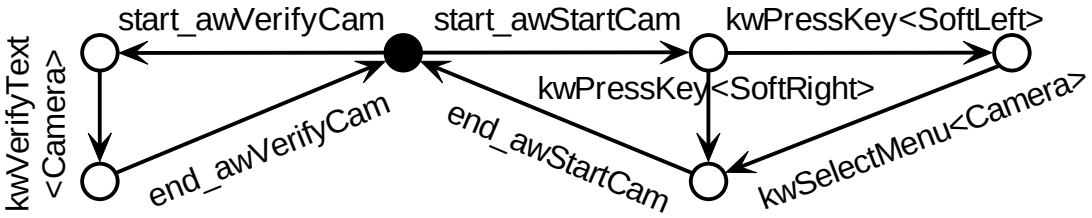

Figure 4: Refinement Machine for Camera Action Machine [17]

The intermediate tier is the *Action Tier*. The LTSs on this level, the *action machines*, describe the testable behaviour by using action words which relate to the high-level concepts. Action words are refined to sequences of keywords by refinement machines in the keyword tier. When the interactions between two (or more) applications are tested the required action machine can be built by combining the action machines of these applications. An example of an action machine in a S60 cellular phone is illustrated in *Figure 5*. The notation of circles is similar to the refinement machines. The arrows denote

actions. Initials “aw” indicate an action word while the other names represent interleaving events.

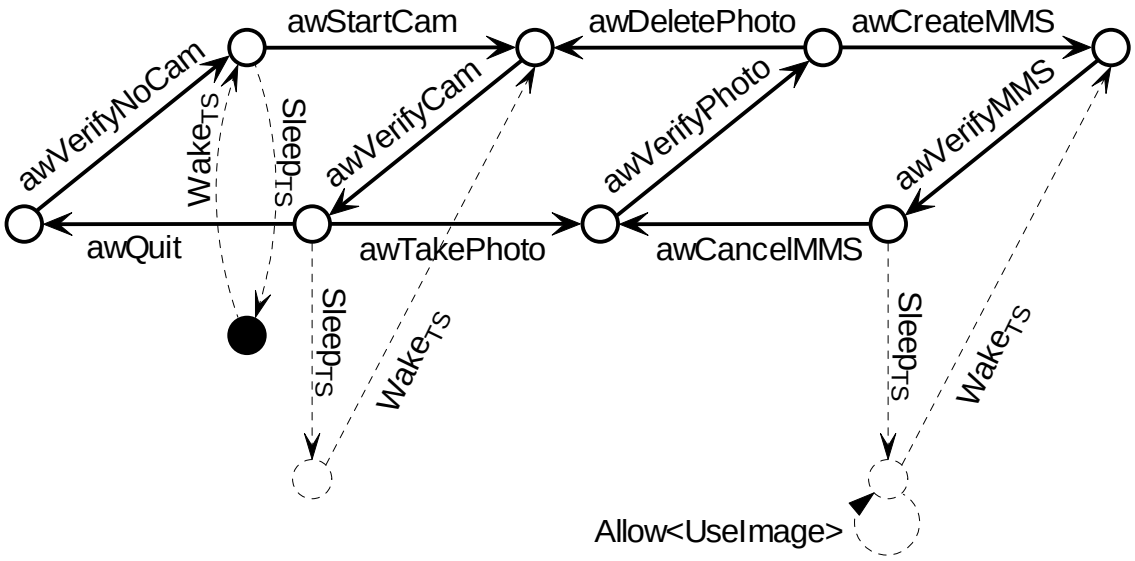

Figure 5: Camera Action Machine [17]

There are usually *Sleep* and *Wake* actions in a typical action machine (*Figure 5*). These actions denote points where the execution is allowed to overlap with other action machines. *Sleep* is for releasing the control and *Wake* is for gaining it. The circles drawn with dashes denote sleeping states. There can also be two other kinds of interleaving actions: *Allow* and *Req*. They are for resource sharing. *Allow* denotes a point where a reserved resource is released to be utilised by other action machines. On the contrary, *Req* is for requesting and gaining a resource that is required by the action machine.

Due to the requirement of looping and branching in the test flow, there have to be verifications of the states where the SUT must be in certain situations. Therefore, effective modelling requires *state verifications* (sv) in addition to *action words* (aw). Although state verifications act as action words in several cases, they should be notated separately due to clarity reasons. Verifications of states are required in the modelling because not every action word can be executed in every state of the SUT.

The highest-level tier is the *Test Control Tier* where the LTSs are called *test control machines*. A test model can be constructed by designing LTSs in the two lower-level tiers. However, in this tier the type of the test is defined (e.g. long period tests, smoke

tests etc.) as well as the coverage objectives. The preferred test guidance heuristics are also chosen.

3.6 Choosing Test Modelling Language

A major factor in the deployment of MBT is the choice of test modelling language. Although a proper language is a notable benefit, choosing a modelling language involves compromises in most cases [18]. The three major types of modelling languages in this context are domain-specific, test-specific (e.g. TTCN-3 [19]) and generic languages (e.g. UML).

The modelling languages that can be developed exclusively for the problem domain are a promising approach to MBT in cases where system testers are not fluent with any generic modelling language [20]. Due to the product family approach and the S60 platform (*Chapter 2.5*), a domain-specific modelling language is a preferred choice in the TEMA project. This way test modellers do not have to be software development experts, or even possess programming skills. Moreover, the development costs of the language and the associated tools are covered over a long period of time.

In the TEMA project, GUI test automation is emphasised and the purpose of the language choice is to model the behaviour of the cellular phone user at a high level of abstraction. Thus a domain-specific modelling language is a preferable choice because the focus can be set on the GUI more accurately than in generic or test-specific options. The utilised methods for developing the domain-specific language include Labelled Transition Systems (LTSs) combined with action word and keyword techniques for the test modelling as described earlier.

3.7 On-Line vs. Off-Line Testing

In addition to modelling the behaviour of the SUT, MBT entails the creation of test cases and descriptions of test objectives. Similarly, the test case execution in the SUT and the assessment of the results are involved. [1; 18] An illustration of these actions can be seen in the upper part of *Figure 6*. This general approach is usually called *off-line testing*. On the other hand, with some non-deterministic SUTs it may be necessary to proceed according to the observed behaviour. Hence, the test steps can be executed once they are created. This approach, which is called *on-line testing*, is sketched in the lower part of

Figure 6. Thus testing becomes a game-like interaction between the SUT and the MBT tool [21] while test cases and suites are implicit.

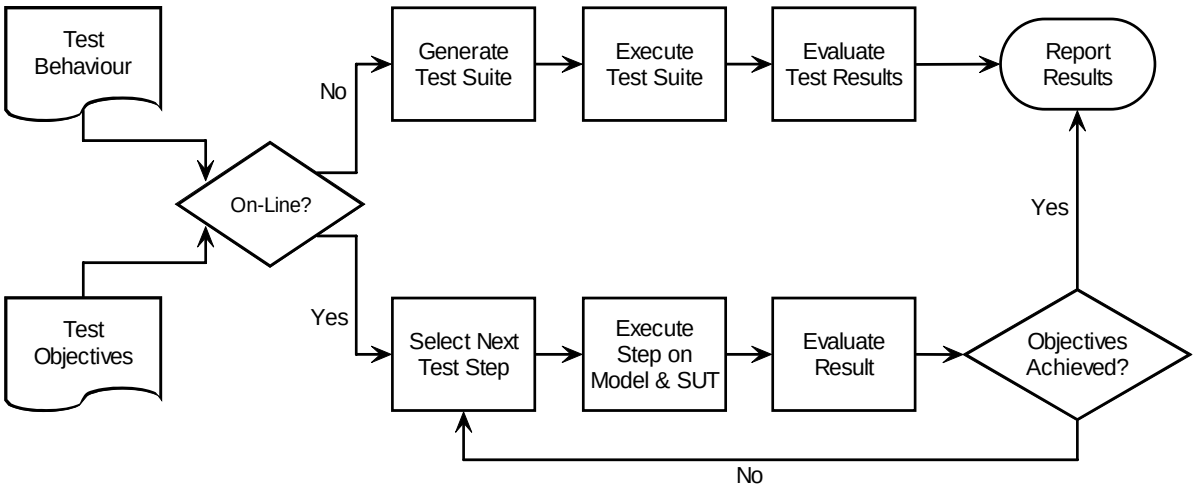

Figure 6: MBT Process – On-Line vs. Off-Line Testing [18]

It must be taken into account that the selection between on-line and off-line testing influences the architecture of the MBT tool. The tool that creates the test cases in the on-line case needs to be linked to the SUT with an adapter. The adapter's task is to interpret all data transfer, input and output, between the SUT and the tool. The process in the off-line approach is somewhat different: the test cases are created first and they are executed later in the SUT.

There are also tools which apply both approaches. An example of this kind of tool is *Spec Explorer* which is utilised within some *Microsoft* product groups [22]. In contrast, TEMA Tool is a pure on-line tool. MFTW and QTP serve as adapters (in addition to other tasks) between the S60 mobile phone (our SUT) and TEMA Tool. The on-line approach is also taken into account in the event capturing tool although the execution is not one of its tasks.

4. Event Capturing Tool

In this chapter the technological body of the thesis is presented. The high-level requirements for the event capturing tool are clarified first, prior to the overall description. Test model construction with the tool is also discussed. Special attention is given to the different types of modelling approaches. Examples of use as well as implementation-specific issues are presented towards the end of this chapter.

4.1 Needs for Event Capturing Tool

There are two kinds of needs for the development of the event capturing tool for GUI test automation. Firstly, there are academic needs, open questions and areas of scientific discussion where progress is needed. Secondly, there are project-specific needs: the TEMA project and TEMA Tool possess particular development needs and the tool set needs to be extended.

As noted earlier, some promising proposals for introducing MBT in industrial/corporate use have been presented in the scientific community. However, there are still open questions that need to be answered. The manners in which an average tester could build an effective test model must be explored. Likewise, a study must be conducted on the development of an easy-to-use tool for model construction purposes in model-based GUI test automation.

These questions are the main targets of study in this thesis. The development of an event capturing tool for GUI test automation is a proposal for easing the modelling task of average testers. Likewise, a test modelling case study is conducted with a commercial SUT (a Symbian S60 mobile phone) to test the utilisation of the tool and to retrieve

knowledge on how an average tester could build an efficient test model that can be used for detecting defects effectively.

Within TEMA Tool the *Design Tool* part (*Figure 1*, p. 14) has to be designed well. It is particularly important to develop tools that ease the model construction and make it possible for people who have not acquired programming expertise. In this context the event capturing tool operates as a model fabrication tool in the general design process.

One goal of the TEMA project is to develop domain-specific languages for model-based GUI testing with action words and keywords. The event capturing tool can be useful in this process for it converts GUI events into sequences of keywords that function as implementations of higher-level action words that correspond to concepts of the problem domain. The keyword sequences are associated with action words within the event capturing tool with the intention of creating model material.

The test modelling can be performed with the defined domain-specific language, including the action words implemented with the event capturing tool. The modelling is made easier by catching GUI events directly and producing keywords from them automatically. The keyword sequences are then easy to develop further into action words which the LTSs consist of.

4.2 Overall Description

The event capturing tool records GUI events in a similar manner to the old capture/replay tools. However, instead of generating scripts that are laborious and difficult to maintain, it generates keyword sequences which are developed into LTSs in which action words are utilised as transition labels. Keywords describe the GUI navigation while action words correspond to the user behaviour at a high level of abstraction. The keyword sequences recorded by the event capturing tool are linked to action words as implementations.

As noted earlier, the introduction of MBT tools can be eased by developing a domain-specific modelling language. Ideally the language can be achieved from the design model. Nevertheless, the difficulty in this alternative is that the design model does not necessarily exist in a normal case and there are no available transformation tools. In practice, it is usually recommended to construct the test models by developing them manually from scratch. The event capturing tool is helpful in this process.

It is important to emphasise that the event capturing tool must not be associated with the first generation capture/replay tools which are generally ineffective [9, pp. 103-104]. The event capturing tool does not conduct replay, and the capture functionality is very different. The tool captures user events, interprets them as keywords, and interconnects them as implementations of higher level action words that the behavioural model is composed of. The idea of the tool is test modelling with keywords and action words and the fabrication of a testing model by adding state information with the intention of enabling branching and looping in the test flow (executed by other tools).

4.3 Constructing Domain-Specific Test Models

The utilised action words define a domain-specific language for the domain. In the TEMA project this is applied to test modelling in two ways: top-down approach and bottom-up approach [4].

4.3.1 Top-Down Approach

If the focus is on the maintenance of the test models the top-down approach is preferable. In this approach the starting point of the modelling is the action word model. There are, however, several options for proceeding.

Firstly, the test modeller might possess some action words that do not involve one or more keyword sequences implementing them. In that case the top-down approach is utilised as a complementing method for achieving a solid model. A second option is that an implementation of an action word needs to be altered. For instance, this might be due to changes in the keyboard for the next product in the product family. Thus, this option is seen as a maintaining method.

A third way is to start the modelling from scratch by drawing an LTS of the action word model and opening it in the event capturing tool. In this option there are no keyword implementations for the action words at all at first. They are inserted one-by-one to implement selected action words.

The event capturing tool can be used to record the keyword sequences in all of these situations. Subsequent to selecting an action word the recording is started and the tool adds the detected keywords to the sequence by using the GUI of Mercury Functional

Testing for Wireless (MFTW). The modeller can also override or delete an existing keyword sequence by recording a new one or by editing the implementation manually.

4.3.2 Bottom-Up Approach

When the test models are constructed for a product family the bottom-up approach is preferable. In this approach the event capturing tool is utilised for building the test models from scratch. The test modelling is started by recording a test with the event capturing tool utilising the GUI provided by MFTW.

As the keywords are recognised the tool enters them into the end of the sequence. Free-form text can be inserted between the keywords to define the states of the SUT where the execution can, has to, or must not be. This allows branching and looping in the test model afterwards. Without looping and branching the tests generated from the models would be similar to linear and static test scripts with their limitations in finding defects effectively.

The recording requires also verification keywords (verification words) that check whether the desired text exists on the screen or not. These verifications could be performed subsequent to every event. However, that would weaken the performance. The other option is to place the verifications only subsequent to whole action word sequences but that would make debugging more difficult. Thus the choice between these two options is always a compromise. The person who constructs the test model has to make a design decision whether to emphasise performance or debugging ability.

When the test recording is finished the sequence of keywords is split into action words with the Model Designer tool. Model Designer reads in the generated keyword sequences and visualises them as LTSs. In the LTSs those states that carry equal names are equated for retrieving loops. Whenever there are states that act as the ending and beginning points of more than one loop (and the loops are equivalent to the same event sequences) the tool asks the user whether or not to delete the redundant ones.

The Model Designer tool tries to find the keyword sequences that match the ones that have been archived and suggests replacing them with the corresponding action words. The user has to select keywords and enter an action word name in order to archive a new sub-sequence. The sub-sequences must not contain branching. The process of recognising keyword sub-sequences and replacing them with action word names is continued until the model contains only action words. An alternative way of doing this is to recognise the names of the beginning and ending states of the keyword sub-sequences, and if the names

match with the ones in the archive, replace the corresponding sub-sequence with an action word.

Subsequent to the replacement process, the model is finished by inserting sleeping states to places where the execution is allowed to overlap with other action machines. In addition, *Allow* and *Req* transitions can be inserted in these points for resource sharing. *Req* is for requesting resources and *Allow* is for releasing them.

4.4 Description of Use

The event capturing tool recognises two file formats: labelled state transition systems (.lsts) and comma separated values (.csv). In the case of LSTSs the opened files contain all necessary information for defining a valid state machine. However, the tool reads in only action words and state verifications and lists them in a list control in its main window. State verifications act like action words in the event capturing tool. An example of an LSTS file is shown in *Appendix A.*

CSV is the internal file format of the tool (*Figure 7* illustrates an example). Within a CSV file, action words are listed at the beginning of every line that does not start with a comma. The lines that start with a comma are the keywords of the previous action word in the file. In the same line with a keyword are its parameters separated with a comma but not with a new line. A new line would indicate a new keyword. If there are several alternative implementations of an action word, the name of the action word is repeated in the file followed by another keyword sequence. CSVs can be opened for example in Microsoft Excel and LSTSs with a visualising tool (e.g. TVT [13]).

In a typical CSV file (*Figure 7*) the text string initials denote the types of abstraction which the strings represent. For instance, “kw” denotes a keyword, “aw” refers to an action word, and “sv” indicates a state verification. The keyword parameters are usually in angle brackets or in inverted commas. A tilde (~) denotes a negation of a keyword. That is, the execution of the keyword is not successful. For example, `~kw_VerifyText, 'Gallery'` means that the string “Gallery” must not be found on the screen.

GalleryImages_s60.csv

```
awNext
, kw_PressHardKey, <South>
awVerifyImageSelected
, kw_PressHardKey, <SoftLeft>
, ~kw_VerifyText, 'Download'
, kw_PressHardKey, <SoftRight>
awOpenImage
, kw_PressHardKey, <CenterPush>
awToMain
, kw_PressHardKey, <SoftRight>
awToVideoClips
, kw_PressHardKey, <East>
awFromMain
, kw_PressHardKey, <CenterPush>
awFromVideoClips
, kw_PressHardKey, <West>
awBack
, kw_PressHardKey, <SoftRight>
awNext
, kw_PressHardKey, <North>
awVerifyImageSelected
, kw_PressHardKey, <SoftLeft>
, kw_VerifyText, 'Download'
, kw_PressHardKey, <SoftRight>
, kw_PressHardKey, <South>
awVerifyImageSelected
, kw_PressHardKey, <SoftLeft>
, kw_VerifyText, 'Download'
, kw_PressHardKey, <SoftRight>
, kw_PressHardKey, <North>
svGalleryImages
, kw_VerifyText, 'Images'
, ~kw_VerifyText, 'Gallery'
```

Figure 7: Example of CSV file

The event capturing tool is a dialog-based Windows application which denotes that its main view is a Windows dialog. No menus exist. As can be seen in *Figure 8* there are several controls in the dialog that correspond to the associated actions. The controls inside the *Source Selection* box are for file opening, shortcutting and viewing the type of the model and the mobile phone. The *Action Word Selection* box contains the list of action words and the corresponding numbers of their implementations. In this box new action words can be created and existing ones deleted.

The keyword implementations of the selected action word are viewed and edited with the controls of the *Keyword Sequences* box. The keywords and their parameters can be edited freely. The list on the right (*Figure 8*) contains some common keywords. If a file

"KwList.txt" (*Figure 9*) exists in the same directory as the main executable of the tool, the keywords are taken from it. If it does not exist, the hard-coded keywords, seen in *Figure 8*, are listed instead. (KwList.txt is not in CSV format. It is just a list where keywords are separated by a new line.) The *Recording* box contains the start and stop buttons for recording. The recording, once started, adds every recognised keyword to the end of the keyword list on the left. From the *Operations* box the model can be saved to a file and the application exited.

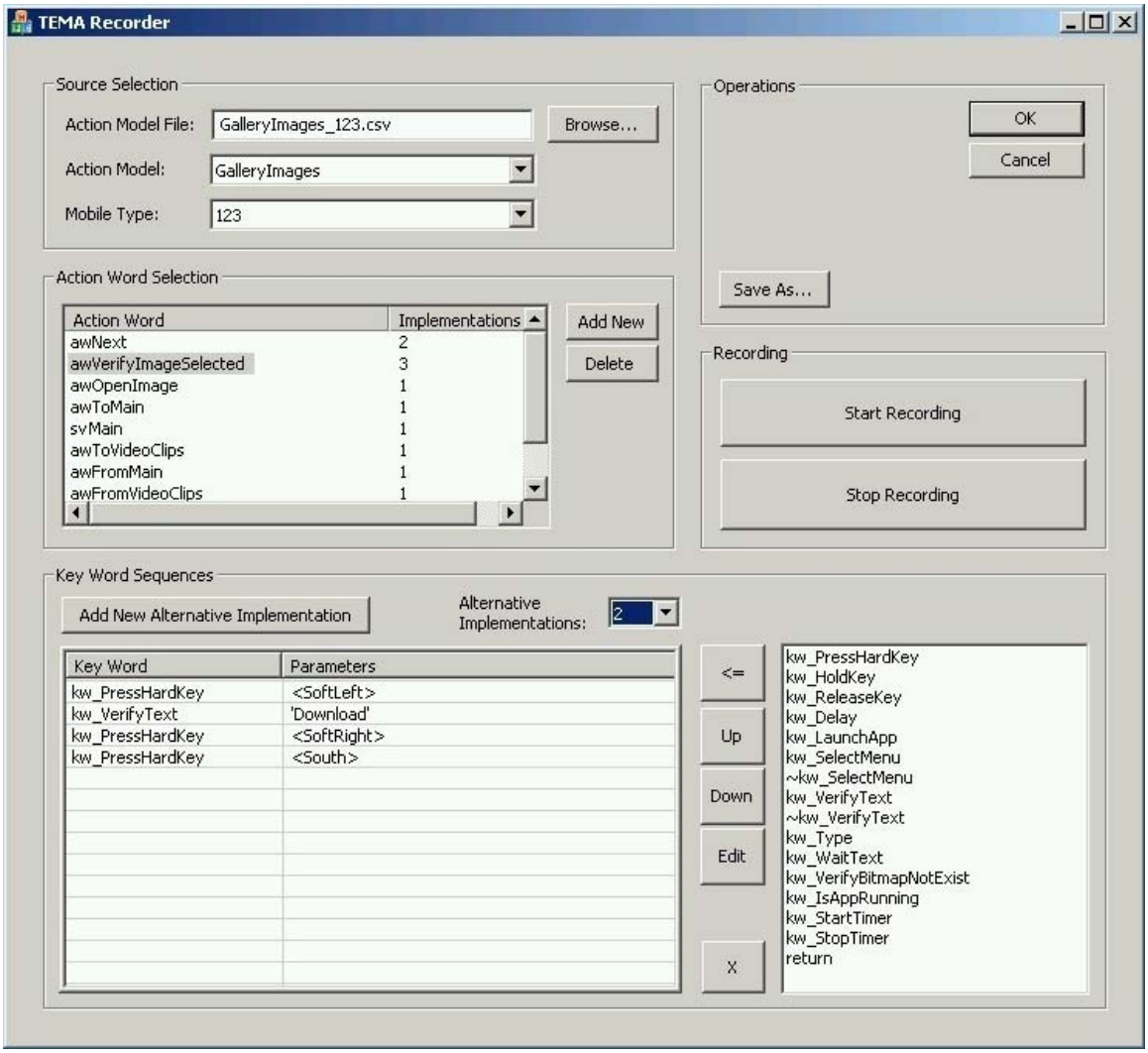

Figure 8: Main Window of Event Capturing Tool

In the case of an LSTS file there is only a list of action words ("aw" initials) and state verifications ("sv" initials) at first. No keyword implementations exist. The modelling is

started by selecting the desired action word in the list and adding keywords as its implementation. This can be done by either recording the keywords from the GUI of MFTW or entering and editing the keywords manually. When all the action words involve at least one keyword implementation the model is complete and ready for later processing.

CSV files might contain only action words but usually they also contain keyword implementations and keyword parameters. In most cases CSVs are opened for altering or extending the model. However, they might also be unfinished models when a large model is constructed in a piece-by-piece manner.

4.5 Analysis and Design Decisions

It was clear in the beginning of the development that the event capturing tool would require many GUI controls. The needs for manipulating action words, keywords and keyword parameters affected the design decision whether to use classic menus, right-click pop-ups, task bars or dialog controls. In this case dialog controls seemed to provide the best usability. Thus the solution was built as a dialog-based application.

The fact that Mercury Functional Testing for Wireless (MFTW) is a Windows application determined the operating environment of the event capturing tool. Cross-platform solutions, although technologically advanced, would have required too much effort and provided too little advantage. By contrast, Windows provides beneficial opportunities for GUI event capturing and interpretation. Developing the event capturing tool as a Windows application was therefore an easy design decision.

The requirements for the event capturing tool included the demand to process LSTS files. However, it was still undecided what would be the file format for the models containing keyword implementations. If the format was too complicated, it would limit the choices for the model processing with other tools afterwards. The comma separated values (CSV) file format seemed to be a suitable choice. It is simple and widely understood by programs but it is also expressive enough for the purposes of test modelling. One specific reason for choosing CSV was that Microsoft Excel supports it. CSV files can be opened, edited and processed further in Excel.

Due to the decision that the event capturing tool will be a Windows application the GUI event capturing techniques had to be Windows low-level mechanisms. Therefore, it was not profitable to implement the application within Java Swing or .NET frameworks.

Managed code in .NET and Windows interface in Java would have required a considerable amount of additional unnecessary work and exposed the implementation to many potential coding defects and difficulties. On the other hand, the native Win32 API would have required unjustifiable effort regarding GUI-specific issues and control implementation. Hence, the Microsoft Foundation Classes (MFC), which is basically a large wrapper of Win32 API functions, seemed to be a suitable choice as the development framework.

MFC is a framework strongly connected to Microsoft. Thus it seemed natural to take this into account when selecting the integrated development environment (IDE). Although IDE manufacturers emphasise the overall compatibility of their products, the experiences of many senior software developers have shown that this might not always be the case. Microsoft Visual Studio seemed to be a secure choice as the IDE due to its close connection to MFC.

Since the choice was made to develop a dialog-based application, the architectural design was a fairly simple task. The main window would be the main class that utilises various other dialog classes (helper dialogs), clarifying concept classes and classes implementing the recording properties. No real requirement occurred for separating the *view* layer from the *controller* as the typical Model-View-Controller (MVC) architecture does. In dialog-based applications the view and control parts are closely connected and thus it is not de rigueur to apply the MVC style that is typical for the general GUI application development. The architecture and class diagrams are illustrated in *Chapter 4.7*.

4.6 Implementation Issues

Microsoft Windows operating system provides a mechanism called “hooks” that allows the programmer to catch messages before they reach their destination. (Messages are a technique for the inter-process communication in Windows.) The hooks must be situated in a separate DLL file (*Hooks.dll* in *Figure 9*). Furthermore, the implementation requires a couple of MFC-dependent DLLs (*mfc71.dll, mfc71d.dll, msvrc71.dll, msvrc71d.dll*) that need to be located in the environment variable path or in the same directory with the main executable. The overall file scope of the event capturing tool can be seen in *Figure 9*. *Recorder.exe* is dependent on *Hooks.dll* and uses MFC-dependent DLLs. If *KwList.txt* exists in the same directory as the main executable, the pre-defined keywords are taken from it.

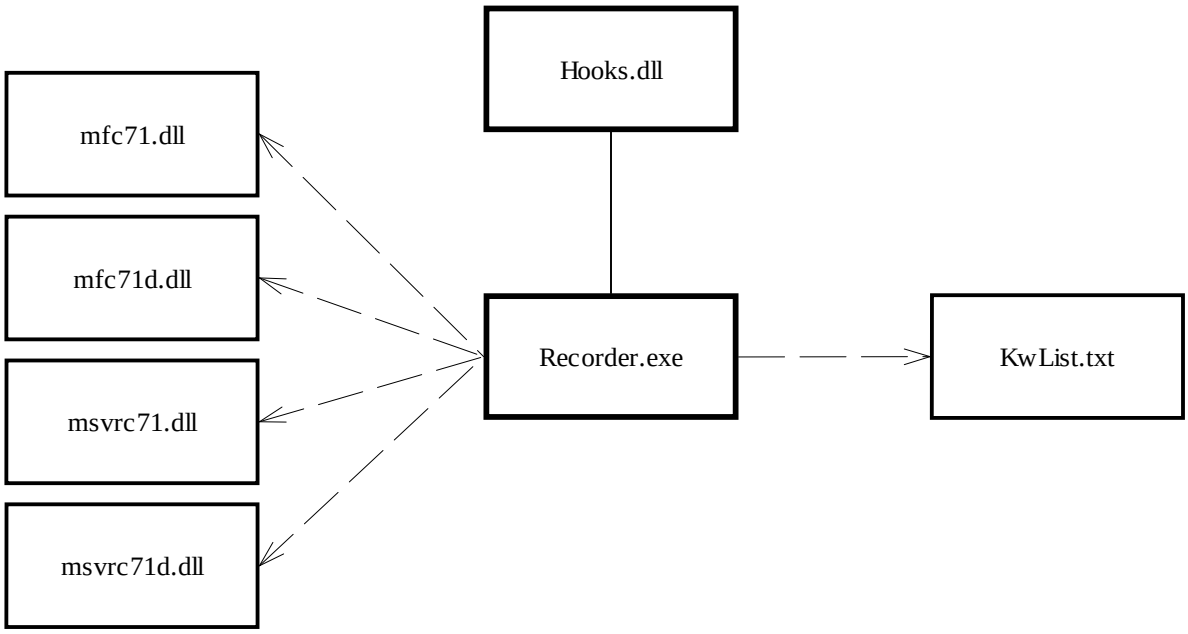

Figure 9: File Scope of Event Capturing Tool

The implementation of the recording functionality required some functions that are available only in Windows 2000/XP environment. This fact limits the number of platforms on which the event capturing tool can be utilised. In theory Windows 98/ME should provide the same functionality if the Microsoft Layer for Unicode (MSLU) is installed properly. However, Windows 98/ME environment was seen as a less important platform and thus the event capturing tool was not tested on it. It was more important to prioritise the platform testing in emulator use. The event capturing tool was smoke tested on Wine Windows emulator with no problems.

Due to the nature of GUI event capturing and the need for attaching the event capturing tool to the MFTW's user interface, the implementation was not a straight-forward task. For example, the need to retrieve information from another program's list-view control required "stealing" its memory. Naturally a pointer must be passed to the target application. However, the way that Windows uses virtual memory makes cross-application pointers invalid.

Microsoft Windows allocates memory to all of its programs by using virtual memory. It makes programs "believe" that they possess many gigabytes of memory. Furthermore, Windows prevents programs from using each other's memory. Hence, if one program fails the whole system is not terminated. Under these circumstances the only option for

retrieving the correct information is to utilise Windows functions like VirtualAllocEx, VirtualFreeEx, ReadProcessMemory and WriteProcessMemory to create an inter-process memory bridge.

There were also other programming tricks like this that had to be accomplished in order to achieve full access to the GUI of another application. Therefore, the implementation of the event capturing tool was far from primitive programming. If the problems were not fatal (i.e. did not prevent the utilisation) and a premium solution would have required far too much effort, the problems were left unsolved. For instance, attaching the hooks properly when two displays are connected to the system would have been a far too complex task to accomplish in such a small project like a constructive study of a master's thesis. It should be noted that even various commercially available testing tools have not been able to eliminate this problem.

4.7 Architecture and Class Diagrams

As noted in *Chapter 4.5* there is no real separation between view and control parts in the architecture of the event capturing tool. The application is constructed by utilising a general IDE-provided *application framework* for dialog-based applications (*Figure 10*). The *main dialog* with its *helper dialogs* serve as both view and control parts while the *journal processor* and the *hooks* function only in control purposes. The *data model* is separated from the *view & control* part and is used from within the *main dialog*.

This architecture allows a somewhat good separation of concepts. For instance, if the recording functionality needs to be changed, because the linked application (currently Mercury Functional Testing for Wireless) must be changed for some reason, only the *journal processor* part has to be re-implemented. Moreover, if a need occurs for additional properties that require user interaction, the main dialog does not have to be altered. Only a new helper dialog class has to be added and implemented.

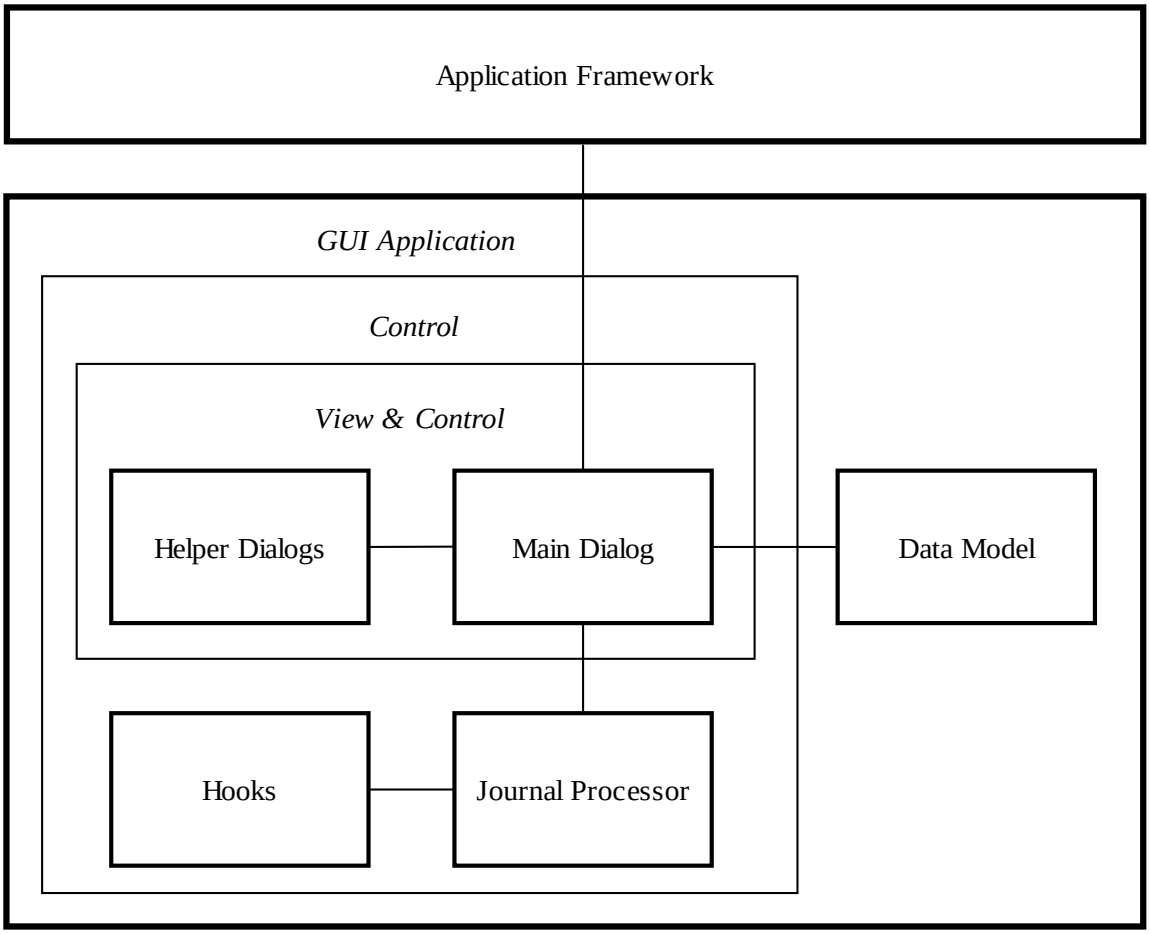

Figure 10: Architecture Diagram

Since the event capturing tool is a dialog based application, the dialog window and its controls are crucial and central. They are located in the main dialog class *CTEMARecorderDlg*. As *Figure 11* shows, this dialog class utilises most other classes in the application. The classes on the left (*CAboutDlg, CPropDlg, NewAwDlg, RecogniseDlg*) correspond to the other dialogs of the application. *CAboutDlg* is responsible for the "About"-screen, *CPropDlg* is for editing the keywords and their properties, *NewAwDlg* creates a new action word, and *RecogniseDlg* is used for text recognition in the recording mode of the application. The classes on the top (*CTEMARecorderApp, CTEMARecorderDlgAutoProxy*) are responsible for the general Windows application routines. The *AcionWord* and *KeyWord* classes describe the domain concepts.

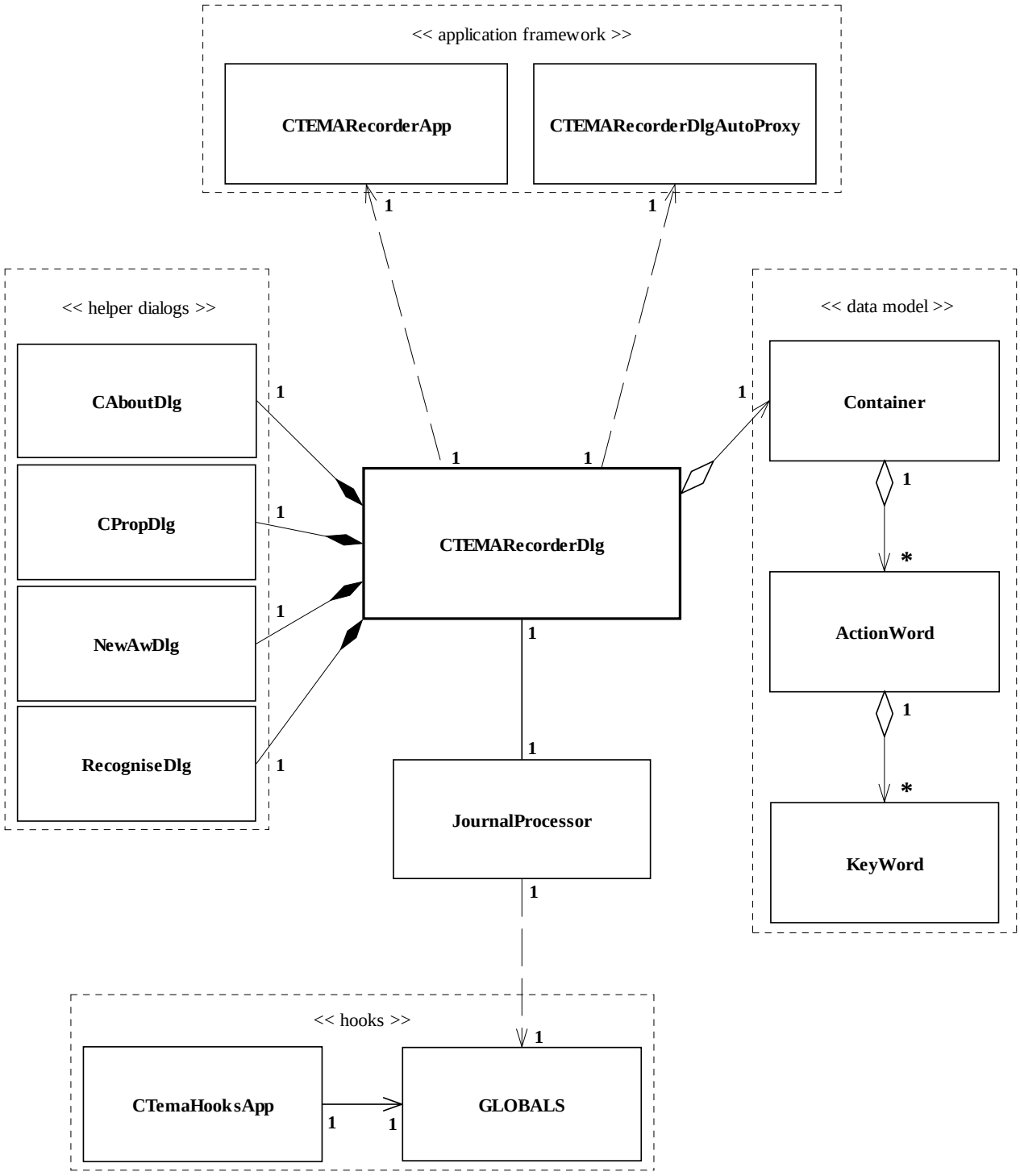

Figure 11: Recorder Class Diagram

JournalProcessor class is responsible for all the recording properties of the event capturing tool. It processes all the user events and interprets them as keywords. *JournalProcessor* utilises the *Hooks.dll* file which contains the low level functionality to capture Windows messages. *Hooks.dll* contains *CTEMAHooksApp* class, which is for the application management, and *GLOBALS* file scope that contains low level operations for attaching to the operating system's message transmission.

5. Case study

One of the objectives of this thesis work was to study how the event capturing tool could be utilised in real test modelling and how it could simplify the tasks of an ordinary tester. The test modeller's job (*Figure 1*, p. 14) is to perform the actual modelling: setting proper action words and refining them with keyword implementations. How would the event capturing tool be employed in this process?

5.1 Testing in Real Environment

It was necessary to utilise the event capturing tool in a real environment subsequent to its implementation. In addition, there was a prominent need for test modelling in the Symbian S60 environment within the TEMA project. As a consequence, the key functionalities of an S60 mobile phone were decided to be modelled with the event capturing tool. The questions on the applicability of the tool and the ease of introduction were taken into account during the modelling.

The concrete modelling was focussed on ten applications e.g. gallery and calendar. A few dozen test models were constructed. Both top-down and bottom-up approaches (*Chapter 4.3*) were utilised. However, in several cases with the S60 mobile phone it appeared to be effective to use a combined form of these approaches. That is, some action words are constructed with the top-down approach and others with the bottom-up approach.

The case study was performed at a university location, but otherwise the testing environment resembled as much as possible a real environment. The tester who performed the actual test modelling in the case study was a summer job worker (a student) at the Institute of Software Systems. Over a period of two months, he concentrated on test modelling with the event capturing tool and visualising models with

the temporary substitute of Model Designer – the visualisation tool of TVT (Chapter 2.3). The tester had taken courses on programming etc., but did not possess prior experience on software testing or industrial software development.

In the case of bottom-up approach the tester started the modelling by selecting the application to be modelled in the SUT. First, prior to any technical action, the tester sought an overall understanding on how the application operated. Subsequent to that the tester attempted to decipher how to describe the functionality of the application with action words. An action word model was constructed with the visual modelling tool (*Figure 12*) of TVT.

When the top-down approach was used the tester sought all the possible ways in which the action word model could be implemented as keywords. The tester opened the action word models in the event capturing tool and either recorded the keyword implementations or added them manually.

The utilisation in practice showed that the most difficult part of the modelling was the functionality that required looping. The possible solutions appeared to be additional state information added within the event capturing tool and the loop construction in a visual modelling tool. A classic solution would require one action word per each loop iteration. However, that would expand the model exponentially.

Some useful tips on improving the event capturing tool were reported as well during the modelling. This showed that the maintenance of the test tool is important and must be taken into account pending the introduction of the tool and subsequent to it. The old wisdom seemed to be true: if the user's needs for maintenance should cease it is a sign of abandoning the tool.

Visualisation tools (*Model Designer*) are still under construction in the TEMA project. Nonetheless, an acceptable first-phase tool already exists (i.e. TVT) to demonstrate the model visualisations. An example of model visualisation can be seen in *Figure 12*. The model of which visualisation is shown in *Figure 12* describes the *FileManager* application in an S60 mobile phone.

Various suggestions for improving the TVT visualisation tool were reported pending the test modelling case study. They involved mainly usability and clarity issues. For example, the names of the actions could be visible in the model aside the dash patterns. This would clarify the model and make it more understandable for other people than the modeller. Additionally, a requirement occurred for placing nodes and dash patterns in

different layers (that can be set invisible when needed). Currently several models are quite confusing due to this limitation. A possibility to change the order of action names in the list was required as well. Subsequent to several additions, removals and fixes, the actions were listed in a very arbitrary order. There was also a need for a snap-to-grid property which would improve the visual representation by clarifying the layout. These findings provide important feedback for the future development of *Model Designer*.

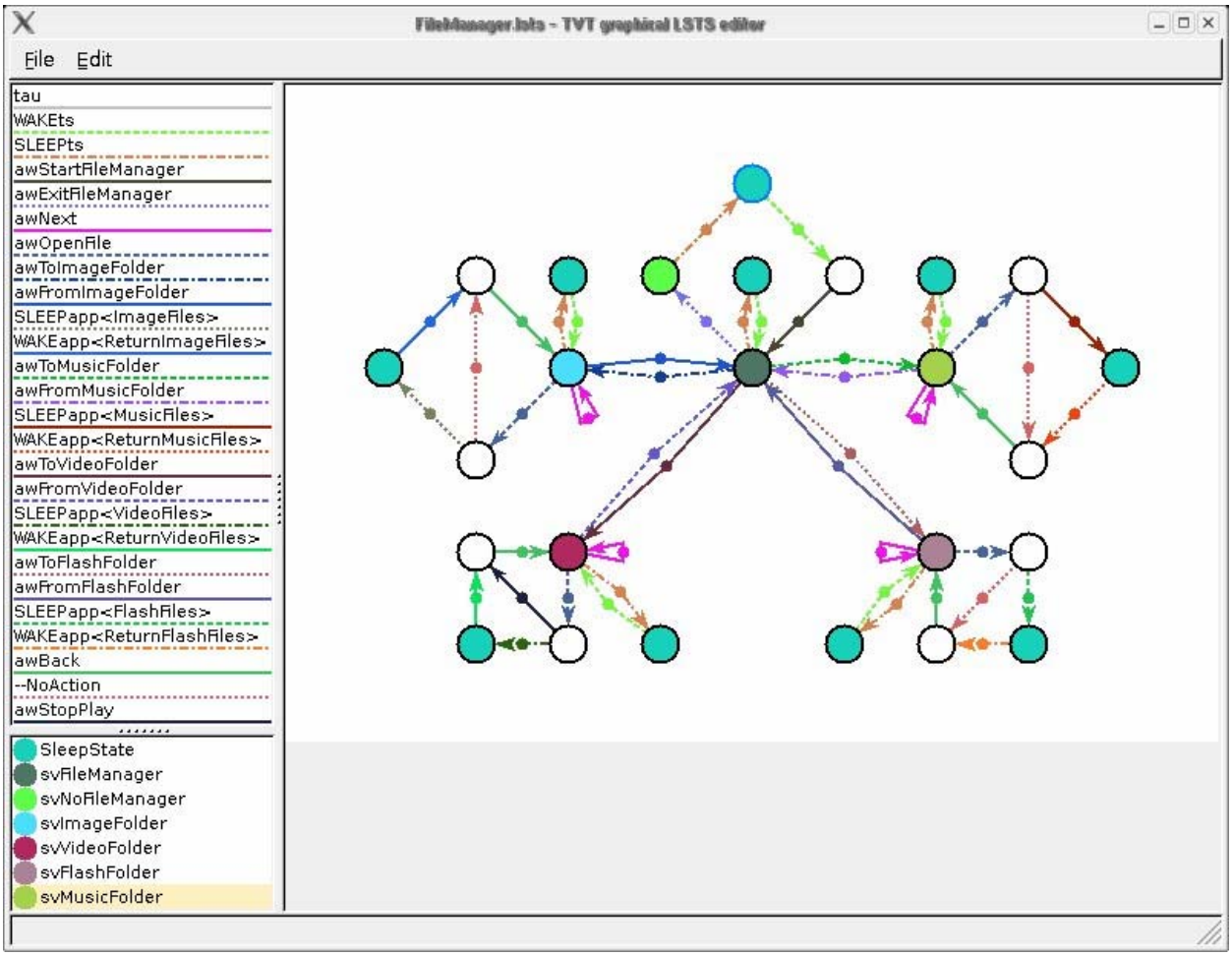

Figure 12: Model Visualisation Example

In model visualisations (*Figure 12*) the hollow circles represent the states of the SUT. The circles filled with turquoise colour are sleep states where the test flow can be transferred to other models. The other coloured circles represent state verifications (*sv-* initials in the list on the left of *Figure 12*) which are utilised to assure that the test flow is

in a correct state. The transitions (represented by arrows) correspond to the actions taken described by action words (*aw*-initials in *Figure 12*).

5.2 Problem Setting and Solving

The research questions of the thesis are:

1. How to develop an easy-to-use tool support for model creation purposes in model-based GUI test automation?

2. How could an average tester construct a productive test model that is capable of detecting defects efficiently?

A solution to the first question is the event capturing tool and its development which are described in *Chapter 4*. In practice it is often necessary to construct test models by designing them by hand from scratch. Test modelling can be eased by capturing GUI events directly and producing keywords from them automatically. The keyword sequences are then easy to process further into action words. A tool that suits these aims should advance the ease of use and thus simplify the introduction of MBT.

Moreover, the tool development showed that it is particularly necessary to take the operating system specific issues into account. These issues may take a major amount of development time. It was also noticed that the problem domain concepts must be prioritised. Additionally, a practical architecture design improves the re-usability of the tool and eases maintenance.

The second question was studied by performing the actual test modelling with the tool for and within the TEMA project. The industrial case study was conducted on a S60 mobile phone by modelling the most essential parts of its functionality from the testing point of view. The case study showed that it was not necessary for the test modeller to possess programming skills or understanding of the software development process. Some requirements for improving the usability and editing properties were discovered. These improvements would decrease the amount of manual manipulation of the models and therefore reduce the need for understanding implementation-specific issues.

5.3 Evaluation

The actual test modelling in a real environment provided a concrete way to observe how the event capturing tool could be utilised and if it really concealed the programming side and implementation-specific issues that easily become bottlenecks in conventional testing deployments. It was observed that the modelling with the event capturing tool was successful from an introductory point of view. No programming skills were required during the test modelling case study. In addition, those parts that required some knowledge on the implementation were possible to solve by improving the usability and properties of the tool.

In terms of modelling complexity, the Symbian OS S60 mobile phone was found to be intermediate but by no means trivial. There were some areas of logic, such as the SMS functionality connected to contacts application, which required deeper acquaintance with the SUT than an ordinary user would possess.

6. Conclusion

Model-based testing (MBT) involves several significant benefits compared to conventional scripted testing (e.g. automatic creation of test cases). However, problems occur frequently in the deployment of this methodology in industrial and corporate contexts. There are usually managerial problems, problems of making easy-to-use tools, and problems with reorganising the work with the tools. The focus of this thesis is on the tool problem.

Moreover, MBT introductions have been utilised mainly for API testing although it is equally important to conduct the testing through a GUI. Especially in GUI testing the testers must be aware of the client requirements and demands. Thus this kind of testing is usually performed by environment experts who do not necessarily possess programming skills and therefore need easy-to-use tools to support their work. The question of tools is particularly important in GUI testing.

Notwithstanding some promising proposals of MBT for GUI test automation, presented by e.g. Rosaria & Robinson [5] and Katara & al. [4], there are still open questions that need to be answered. Firstly, how could an average tester construct a productive test model that is capable of detecting defects efficiently? Secondly, how to build an easy-to-use tool for model creation purposes in model-based GUI test automation? The intention of this thesis is to provide solutions to these questions.

The outcome of the work is an event capturing tool, its development and a test modelling case study (*Chapter 5*) conducted with it in an industrial context with a commercial SUT, a Symbian S60 mobile phone. By developing an event capturing tool for GUI test automation, a proposal is introduced for easing the model construction performed by average testers that do not necessary possess programming or software developing skills. In addition, a test modelling case study is conducted with a commercial SUT to test the

applicability of the tool and to gain knowledge on how an average tester could construct a productive test model that is capable of detecting defects efficiently.

As a result of the study, it was discovered that the test modelling without expert software knowledge is conceivable and will be simplified by developing a proper tool. The test modelling did not require understanding of the programming issues since the problem domain concepts were emphasised in the tool. Some implementation-specific knowledge was needed. However, this was fixed by improving the usability of the tool. In industrial deployment it is particularly significant to bring theoretical knowledge into productive action. The event capturing tool appears to function well towards this aim.

The results show that a properly selected and developed methodology and an appropriate tool that implements it enhances the ease of use in an industrial case and thus make the introduction easier. Further, the practical applications of this work include the ability to link it to *TEMA Tool* and thus be a part of a larger technological study on MBT. The results were as expected and support earlier research and findings by e.g. Kervinen & al. [15] and Fewster & Graham [16].

The tool question is only one part of the process of easing the introduction of MBT in an industrial environment. The questions regarding the test personnel roles and other managerial problems need to be studied more thoroughly in the future. In addition, the problems on reorganising the work with the tools are an area of application that must be examined more in-depth. The parallel introduction of managerial and organisational solutions with the technological solution has to be analysed carefully as well.

The contributions of this thesis challenge to take the model-based paradigm increasingly to the very core of the whole software development process. Further research could be conducted on how to combine MBT with model-based analysis, design, implementation and maintenance of software to create a solid model-based software development theory and its applications.

References

[1] El-Far, I.K., Whittaker, J.A.
Model-based Software Testing.
In: Marciniak J.J. (ed.).
Encyclopedia of Software Engineering.
New York, USA 2001, Wiley.

[2] 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, December 2003.
Germany 2003, Imbus AG.
pp. 118–127.
Available at:
http://www.model-based-testing.org/ObstaclesAndOpportunities.pdf.
[Referred May 2006]

[3] Blackburn, M., Busser, R., Nauman, A.
Why Model-Based Test Automation is Different and What You Should Know to Get Started.
In Proceedings of the International Conference on Practical Software Quality and Testing, PSQT/PSTT'2004 East.
Washington, D.C., USA,
March 22-26, 2004.
Software Productivity Consortioum, 2004.

[4] Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Satama, M.
Towards Deploying Model-Based Testing with a Domain-Specific Modeling Approach.
In Proceedings of the Testing: Academia & Industry Conference – Practice And Research Techniques, TAIC-PART 2006.
Cumberland Lodge, Windsor, UK,
August 29-31, 2006.
IEEE Computer Society, 2006.

[5] Rosaria, S., Robinson, H.
Applying Models in Your Testing Process.
Information and Software Technology 42(2000)12,
pp. 815-824.

[6] *TEMA Research Project homepage*.
Available at: http://practise.cs.tut.fi/project.php?project=tema
[Referred July 2006]

[7] Symbian Ltd.
Symbian Operating System homepage.
Available at: http://www.symbian.com
[Referred May 2006].

[8] Buwalda, H.
Action Figures.
Software Testing and Quality Engineering 5(2003)2,
pp. 42–47.

[9] Kaner, C., Bach, J., Pettichord, B.
Lessons Learned in Software Testing.
New York, USA 2002, Wiley.
286 p.

[10] Bergstra, J.A., Klop, J.W.
Algebra of Communicating Processes with Abstraction.
Theoretical Computer Science 37(1985)1,
pp. 77-121.

[11] Cankar, N.
Model-Based Testing Using UML.
Master of Science Thesis.
Helsinki 2003.
Helsinki University of Technology.

[12] Virtanen, H., Hansen, H., Nieminen, J., Erkkilä, T.
Tampere Verification Tool.
In Proceedings of TACAS 2004.
Vol. 2988 in Lecture Notes in Computer Science,
Springer, 2004.

[13] *Tampere Verification Tool homepage*.
Available at: http://www.cs.tut.fi/ohj/VARG/TVT
[Referred July 2006]

[14] *S60 homepage*.
Available at: http://www.s60.com
[Referred July 2006]

[15] Kervinen, A., Maunumaa, M., Pääkkönen, T., Katara, M.
Model-Based Testing through a GUI.
In Proceedings of the Formal Approaches to Software Testing,
5th International Workshop, FATES 2005,
Vol. 3997 in Lecture Notes in Computer Science,
Edinburgh, UK, July 11, 2005.
Springer, 2006.

[16] Fewster, M., Graham, D.
Software Test Automation: Effective use of test execution tools.
New York, USA 1999, Addison Wesley.
574 p.

[17] Kervinen, A., Maunumaa, M., Katara, M.
Controlling Testing Using Three-Tier Model Architecture.
In Proceedings of the 2nd International Workshop on Model-Based Testing, MBT'06,
Satellite workshop of ETAPS 2006,
Vienna, Austria, March 25-26, 2006.

[18] Hartman, A., Katara, M., Olvovsky, S.
On Choosing a Test Modeling Language.
Unpublished manuscript.
Haifa, Israel, 2006.

[19] *TTCN-3 homepage*.
Available at: http://www.ttcn-3.org/
[Referred July 2006]

[20] Domain-Specific Modelling Forum.
DSM case studies and examples.
Available at: http://www.dsmforum.org/cases.html
[Referred May 2006].

[21] Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.
Optimal Strategies for Testing Nondeterministic Systems.
In Proceedings of the International Symposium on Software Testing and Analysis, ACM SIGSOFT 2004,
Boston, MA, USA, July 11-14, 2004.
ACM Press, 2004.
pp. 55–64.

[22] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W.,
Tillmann, N., Veanes, M.
Testing Concurrent Object Oriented Systems with Spec Explorer.
In Proceedings of the Formal Methods 2005,
Vol. 3582 in Lecture Notes in Computer Science.
Springer, 2005.
pp. 542–547.

Appendices

Appendix A: Example of LSTS file

The example in this appendix is an LSTS file that contains information concerning a gallery application within a Symbian S60 mobile phone. The event capturing tool extracts only action words (*aw*) and state verifications (*sv*) out of an LSTS file. The actual state machine of a file such as this can be visualised with other tools.

```
Begin Lsts

Begin History

  1
  "gallery-aw-kw.lsts"
  "13.01.2005 11:52 <command line not given>"
  "States: 28. Transitions: 37."

End History

Begin Header

  State_cnt = 28
  Action_cnt = 36
  Transition_cnt = 37
  State_prop_cnt = 5
  Initial_states = 1;

End Header
```

```
Begin Action_names

   1 = "start_awStartGallery"
   2 = "end_awStartGallery"
   3 = "start_awVerifyGallery"
   4 = "end_awVerifyGallery"
   5 = "start_awVerifyImage"
   6 = "end_awVerifyImage"
   7 = "start_awViewImageFS"
   8 = "end_awViewImageFS"
   9 = "start_awVerifyImageFS"
  10 = "end_awVerifyImageFS"
  11 = "?kwStartApp<'Gallery'>"
  12 = "?kwVerifyText<'Images',Invert_off>"
  13 = "?kwVerifyText<'Video clips',Invert_off>"
  14 = "?kwPressKey<CenterPush>"
  15 = "?kwPressKey<South>"
  16 = "?kwPressKey<SoftLeft>"
  17 = "~?kwVerifyText<'Options',Invert_off>"
  18 = "~?kwVerifyText<'Back',Invert_off>"
  19 = "start_awImageList"
  20 = "end_awImageList"
  21 = "start_awVerifyImageList"
  22 = "end_awVerifyImageList"
  23 = "start_awSoundClipList"
  24 = "end_awSoundClipList"
  25 = "start_awVerifySoundClipList"
  26 = "end_awVerifySoundClipList"
  27 = "?kwVerifyText<'Sound downlds.',Invert_off>"
  28 = "start_awGetBack"
  29 = "end_awGetBack"
  30 = "?kwPressKey<SoftRight>"
  31 = "?kwSelectMenu<'Full screen'>"
  32 = "?kwSelectMenu<'Sound clips'>"
  33 = "?kwSelectMenu<'Images'>"
  34 = "start_awGetBackFromSoundClipList"
  35 = "end_awGetBackFromSoundClipList"
  36 = "?kwPressKey<North>"

End Action_names

Begin State_props

  "Gallery main menu" : 2 15 18;
  "Gallery/Images, there is at least one image" :;
  "Gallery/Images/Image is showed" : 3 9;
  "Gallery/Images/Image in full screen" : 12;
  "Gallery/Sound clips" :;

End State_props
```

```
Begin Transitions

  1: 5,1 2,3 9,5 3,7 12,9 15,19 17,21 18,23 20,25 22,
     28 24,34;
  2: 8,12;
  3: 11,16;
  4: ;
  5: 6,11;
  6: 1,2;
  7: 1,4;
  8: 7,13;
  9: 1,6;
 10: 1,8;
 11: 10,31;
 12: 14,17;
 13: 1,10;
 14: 13,18;
 15: 16,33;
 16: 1,20;
 17: 1,22;
 18: 19,32;
 19: 1,24;
 20: 21,27;
 21: 1,26;
 22: 23,30;
 23: 1,29;
 24: 28,30;
 25: 1,35;
 26: 27,36;
 27: 25,36;
 28: 26,36;

End Transitions

Begin Layout

  1 246 177
  2 386 -38
  3 542 414
  4 -6 -32
  5 246 -36
  6 285 49
  7 329 101
  8 381 48
  9 108 240
 10 302 244
 11 460 424
 12 409 482
 13 264 287
 14 310 382
 15 509 21
 16 474 87
```

```
 17 92 154
 18 548 123
 19 489 179
 20 202 484
 21 193 312
 22 150 -25
 23 207 25
 24 563 255
 25 356 229
 26 466 288
 27 412 259
 28 528 320

End Layout

End Lsts

  "Run time: less than 5 sec."
```