An approach to automating mobile application testing on Symbian Smartphones

Functional testing through log file analysis of test cases developed from use cases

ISAK FÄRNLYCKE

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

An approach to automating mobile application testing on Symbian Smartphones: Functional testing through log file analysis of test cases developed from use cases

Isak Färnlycke

Master Thesis Report

isakfa@kth.se

2013-01-27

Thesis project performed at OptiCall Software AB in Älvsjö, Sweden.

Examiner: Professor Gerald Q. Maguire Jr.

Supervisor: Jörgen Steijer, OptiCall Software AB

School of Information and Communication Technology,
Kungliga Tekniska Högskolan, Stockholm, Sweden

Abstract

Many developers today have difficulties testing their applications on mobile devices. This is due to a number of factors, such as the fact that the mobile phone market has become even more fragmented with the introduction of touch screen technology. Existing software that was designed for traditional mobile handhelds is not necessarily compatible with the newest models and vice versa. For developers this incompatibility increases the difficulty when creating software.

Lack of resources for testing the application may lead to the application being limited to either just a specific model or in some cases to only one specific version of the operating system software. Without providing support for a large number of models the product may have difficulty attracting customers, and hence fail to gain the desired market share.

The challenge is to find a way to make testing simple, effective, and automated on a large number of mobile devices. To achieve this test automation applications are needed and a test strategy must be devised. Additionally, testing is often described as never-ending since testing generally reveals errors rather than demonstrating when errors are absent. Because of this some limitations of testing are justified.

In order to limit the scope of this thesis I have selected some of the most appropriate methods for testing, and will only examine these specific methods. The focus for the testing is not specifically to find errors, but rather to confirm that the product offers the specified functionality.

This thesis describes an approach to functional testing of an application for Symbian mobile devices based upon log analysis. Unfortunately, testing applications on mobile devices is still not straightforward, and this thesis does not shed any light upon how to lessen this complexity. However, I believe that both testing and development will be more and more built around use cases in the future. Unfortunately, automation of testing based upon these use cases will be further complicated by the increasing use of touch screens and physical input (such as gestures).

Sammanfattning

Idag har många utvecklare problem med att testa sina applikationer på mobila enheter.Detta har många orsaker, exempelvis att den globala mobila marknaden har blivit än mer fragmenterad i och med introduktionen av pekskärmstekniken och de snabba förändringar som sker idag. På grund av de många telefoner som idag finns så finns det ett behov för en automatiserad testprocess då det tar för lång tid att göra manuellt. OptiCall Solutions AB har utvecklat en applikation för Symbian S60 som behöver kunna köra på många olika telefoner.

Denna masteruppsats har målet att hitta ett sätt att automatisera testning av mobilapplikationer på olika enheter, mer specifikt enheter som kör Symbian S60. OptiCaller är målet för testerna. Testmetodologier och verktyg har analyserats och kraven har samlats in på den önskade lösningen.

Lösningen består av ett program som kör testskripten direkt på telefonen, mjukvara som analyserar testresultaten och presenterar dem i ett GUI, ett teststrategidokument, samt ett felrapporteringssystem. Med hjälp av dess kan testaren skapa sina egna skript för att automatisera och sedan samla in resultaten för analys. Detta eliminerar behovet för manuell testning och gör testningen effektivare, speciellt när man kör många tester. Analysmjukvaran är även integrerad med Felrapporteringssystemet för att underlätta felrapportering.

Table of Contents

Table of Figures

Table of Tables

List of Acronyms and Abbreviations

AB	Swedish: Aktiebolag (Joint stock company)
ANSI	American National Standards Association
DoS	Denial of Service
DTMF	Dual-tone multi-frequency signaling
GPL	GNU Public License
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
LSK	Left Soft Key
MEX	Mobile Exchange
NTS	Nokia Test Suite
OS	Operating System
PBX	Private Branch Exchange
PC	Personal Computer
PHP	PHP: Hypertext Preprocessor
Plc.	Public Limited Company
PSTN	Public Switched Telephone Network
Q2	Second Quarter of a calendar year
RDA	Remote Device Access
RPM	Remote Phone Management
SIP	Session Initiation Protocol
SMS	Short Message Service
STIF	System Integration and Test Framework
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
XML	Extensive Markup Language

1 Introduction

This chapter introduces the problem and the challenges in finding a solution to this problem. Also, the scope of the thesis will be presented, along with the requirements on the solution.

1.1 Problem Statement

With the growth of the cell phone market and the introduction of touch screen technology the demand for mobile applications has increased enormously. The result of this demand has been a tremendous increase in the number of mobile applications. As a result mobile software testing has become even more important, especially for entrepreneurs and companies that want to profit from applications specifically developed for the mobile market. Unfortunately, the increased fragmentation, i.e. multiple operating systems and versions of mobile phones and operator branded or manufacturer modified systems, means that this testing needs to cover more different devices and systems [1]. With all the manual work going into testing of mobile applications on all these different devices and versions, the problem is to find a way to find an automated way to perform this testing.

1.2 Background

With the growth of the smart phone market many application developers are developing software for mobile devices. Currently the major mobile software platform vendors (e.g. Google, Apple, Microsoft, RIM, and Nokia) are fighting to get developers to write programs for *their* platforms by making simple and user friendly tools to facilitate application development for their specific platform. However, problems often arise when developers need to ensure that their applications actually work on the devices that run these platforms. Also, backward compatibility with older versions of the platforms is an issue that is frequently raised as the market becomes increasingly fragmented.

One solution is to buy all the different models of smart phones needed and test each software release against every device. A more common approach is to develop for a single model using the standard libraries for the platform. When released, if this software version works satisfactorily on the particular smart phone model that you have tested your software on, then it probably works on some other models that are running the same version of the operating system (OS). Of course customers and end-users may not be happy about this approach to testing, since the application might not work at all on their smart phone or even worse, might destroy settings or other valuable information on their devices.

These compatibility problems occur because the handsets are running different versions of the software, they have different display sizes, there is a difference in the software due to operator branding, or some other factor that is different in the software or hardware configuration of the devices. What developers need is an easy and cost-effective way to test their application on multiple devices. As of December 2012 there were approximately 90 different models of smart phones on the market running the newer versions of Symbian S60 (specifically the third and fifth editions).

It is not feasible to test an application on each and every one of these phones manually as testing a specific application on each different cell phone takes from one day to a week. The objective of testing the OptiCaller mobile application for this thesis project was to find a way to test the company's software releases in order to ensure that each release of the product is stable and functions on as many different smart phone models as possible. To accomplish this, a testing strategy is needed which is as effective and automated as possible. For example, remote devices, emulators, or key generators could be used for the testing.

1.3 Overview

This thesis project tries to provide automated testing of applications on Symbian devices. Initially the focus of this testing was to confirm that the client works on different phone models and after that the testing will primarily be regression testing (to ensure that changes in this software continue to function on these different phone models). Unfortunately, testing on mobile devices today is often a manual task; hence automating this testing could be very useful, as this could minimize the need for manual procedures or perhaps even eliminate the need for manually manipulating the physical devices during testing. However, with complex applications and different software platforms manual testing cannot be fully eliminated.

This master's thesis project was originally specified as follows:

1) Evaluate possible testing methods and tools,

2) Write test specifications and test scripts,

3) Execute and evaluate these tests scripts on different phones, and

4) Implement a bug reporting system.

The first step was to examine different methods of testing on mobile handheld devices and to evaluate the usefulness of each of these methods. Also test methodology was to be researched. The results of this first step can be found in chapters 4, 6, and 7. The second step was to write test specifications for the proposed testing. In particular it was desirable to have a template when designing tests for an application. Some research was needed to decide upon the properties of this testing and the test template. Additionally, scripting is very commonly used for testing, so the use of test scripts was also examined. The results from this step are reported in sections 4.2 and 7.2. These results complete our discussion of step 2.

The third step was to test the application according to the test plan. The results of this testing were evaluated to measure how well the method and the tools work, as well as to suggest improvements in testing. The evaluation's primary focus is on the question of whether the test strategy covers the essential functions of the software that need to be tested and if the test procedure finds the most critical faults and issues that might occur when the application is used by a customer. This step is documented in sections 3.5 and 6.2. Finally, bugs found during testing need to be systematically reported to the developers (as noted in step 4). Therefore a bug tracking system was proposed to facilitate the achievement of development and release *mile-stones*; this bug tracking system is described in section 7.4. Finally the thesis ends with some conclusions and suggestions for future work (see chapter 10).

2 Technology

This chapter briefly introduces the Symbian operating system. Following this OptiCall Solutions' OptiCaller mobile application is introduced along with a description of the technology behind this application and the functionality to be tested is clearly stated.

2.1 Symbian

Symbian is an operating system designed for mobile devices and smartphones. This operating system was previously known as EPOC. The operating system was developed by Symbian Ltd. In 2008 Symbian Ltd. was acquired by Nokia and the Symbian Foundation was established. The goal of the Symbian Foundation was to make Symbian open source – as is the case for the latest Symbian OS release, Symbian^1.

This release is licensed under the Eclipse Public License (EPL) [2]. While this operating system is no longer actively being developed there are a very large number of existing devices that utilize this operating system, it is these devices that are the target platform for the Symbian version of the OptiCaller mobile application. The application will be described in the next section.

2.2 OptiCaller

This thesis focuses on testing the OptiCaller software developed and sold by OptiCall Software AB. OptiCaller was originally developed for Symbian Series 60 3rd edition smartphones. Additional development has been done and today the program runs on Symbian^1 (also known as Symbian S60 5th edition) smartphones. The OptiCaller client utilizes a large number of technologies, including Advanced Two-Step Dialing (call-through), data and SMS Call Back, and offers presence reporting and mobile extension.

In Advanced Two-Step Dialing (Call-through) the user calls a Private Branch Exchange (PBX) which forwards the call to the desired party. When a connection to the PBX is established, the client automatically transmits dual-tone multi frequency (DTMF) signals to indicate the desired callee's phone number. The PBX recognizes the dialed digits and setups a call to the callee. If the PBX has lower cost for making calls than the caller, then this approach can be less expensive than for the caller to directly call to the callee. This is often the case since the PBX is connected to a landline, while the caller frequently uses a cell phone with an expensive pricing plan; this is especially true for mobile-to-mobile calls when the caller is calling a callee who is a subscriber to another operator.

Call-back can be used when calling from networks with a high cost for calling abroad, e.g. when you are using a mobile while roaming in a foreign country. In this approach the server/PBX setups the call between caller and callee by calling both parties. As a result the outgoing call is changed into an incoming call to the caller – which is in turn bridged by the server/PBX to another call to the actual callee.

To initiate the call the caller sends either an HTTP(S)-request or an SMS to the PBX/server containing the caller and callee phone numbers, and their own login information. The PBX authenticates the user; if authentication is successful then the PBX first places a call to the caller and after the caller answers then the PBX places a call to the callee. The PBX then bridges the two calls together enabling the caller and callee to communicate. Further details of both call-through and call back can be found in the master's thesis of Tao Sun [3].

The OptiCaller presence solution enables the user to report their presence (i.e., status), such as "In a meeting until 2 pm", to the system by DTMF signaling. Normally this service is only available within a PBX, but the OptiCaller software makes this service available for mobile users. To further enhance the user's mobility the user can use OptiCaller Mobile Extension (MEX); this connects the user's mobile number with a landline number to facilitate business relations. (For further details of the MEX service see Tao Sun's master's thesis [3].)

3 Testing

This chapter introduces the concept of a test strategy, test cases, and various types of testing. Also, an introduction to the testing of the OptiCaller application is given.

3.1 Test Strategies

One of the first steps in testing is to develop a test strategy. A test strategy describes the testing procedure in detail and is important in order to systemize your testing. According to *Shiva Kumar's* handbook of testing [4] , the purpose of a test strategy is to:

- Provide a framework and a focus for improvement efforts, and to
- Provide a means for assessing progress.

All the requirements, system design, and acceptance criteria are merged into the testing strategy document. Additionally, a description of the testing is needed along with notations of objectives, scope, and other aspects. According to *Software Testing Times* [5], a testing strategy document should also cover:

- Project Scope

  Description of what is to be tested and how, how thorough the testing should be, and how much testing is needed.
- Test Objectives
  - Listed in order of importance and weighed by risk.
  - Requirements and acceptance criteria must be mapped to specific test plans to measure and validate results.
- Features and functions to be tested

  Exceptions must be listed with reasons for exclusion.
- Testing approach

  Describes the levels and types of testing to be conducted
- Testing process

  A detailed description of the steps in testing
- Testing tools

  A list of the tools that are used in the testing

The test strategy document that I developed is attached as Appendix A: Test Strategy.

3.2 Test Cases

A test case is a set of test inputs, execution conditions, and expected results. Each test case is developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [6].

Test cases are an important part of testing because they describe the testing procedures and goals. Sometimes more than one test case is needed to determine that an application functions as intended. Each requirement needs at least one test case, and sometimes it is useful to create two test cases per function or requirement to test the functionality thoroughly. This is especially important when conducting unit testing, in which you divide the application into so called "units" and each unit is tested individually [7]. A unit can be a method, a class, or a service. Each test case has a basic structure with several essential parts [8]:

- Purpose of the test
- Software and Hardware requirements
- Configuration requirements
- Test description
- Acceptance criteria or expected results

A popular technique is to utilize *use cases* as test cases. Use cases are possible scenarios that a user might perform while navigating through an application [9]. In the case of OptiCaller a use case is to set up a call using Data Call Back or retrieve new settings from the provisioning system. However, employing *use cases* directly as test cases is only applicable to some types of testing and has some distinct disadvantages [10].

Some of the problems with using use cases as test cases include the fact that these test cases might be incomplete, not detailed enough, too few, not updated, inaccurate, or ambiguous. However, u*se cases* are useful for functional testing, manual black box testing, and automated regression testing. Details of these types of tests are given in section 3.2. To conclude, *use cases* are most useful in so-called *positive testing*; i.e., testing that is not focused on finding bugs, but rather to confirm that the application works as it should.

3.2.1 Software Testing

Most people think that testing is simply a procedure to find faults in a product. While in some cases this might be true, today testing is more than just debugging. Testing is not only used to locate defects so that they can be corrected; but can also be used for validation, verification, and reliability measurements [11].

Validation and verification is the process that is used when evaluating whether the application behaves as specified. Reliability on the other hand often involves both hardware and software, but is related to validation and verification since it also measures, or tests, how well a system conforms to its specifications.

There are many strategies for verification and functional testing, both of which can be called *positive testing*. However, some testing is done to find faults or problems; this type of testing is sometimes called negative testing.

Chandran & Pai have stated, "*When testing in an emulator [...] some issues which are hit by the speed at which input was given cannot be reproduced easily*" [12]. Timing errors can indeed be very hard to reproduce and demand a lot of time for debugging, especially when using an emulator as not only do emulator's differ from the physical devices, but newer mobile phone models can be much faster (2-3 times) than the previous generation leading to the same type of timing problems as when using an emulator. Timing problems can also arise within threaded programs, as the behavior of the software may also depend heavily upon the order of execution of the threads.

Faults that arise due to timing problems are hard to fix, or even identify, without deep knowledge of the code structure and the hardware and software architecture. In practice timing problems can usually be fixed by reordering the program execution, adding delays, using semaphores, or introducing other control mechanisms to control the execution of threads. This leads to the question (from a testing point of view): Should the tester have deep knowledge of the code, or just know the requirements of the final product? The two answers to this question lead to the major the difference between White Box and Black Box Testing. These approaches to testing will each be described in the following subsections.

3.2.2 Black Box Testing

Black box testing, also known as *Opaque Testing* or *Functional/Behavioral Testing*, is a testing strategy in which the tester does not need knowledge of the internal design of the product. Some testing authorities say that the tester should not have access to the source code as the program should be considered a black box into which you input information. For example, *Laurie Williams* defines black box testing as: "*Black box* is *testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions."*[13]

In order to get useful results *w*hen performing black box testing the tester needs to turn the requirement specifications into a set of test specifications that state how the system should respond to a particular input when in a given state. A test is successful if the responses to the specified inputs correspond to the expected output. The expected output is defined in the test specification. This approach to testing is frequently used as a part of the acceptance testing for a product (i.e., to ensure that a given product meets the stated criteria for this product). Black box testing is most often used to validate that a product meets a specific requirement.

Advantages of black box testing include the fact that tests are done from the user's point of view (i.e., they verify that given certain input the expected output is produced) and the black box tester can work independently of the application programmer. It is important to note that test failures that occur when following the test plan may be due to inconsistencies in the specifications, rather than the software. This occurs because the tests are derived from the specifications and not the source code.

Disadvantages of black box testing include the fact that not all possible inputs can be tested due to limited time, test cases are hard to design for bad specifications, and all of the paths through the code might not be tested.

3.2.3 White Box Testing

White box testing (also called structural testing and glass box testing) is testing that takes into account the internal mechanism of a system or component [13]. Most often the white box tester is the developer of the code, but could also be a tester who has access to the source code written by the development team. The tester is supposed to know the code inside-out, thus they can generate specific inputs in order to test the product.

For example, inputs can be systematically generated to ensure that all execution paths in a program are traversed. The negative side of this is the tester may also be blind to weak spots in the architecture, hence they might choose the easiest and not necessarily the best way to test something. Because of this transparent way of testing, test results will most often be the expected results, but this may actually be the results someone else would expect. However, if the tests were conducted in a different order, with slightly changed inputs, or with some change in the state of the system the results might not be what is expected.

3.2.4 Function Testing

Function testing can be described as "black box unit testing" where the approach is to test each and every function one at a time. Some tasks performed in function testing are:

- Identify the program's features or commands.
- Identify variables used by the functions and test the behavior of the functions at its boundaries.
- Identify environmental variables that may constrain the function under test.
- Use each function, i.e. using positive testing, and then expand the range of inputs used for testing to cover the range of inputs as much as possible.
-

3.2.5 Fuzz Testing

Fuzz testing can be seen as a type of negative testing. The fuzzer generates invalid, random, or unexpected input to a program in order to test for vulnerabilities (i.e. that the program does not exhibit undesirable behavior due to these inputs). This method is useful for testing error handling, exception handling, and memory safety. The disadvantage is that fuzzing only tests a random sample of the program's possible inputs and therefore successfully passing fuzz testing cannot be considered an assurance of quality, but rather fuzz testing is primarily a bug finding tool. In addition, faults found by fuzzing are, most of the time, quite simple [14].

3.2.6 Regression Testing

The IEEE Standard Glossary of Software defines regression testing as*:*
"Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements."[15]

Throughout the testing process and product development, regression tests are run to ensure consistency of the implementation with the product's specifications. These regression tests are, as noted above, tests that are *rerun* to verify the product's functionality after changes have been made. Regression tests are often a subset of the original test cases that cover important functions. After any significant changes to the code base are made (e.g. faults are corrected or new features are added) then the regression tests are run.

The purpose of regression test is to ensure that the new code works properly and that no earlier functionality has been damaged by these changes. Faults that surface after a new revision are called regression bugs. According to Laurie Williams, some of the guidelines for regression testing are [13]:

- Choose a representative sample of tests that exercise all the existing software's functions
- Choose tests that focus on the software components/functions that have been changed.
- Choose additional test cases that focus on the software functions that are most likely to be affected by the change

3.2.7 Boundary Testing

Boundary testing is also known as boundary value analysis [16]. As the name implies, boundary testing utilizes test cases that are generated from extreme input parameter values. Extreme values are maximum, minimum, and values just outside the boundaries of expected input. Boundary testing can be used to test the input validation within a system. Therefore inputs outside the range at which the program is to operate should be rejected by the application and the user notified that the input is outside the accepted range for this input parameter.

3.2.8 Stress and Load Testing

Stress testing is testing conducted to evaluate a system or component at or beyond the limits of its specification or requirement [15]. Stress testing is also used to determine the stability of the system at (or near) an expected breaking point. For software this might correspond to avoiding a system crash due to a denial-of-service (DoS) attack or providing an orderly response when there is insufficient free memory or storage space on the device. Stress testing is very useful for testing websites that are production-critical. Types of stress testing include DoS attacks, running many resource-demanding applications simultaneously, and making countless accesses to one specific resource, such as a webserver, in a short time period [17].

In the OptiCaller case, stress testing might be used to test how the application handles multiple calls at the same time or how many calls within a short time can be correctly handled by the program. In our case these test cases are not primarily concerned with applying stress to the OptiCaller application itself, but rather to stress test the Symbian S60 OS. The reason for this focus is that the OptiCaller application is mainly used by persons to perform user driven communication. The rate at which the user gives input to the system is relatively slow in comparison to the rate at which an application can put demands on the OS.

Load testing is similar to stress testing, but differs in some aspects. The main goal of load testing is to minimize the response time under heavy load; whereas stress testing aims to crash the system by performing multiple transactions. Load testing is primarily done on servers, thus it is less relevant for the OptiCaller client. The main requirement of the client is that is needs to support all the tasks that might occur simultaneously. The ability of the client to execute all of the required tasks is primarily limited by the available memory. Symbian has built-in controls to avoid memory leaks, making extensive load testing less important than for servers.

3.2.9 Smoke Testing

A smoke test is often described as a group of test cases that confirm that the system is stable and that all major functionality is present and works under normal conditions. Most often smoke tests are automated and are run before deciding whether to run further tests; if the system is unstable, then there is no reason to further test it before fixing the already obvious errors. The purpose of smoke tests is not to find bugs, but to demonstrate stability. Sometimes smoke tests are a subset of the regression tests.

3.3 Log file analysis

Log files are output from the program. They can be used when testing in order to find defects in the system. Log file analysis uses a log file analyzer that processes the log file and gives feedback to the tester. The log file logs operations, especially the critical actions of the program, i.e. when there are requirements for the program to enter a new state. An example in the case of OptiCaller would be that the user has to *initiate* a new call before the status of the caller can be *connected*. Advantages of log file analysis include the fact that it is applicable to all programming languages capable of generating output to a file, it does not disturb other testing, and it can be used for many different types of testing and debugging.

A major limitation is that log file analysis does not verify the functionality or correctness of the program as a whole, but only confirms or denies if the specific test runs made of the program reveal faults in the software. Also, not all properties of the program can be tested and confirmed. J.H. Andrews [18] suggests that log file analysis be used when a higher reliability in testing is desired since it adds to the current testing practices. If the current software already logs its outputs and inputs then log file analysis will add an extra layer of testing that can be used to validate the program.

3.4 Test automation

Testing can often be a very manual process and demand a lot of resources. Because of this, many companies strive to automate as much of their testing as possible. However, automation is not always better than manual testing due to a number of factors. First there is the issue of maintaining the test suites and test scripts; as the cost of keeping the test suites updated with the product's requirements may render test automation useless. The resources that previously were used for testing are now utilized for test maintenance. According to SmartBear Software [19], automated software testing is the best way to increase the effectiveness, efficiency, and coverage of software testing. Test automation can lead to a substantial improvement in the development process and also simplify software support and maintenance. Automation is especially powerful when tests are made frequently, such as for regression tests, or when a test needs to be repeated many times, for example for stability or reliability testing.

Yike Liu states in his master's thesis "WCDMA Test Automation Workflow Analysis and Implementation" [20] that automated regression testing could improve test coverage by at least 40 percent for a specific WCDMA software release. Within his project, Yike managed to automate 40 percent of all functional regression tests. He states some reasons why automated testing should be implemented. These reasons not only include the fact that automation saves time, effort, and reduces costs, but also that the saved time can be spent running a wider set of test cases. In his work, he used the Test Management Approach (TMAP) software test process when designing an automated testing tool. TMAP is an approach that defines all the activities in testing, such as planning, preparation, and execution. TMAP is organized as a life cycle that is not fixed at any time. Yike based his testing of radio base stations on an internal test environment platform at Ericsson that provided functions to control basic operations of the radio base station.

3.5 Testing OptiCaller

OptiCaller has some quite complex functions, this leads to complex test cases. The key functionality such as Call Back and Call Through need to be tested thoroughly (i.e., tested multiple times with multiple configurations) in order to confirm stability. Since these functions include making calls there is a need for some type of scripted calling and call receiving. The major goal is to confirm that calls are actually set up by the client when the user calls using one of these methods. Some automation is needed for this to avoid having to make hundreds of calls manually. Other tests include installation, configuration, and testing of basic functionality such as application stability and language localization.

Some of these tests can be performed at installation time and would demand too many resources to automate, but in some cases these tests might need to be repeated to establish that the software functions as expected. Testing of actions that occur only once or twice is a question of efficiency and economics. Depending upon the impact of these operation such tests might not be economic to automate, although some tasks such as installation of the software on a given platform might only occur once - it is critical that the software install properly, otherwise users will not be able to use the software and there will be a high cost due to either product support or returns.

4 Test tools

This chapter introduces the tools available to assist when automating testing. Furthermore, emulators and remote solutions are presented.

4.1 Emulators

Emulators are often useful for testing. An example of when an emulator is useful is when you have limited resources and want to test if your software will run on a device that you do not have access to. Unfortunately, since the emulator does not run on the same hardware as the software will eventually run on - the results may not be exactly the same as if you had tested on the actual device. However, for general testing of interfaces, menus, and configuration using an emulator is a low cost and fairly effective development and testing method.

A Brazilian study of usability testing on mobile devices showed that "*many important usability problems can be found in simpler laboratories approaches. However, the validity of the usability problems identified in the emulator setup may depend on the similarity between the emulator and [the actual] mobile phone's interfaces.*"[21] Thus, emulators are good for finding usability problems, but it is not certain that all of these problems will exist on the actual device. Similarly, problems may occur with the actual device that do not occur when using the emulator.

In short, emulators are good for initial testing and development, but may not appropriate for product testing since they might give incorrect information. Also there is a major disadvantage in the case of OptiCaller since the emulator does not have the ability to make calls.

4.2 Automation Tools

Automating tests is often important when you plan to repeat a test many times. There is software that can help with this automation by providing a framework. Some of these solutions include a graphical user interface (GUI) and can help when testing almost any part of the product. These automation tools can in some cases automatically capture screen images and compare them to the desired output. In other cases they simply run a script or replay the input.

Liu and Wu report that according to a survey of 250 organizations, only 35 percent of the testers who had adopted automated testing tools were still using them after a year (see page 8 of [22]). The reason stated was the inadequacy of the tool. Moreover, they declare that the developers, if not involved in the testing, must at least be familiar with the testing tool. Developers or testers who do not have sufficient knowledge of the tool and its functionality may reduce their enthusiasm for using the tool.

This in turn may lead to new test scripts no longer being created or existing scripts updated, thus the tool will lose its value for the testers. The advantage of many of these tools is that you can easily write scripts using a predefined syntax, using *action words*, or *test verbs*. There are of course limitations of these different tools.

Antti Kervinen, et al. write about automated GUI testing: "*Among the test automation community, however, GUI testing tools are not considered an optimal solution. This is largely due to bad experiences in using so-called capture/replay tools that capture key presses, as well as mouse movement, and replay those in regression tests."* [23]

Antti Kervinen, et al. [23] state that the bad experiences are mostly due to high maintenance costs for these tools, since the GUI is frequently a very volatile part of the system. For example, a minor change in the GUI might generate false *negative* test results and the testing system might therefore need maintenance every time the GUI changes. Despite this testing, testing the application on the actual device is essential to confirm the functionality of the application. For complex applications and systems, automation of testing is a good means to improve stability and lower total costs.

4.2.1 TestQuest Countdown

TestQuest Countdown [24] is a package of tools developed by Bsquare for testing mobile devices. With Countdown you can easily construct test cases using their TestDesigner, the graphical test design and development part of CountDown. You can also automate these test cases using TestRunner, which utilizes servers connected to mobile devices to offer distributed test execution with logging. Countdown is a complete solution that works for many different platforms and is effective for testing applications if you do not have any in-house testing software. Unfortunately, as with many other advanced solutions, the software is proprietary.

Bsquare has a second product called TestQuest Pro. This software is for test automation and uses the same techniques as Countdown, but lacks test management functionality. In TestQuest Pro you can write your test scripts in ANSI C or using a Script Recorder that uses Test Verbs. An example of TestQuest Test Verb Technology based scripting is the following script [25]:

```
TEST_CASE_START("Default System");
SET_POWER( ON);
NAVIGATE_TO("ADDRESS_BOOK");
ADD_CONTACT("Joe Smith");
//Verify entry is present
VERIFY_CONTACT("Joe Smith");
DELETE_CONTACT("Joe Smith");
//Verify entry is deleted
NAVIGATE_TO("ADDRESS_BOOK");
TEXT_MUST_NOT_BE("Joe Smith");
TEST_CASE_END( );
```

4.2.2 UserEmulator

UserEmulator [26] is an on-device emulator for Symbian developed by Orange. It currently runs on most Symbian S60 3rd Edition and Symbian^1 handsets [26] . With UserEmulator you can simulate user activity by generating key presses and perform random stress tests. The scripting language is related to *test verbs* and very straightforward. In many ways UserEmulator is similar to TestQuest. UserEmulator can be a useful tool when the tester needs to automate functional, stress, or regression tests [26]. UserEmulator uses XML encoded test scripts to describe events, and can also use XML encoded files containing recorded user input.

Using UserEmulator the tester can easily write or record the tests that she/he wants to perform live on the device. There is also an option to run a random stress test on an application using a built-in class. UserEmulator will automatically take screenshots of panics and log crash events and other system details when running these random tests. Using UserEmulator on touch screen devices is difficult since you cannot generate a specific key press, since there are no keys, but only pointer events at different locations on the display. This leads to the problem that you cannot use the same test script for multiple phone models; thus you basically have to create a script for each touch model - unless they are very similar.

Despite this drawback, be it major or not, UserEmulator should be a very useful tool for automating tests. One advantage is that you do not need a connection to a PC to run the tests; this is due to the fact that UserEmulator runs directly on the device. UserEmulator is installed on the device and then scripts can be imported and run instantly. Below is a short fragment of a UserEmulator XML script:

```
<action>
  <name>Orientation</name>
  <type>orientation</type>
  <params>portrait</params>
</action>
<action>
  <name>Pause</name>
  <type>wait</type>
  <params>1004711</params>
</action>
<action>
  <name>key</name>
  <type>keypress</type>
  <params>0</params>
  <keys>LSK</keys>
</action>
```

This script fragment sets the screen orientation to portrait and then waits for 1004711 milliseconds after which it presses the left soft key (LSK), that is, the left key underneath the display in most cases. There are other tools available from the Symbian foundation, but none offer test automation. One of the official Symbian tools is the STIF Test Framework, a toolkit for test case implementation and test case execution [27].

4.2.3 Digia UsabilityExpo

The Finnish software company Digia Plc. previously had a testing suite called QualityKit that included the test automation program AppTest. Digia QualityKit was cancelled in 2007; instead the company has developed a separate application for each part of testing. Their UsabilityExpo is a tool for doing usability tests on mobile phones. Unfortunately, UsabilityExpo does not offer test automation, but simply provides a user interface and recording of tests. Other features of UsabilityExpo include [28]:

- Logging events and key presses in Microsoft Excel spreadsheet files
- Displaying a real-time image of the phone's display on your PC
- You can use you own keyboard and mouse to control the phone device
- You can record video clips (with an audio track and comments) and take screen shots
- You can connect to the mobile phone device using USB, WLAN, or Bluetooth interfaces (if available)

UsabilityExpo does not quite meet the requirements for the testing software that OptiCall needs, hence it is not a candidate for our testing. UsabilityExpo mainly offers a way to use a cellular phone through a GUI on the PC, but does not offer any real test automation functionality which is the main feature that is required for regression testing. It should be noted that with this connection testers could feed the device with scripts from the PC. However, UsabilityExpo does not offer the scripting functionality that is desired for this thesis project. UsabilityExpo could possibly be used as a tool when performing testing using different software in the future.

4.3 Remote Device Solutions

There are a growing number of remote device access (RDA) solutions on the market. Some of these solutions will be described in the following paragraphs. These services can be free of charge for developers, but mostly they are provided as international subscription services. What the services does is to give you access to mobile devices through a graphical interface via the internet. The devices can then be controlled over the link via a browser window.

Different services offer varying amounts of functionality, but in all services you can install your applications by uploading the executable and then test the application on the devices as you wish. In many cases the services include both an administrative page for reserving timeslots on the devices and a web interface that allows access to the physical device. The web interface is connected to a device server which in turn is connected to one or more of the devices through proxies.

4.3.1 Digia Remote Phone Management

Digia Plc. has developed a solution called Remote Phone Management (RPM) that enables you to test your applications remotely. [29] Both Nokia and Samsung are currently using this solution; under the name Nokia Remote Device Access (RDA) and Samsung Lab. Both are free of charge if you are a developer registered in the respective community. The limitations on the services are speed and the lack of call-out functionality, which makes this uninteresting for testing the part of OptiCaller which demand access to GSM networks.

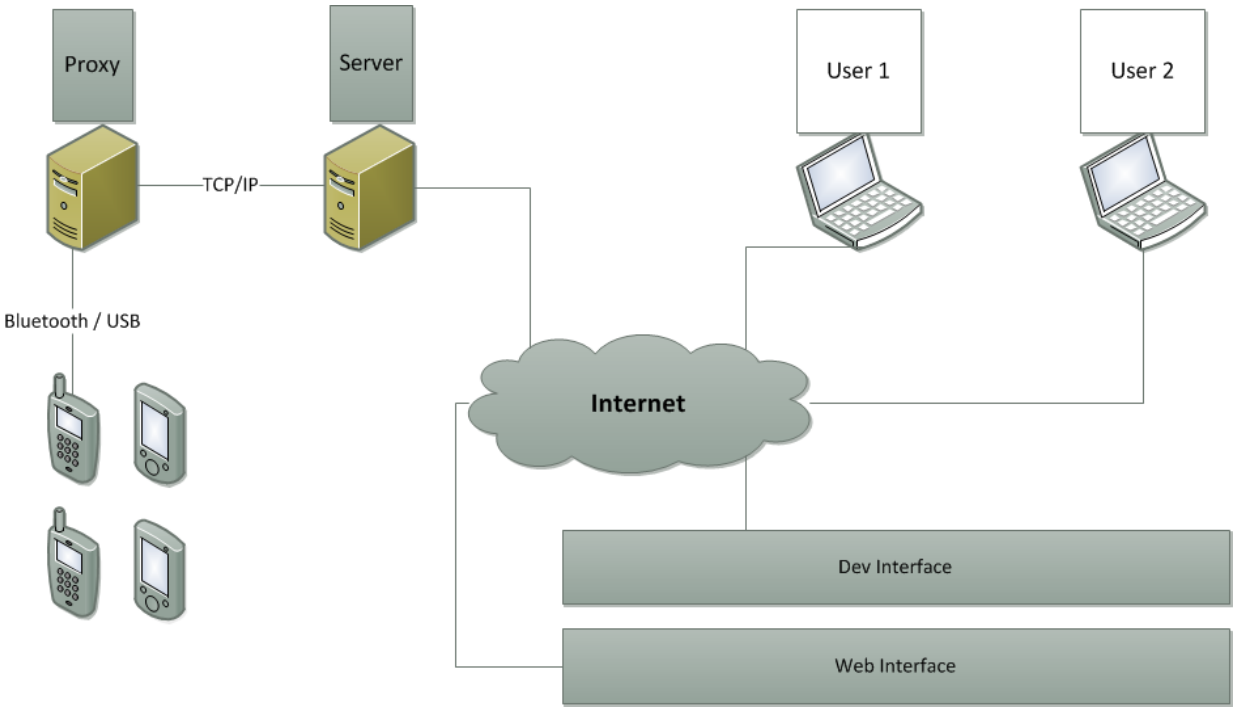

Figure 1: An illustration of the Digia RPM network topology. Two clients are connected; one is using the developer UI and the other the Web UI for reservation of remote devices.

4.3.2 DeviceAnywhere

DeviceAnywhere is a major player in the remote device business and offers their services to Nokia and Symbian developers at a discount. Their solution is a premium service with a monthly fee that includes a fixed number of testing hours on a variety of devices. Unlike Digia RPM, who gives the user access to only some of the device's functionality, DeviceAnywhere's Test Center provides comprehensive support for all hardware functions of the device.

This enables the tester to do anything that they could do with a physical device, including touch screen, multi-tap, pinching, swiping, accelerometer, mute, power on/off, open/close cover, backlights, vibration, volume increase, hardware compass, camera, battery, etc. The drawback is that the device locations are outside Scandinavia, in fact most are in the U.S. and UK [30]. This is a major disadvantage as the testing becomes very expensive if the application is to setup calls to the PBX testbed located in Stockholm. Furthermore, we cannot use their service to test the application with a Swedish operator, which is the company's primary target.

At this stage it is not feasible to conduct tests using testbeds in other countries due to distance and cost. OptiCall Solutions AB does not have the resources to set up testbeds in UK locations, hence the distance issue. The location issue is not insignificant; if testing on a device located in the UK, the response time of the device will not be sufficient for effective testing. From experience, it is not anywhere near what we would expect from a device in our own labs. Simply locating a PBX abroad is not sufficient as testing currently also requires an additional mobile phone with a SIM card to fully test the application. Both of these cellphones could of course be located in a remote test lab, however they would need to call each other through a PBX located in Sweden, thus the costs would be very high for each test run.

4.3.3 Perfecto Mobile

Similar to DeviceAnywhere, Perfecto Mobile offers a subscription service with additional per hour fees. Using their service you can utilize all the functionality of the devices including call setup, which is important for testing OptiCaller. Perfecto Mobile has their labs in France, the US, Canada, and other countries, but does not have a lab in Sweden. Similar disadvantages as with DeviceAnywhere apply when using the solution that Perfecto Mobile offers.

4.4 Symbian Signed Test Criteria

According to Nokia: *"Symbian Signed is a program run by Symbian for and on behalf of the wireless community to digitally sign applications which meet certain industry agreed criteria. The 'for Symbian OS' logo is awarded to applications that are Symbian Signed."* [31]

If an application is Symbian Signed, thus means that the application behaves properly on a specified phone running Symbian. To become signed the application has to meet a number of criteria, called the Symbian Signed Test Criteria. The compliance testing is done by a test house approved by Symbian, such as Sogeti. It should be noted that signing is not a guarantee that the application will always work, nor is it any type of quality assurance. Signing simply confirms that the application responds correctly to some specific events. Thus, Symbian Signed does not guarantee that the application is bug-free or even useful to the customer. The tests that an application must pass in order to become Symbian signed include verification of correct IDs, installation, start and stop behavior, auto start, and assuring that the application does not disrupt calls, text messages, or key applications on the device.

The Symbian Signed Criteria are important for Symbian developers and these criteria should be considered when developing an application. Thus as part of our testing we should ensure that suitable tests are included in order to check that the application does not violate the criteria, as this would probably disqualify the product from being signed. For OptiCaller it is desirable to become signed, as this removes the requirement for International Mobile Equipment Identity (IMEI) registration. Unsigned client software cannot be installed on any Symbian device unless it is self-signed with the IMEI number belonging to each device that the software will be installed on. This procedure is very time-consuming and greatly complicates distribution of the software.

5 Related work

In 2005, Jutta Jokela performed an investigating study of testing software for Symbian [32]. Her findings were interesting, even if the software she tested and researched is now deprecated. Jokela tested Digia's AppTest, Mobile Innovation's TRY, and the Nokia Test Suite (NTS) among others. Unfortunately, all of these are now obsolete and unavailable. NTS was initially created for the predecessor to Symbian Signed (called "Nokia OK") and was compatible with the earliest Symbian S60 handsets. According to Jokela's report, TRY had a lot of problems with errors caused by difficulties in separating which program is foreground and which is background.

Jokela concluded her report with: "*There are several testing tools available for the Symbian market but according to our research it seems that there is no fully covering product available in the market.*" As most of the tools developed for early Symbian versions are obsolete, today there are new solutions such as TestQuest, as well as many test programs from the Symbian Foundation that serve different testing purposes.

Antti Kervinen, et al.[23] performed GUI testing on Symbian using Intuwave m-Test. Intuwave does not exist anymore and the product is no longer available. Kervinen et al.'s report is about model-driven testing; hence it does not address the type of specifications that I will use. They used several tools available at the time, along with a label transitioning system to test every possible execution path in a system. Their method is interesting, but for the case of OptiCaller it is not relevant. OptiCaller has a few critical functions and it was concluded that my testing efforts should be focused on these functions.

6 Discussion

The goal of this thesis project is to create a template for testing the OptiCaller application, perform the tests, and evaluate the results. The testing should be as automated and straightforward as possible. The test strategy is mostly focused on functional testing, with some smoke/regression testing. Also a bug tracker must be set up to handle the test results and to provide some new features. The testing will cover the most important functions of the application and assess whether this functionality is reliable.

6.1 Scope

The thesis project is largely based on the requirements of the company. At a meeting in Gothenburg with the company management and the developers of OptiCaller it was decided that functional testing was the most pressing issue. The reason for this is that the client application is believed to be very stable and a version 2 will soon be released. Hence my first focus is on functional testing; i.e. to see that the product does what the product specification says and consequently to minimize the number of dissatisfied customers. The idea is that functional and regression tests will be conducted according to a test plan that includes specified acceptance criteria for each and every test.

Before this step some initial validation tests were needed to validate the program's input controls. The test steps will be performed on a device under test using a key press emulator, such as UserEmulator [26] (available from Symbian Foundation). Despite the drawbacks of GUI testing, discussed in section 4.2, I believed that it was possible to utilize the GUI in automatic testing.

In addition, since all users will be using the GUI to invoke the functions of the application this means that providing inputs to the application matches what the user might do and thus does not require modifying the program's flow of control. However, as we have not focused on coverage testing, these tests do not represent every possible input that a user might give; hence there will be gaps in the test coverage which users might encounter.

6.2 Method

The most straightforward way of testing (which was also recommended by experts at Cell Telecom Ltd. [33]) would be to use an automated GUI testing application in combination with some type of log file analysis. This way immediately after the tests the tester can examine the testing results without having to observe the process closely in real time. One approach is to create an automated script that simulates key-presses together with a logging process in the OptiCaller client software. The backend of this solution would be a program that analyzes the log file and gives feedback to the tester. This feedback should be easy to interpret and clearly present the most important issues.

The functional testing should be as automated as possible, if possible, and the same should be true for regression tests. The regression tests will be a subset of the functional tests. After performing the tests, the results of testing will be evaluated and improvements to the testing process will be proposed.

An additional task is to deploy some kind of system for reporting bugs that are found during testing. This bug tracking system should be integrated with the testing process and result analysis.

The only problem with this solution is that it is not feasible to apply key-presses for testing with touch screen devices which have unique inputs and no physical buttons. Despite this, it is actually possible to perform some tests, but the inputs must be given in terms of positions on the screen instead of actions on physical buttons. These soft keyboard events can be recorded and then replayed if the test is to be run multiple times.

7 Method

This chapter will introduce the testing method along with the different tools that were used. Also, the application specifics and interface will be presented.

7.1 Logging

The client software reaches some different states that occur when controlling the flow of the program. These states can be useful during logging since they not only are used in the control mechanism but also the transition between states can be used to describe the control flow in the program. These states can be divided into two groups: Call states (see Table 1) and Program states (see Table 2).

Table 1: Call states of OptiCaller

State name	Description
StatusIdle	Call-status when no calls are active
StatusDialling	Call-status for when a call is being made
StatusConnected	Call-status for when a call is connected
StatusHangingup	{not used}
StatusRinging	Call-status when the phone is ringing

Table 2: Program states of OptiCaller

State name	Description
Idle	*When the client is running but not active*
CallIntercepted	*When an outgoing call is aborted so the client can set up it anew*
HandlingIntercept	*Status for when the intercepting is being done*
DialingDtmf	*The software is dialing a call-through call*
ConnectedDtmf	*When connected to the PBX*
DialingDC	*When a direct call is made*
ConnectedDC	*When a direct call is setup*
WaitingCB	*When the software is waiting for a call back*
ConnectingCB	*When a Call back is waiting to be set up*
ConnectedCB	*When a Call back is set up*

It would be possible to log when the program enters/exits each state producing a very exhaustive live log of the program's flow. However, if you just want to confirm that a call is set up correctly, the only states that are important are *Intercept*, *Dialing*, and *Connected*. That is, we can ignore the *StatusIdle*, *Waiting*, *StatusRinging*, and *Connecting* states.

Different types of calls go through different flows in the code and thus have a different sequence of states. This sequence is matched in the analysis web application so the analysis software can distinguish between the different call types. If the sequence is incomplete or broken we can conclude that the test run was unsuccessful. The output of the test, the log, may look something like the following in the case of a Call back call:

CallIntercepted

Idle

CB

HTTP

+4670123456

Answering

Connected

Idle

First the software intercepts the call setup, and then a call back request is sent through HTTP to the PBX. The number we want to call is +4670123456. When the call is received we also output that we are answering and finally report that the call is set up. Another example is call through call setup that uses a slightly different flow:

CallIntercepted

Idle

AA

CT

Calling

0811223344

Connected

DialingDTMF

0811414

The call is initially intercepted by the software and then the software is idle until the call method dialogue pops up. This dialogue will then prompt the user to select a call method. This is when AA is logged (Always Ask, see Figure 2).

The test script then continues with a selection of Call through and the call is made. We can see which number is called (081123344) and also we can see whether a connection was successfully set up with the PBX. When a connection is set up, the software transmits the DTMF signals indicating the callee/recipient and the PBX sets up this call. No more output is generated since after this step the call setup success does not depend on OptiCaller but rather the success of the call set up process of the PBX.

Figure 2: The Call Method dialogue as shown in OptiCaller

7.2 Test Scripts

The GUI testing scripts will be executed with UserEmulator by Orange, see Figure 3. Since the format of these scripts is XML, it is relatively easy to create new scripts and to modify existing scripts. While a script may work perfectly on one phone model if we switch to testing of another model that uses a different set of buttons, then the test team might need to slightly adjust the test script. Furthermore, it is possible to combine a set of different test cases into one that covers more functionality.

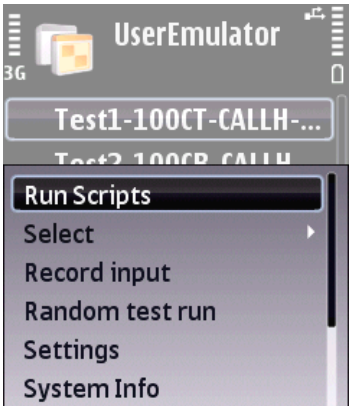

Figure 3: The interface of UserEmulator as shown on a Nokia E66

7.3 Test Analysis

The backend of the system will be a web page that analyses the output, i.e. the log file, and displays the results visually. The requirements for the complete system are:

I. Easy to select test case. Minimum work should be required to start a test run.
II. Easy to extract and submit test data. The extraction of the test data should be easy to perform and not include too many steps. Furthermore, the method of submitting test data to the web application for analysis should be straightforward.
III. Easy to select test data in the web application. The GUI should be easy to manage.
IV. Detailed and clear information about the test outcome
V. Local storage of results
VI. Possibility to transfer results to issue-tracking software installed on one of the machines. This software is also part of the solution, however will not be mentioned other than in passing as is not a very major contribution.

The prototype solution that was built was coded in PHP making it very lightweight and able to run on a wide variety of platforms.

Requirements III, IV, V, and VI were all fulfilled.

The web interface makes it easy to upload test logs from a local drive as well as naming them and storing the information from the test just run (see Figure 4). For comparison, another log file is necessary. This second log must always have the expected result and must be updated if the process changes. Using the interface shown in Figure 5 we can generate listings of both file types and the additional information available per file.

Figure 4: Here a test log can be uploaded from a local storage or a connected device

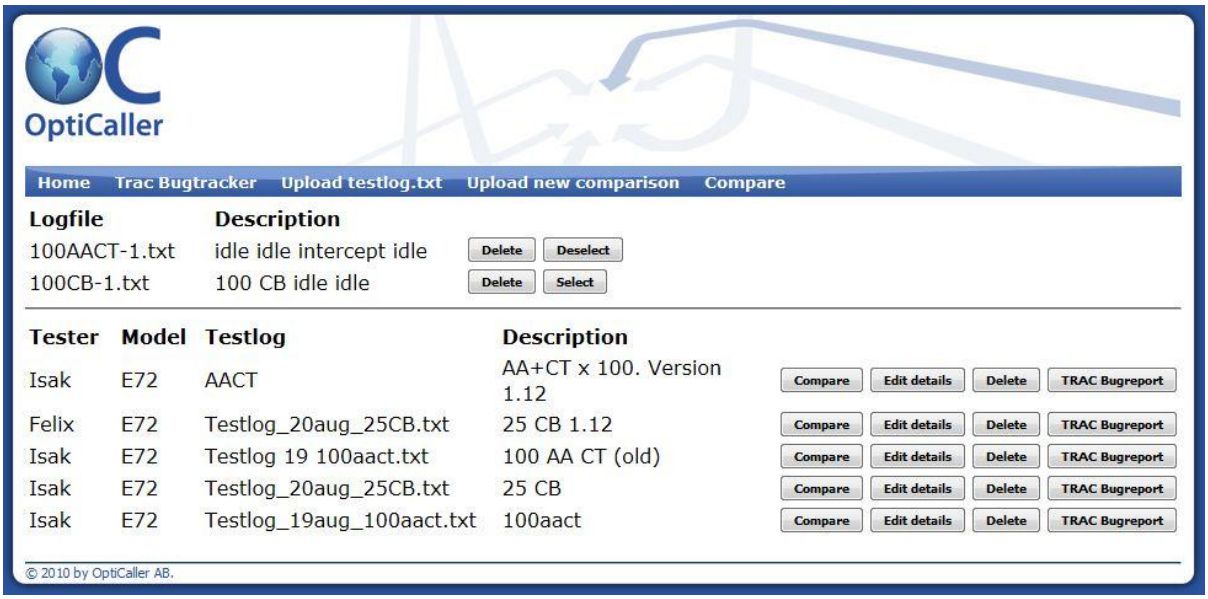

Figure 5: In another view it is possible to compare the two files and edit them

Using the resulting output (shown in Figure 5) it is possible to select the log files to compare. Therefore, if the tester wants to evaluate a recent test she or he selects the file that corresponds to this test and the comparison file that will be used to show how the test was expected to work. This comparison can be from the unit testing phase or from some other device that already passed the test. An example of the comparison results are shown in. There is also a link to the bug report file, or rather; the test can be reported directly through an HTTP call to the TRAC Bug tracker. Clicking on the report bug button will open a window with some data pre-populated to easily submit defects in the software that were found when testing.

Figure 6: Comparison logs can be uploaded for reference

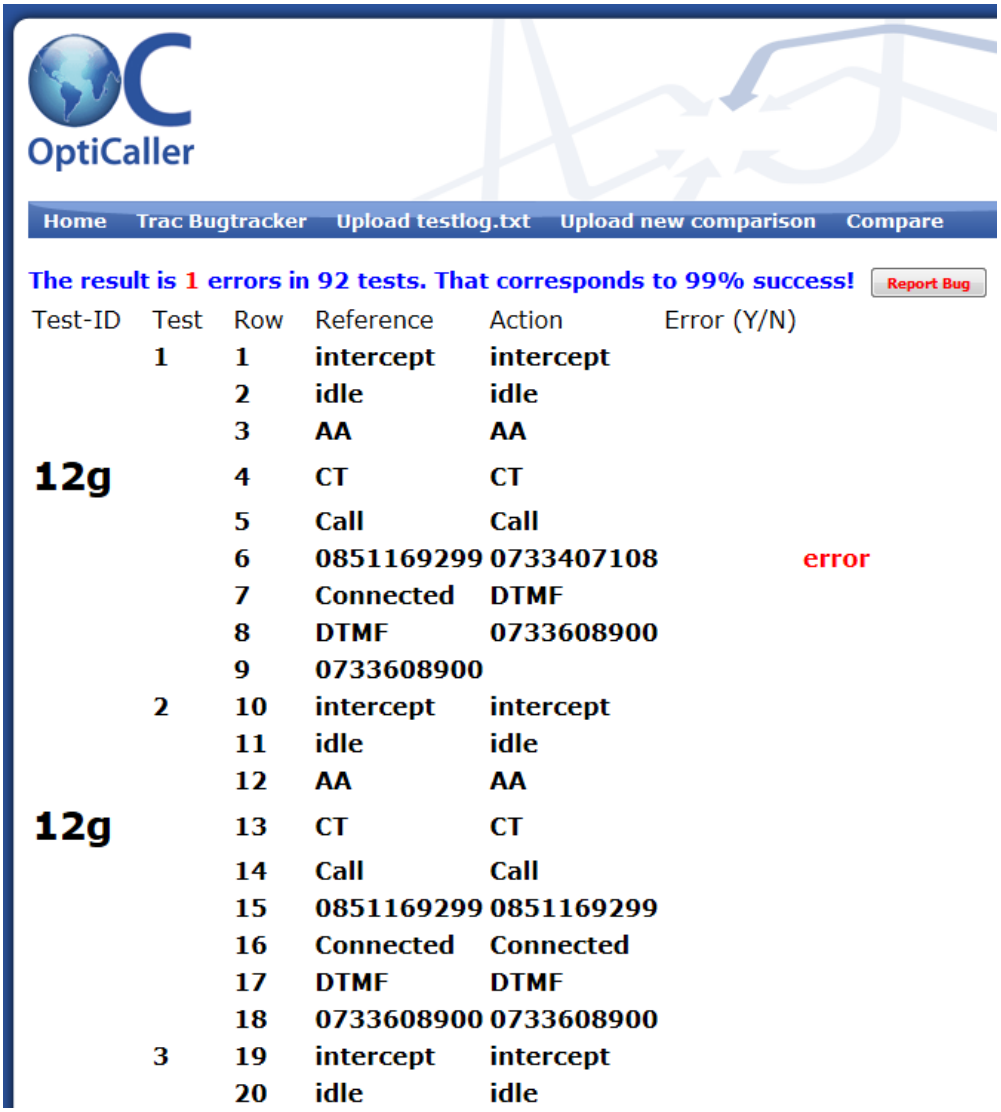

Figure 7: The comparison view. Here the tester can see the success factor as well as the actual logs. 12g denotes the type of test. As can be seen, this particular run had one error initially when setting up the first call.

7.4 Bug tracking software

As mentioned earlier, TRAC bug tracker was chosen for the purpose of tracking bugs and results. This was mostly because TRAC is very configurable and has many options. Also, it was easy to set up and to create projects within. TRAC bug tracker also had a plug-in that allowed bugs to be added through HTTP GET from another webpage. This functionality allowed ease of work when registering issues and analyzing results.

8 Test specification

One of the most important steps in this master thesis project was to write the test specification. This specification is each time a test is run to verify the functionality of the device. Part of the test specification is shown in Table 3 below, while the whole test specification is given in Appendix A: Test Strategy.

Table 3: A part of the Test specification listing the first couple of test cases.

Test Table v1 Name of tester Phone model
Date / 20__

Nr	Case Description and Goal	Initialization	Time (h)	Expected Results/ Acceptance Criteria	PASS/FAIL	Issues and faults
1a	Install OptiCaller	Install OptiCaller from USB or Bluetooth.		OptiCaller icon in applications folder, install success		
1b	Run OptiCaller	Start OptiCaller using the icon in Applications.		OptiCaller starts		
2a	Get license	Insert provision settings and click “Get License”.		Product gets licensed.		
2b	Change a setting	Change a setting entry anywhere, e.g. “Call Back 2”		Settings changed		
2c	Revert Settings	Click the button Options > Setup > Restore Settings to revert		Settings reverted to original…		
3a	Turn on Transparent mode and exit	In Options > Setup > Operational Settings. Select Transparent Mode: ”On”. (default) and exit with the Exit button.		AA icon displayed and OptiCaller in the background.		
3b	Turn off Transparent mode and exit	In Options > Setup > Operational Settings. Select Transparent Mode: ”Off”. (default) and exit by clicking Options > End Program.		OptiCaller doesn’t run in background anymore.		

9 Analysis

This chapter will introduce the analysis of the solution in terms of fulfillment of requirements and test results. Also, the proposed solution is presented.

9.1 Solution evaluation

To evaluate the test solution some testing is needed in order to see to what degree the solution satisfies the requirements. It was required that the solution fulfills the requirements given in section 7.3. Below we consider each of these requirements and comment on whether the solution satisfies this requirement:

I. Easy to select test case

It was easy to select the test case script in the on-device application, and this test is automatically run until it finishes. However, it would be better if the selection of test case could be done remotely and the test results uploaded directly to the analysis server (see the next numbered requirement).

II. Easy to extract and submit test data

With the chosen approach, it is necessary to extract the test file from the device. After that the file needs to be uploaded to the analysis server where it will be processed. A script could easily be created to perform this upload, but was not created which cripples the solution slightly and adds additional workload for the human tester. Adding this script should be done as part of future work.

III. Easy to select test data in the web application

The web application is friendly in the sense that all inputs are clear, as long as the user knows the distinction between a comparison log (earlier referred to as a comparison file) and a test log. The selection of data is easy, but the operation of the web application depends on the user's input. The design of this interface could probably be more user-friendly, and it is a bit too easy to delete test runs. Also, the database design could be improved.

IV. Detailed and clear information about the test outcome

The test outcome is described in great detail with multiple measurements and mappings to the test types. Also this data is easily transferred to an issue tracking system with a single click.

V. Local storage of results

All files are stored in a database and accessible through the web application. Also, results can be easily stored in the TRAC issue tracker.

VI. Possibility to transfer results to issue-tracking software installed on one of the machines.

As been mentioned above with respect to requirements in IV and V, the results can easily be transferred to the bug tracking system. However, the output must first be analyzed in the web application.

9.2 Requirement evaluation

In addition to the requirements on the solution, the following requirements concerning the overall test flow were satisfied:

1. Bugs in the actual technical mechanisms that could be encountered by the customer are found.

Indeed, the log file analysis solution will in most cases find the errors that are directly connected to the mechanisms of call setup and interrupts in the system. For other operating system issues it is not as easy to detect the issues, as there is no automation to compare the visual appearance and other less critical parts with the expected visual appearance. Overall, most issues are dependent upon the specific operating system on which the application runs, if this OS is unstable, then there will be errors in the application's execution.

2. The tests can be automated.

The solution that was used benefitted from automated tests that do not need any human interaction at all.

3. The number of automated tests to perform can be set.

With this solution it is possible to list the test sets that need to be run, and also to set the exact number of tests in each set that are to be run. Unfortunately this has to be specified in the XML file, but in the future a tool should be written to make this easier.

4. Bugs can be reported easily using the analysis application.

Once the test results have been uploaded and imported into the analysis tool, it is easy to report these results through a direct link with an issue tracking tool developed specifically for this purpose. The bug description and test result are sent though HTTP GET request to the TRAC server where the report can be reviewed and stored.

However, one requirement was not met:

5. The application can be run directly on the device, but be started and monitored from a laptop.

This requirement was not fulfilled, as no way was found of achieving this with the chosen approach. Additionally, no solution to this could be found within the project's time period. The application can be run directly on the device, but starting and monitoring the testing process remotely was not realized.

9.3 Notes

All tests were run on Nokia E66 or alternatively Nokia E72 that comes with a QWERTY keyboard. Also, a touchscreen device was tested, Nokia N97 mini, but extensive testing with this device was ruled out due to the difficulties of creating test cases and the inconsistent test results that I observed. It was realized early in the project that touch screen devices cannot be tested in the same way as other devices. The selected method of testing was not suitable for testing touch screen devices.

9.4 Test results

Based on the testing performed, we can draw some conclusions from the results. When a bug was found it was most often at the start of the testing, when setting up a call. This is shown in the column of Table 4 for AACT, which sometimes encountered a problem when opening the menu dialogue to choose the calling method. Out of 880 tests run, 9 tests failed; which means a success rate of 0.99, i.e., 99% of the tests where successful.

A summary of these test results are also shown in Table 4. Not all functions were actually tested, just enough of them to prove that the test method was working for this application. Based on these results I claim that this method is suitable for automated testing of functions that are otherwise very time consuming to test.

For comparison, to run one of these tests manually and analyze the result of the test can take between one and five minutes. Automating the same test might take up to fifteen minutes, but the manual work is eliminated for all subsequent tests of the same functionality. Actually running the automated test including analysis takes about half the total amount of time as for running the manual test. Thus the time to automate the test is rapidly amortized it the test is run more than a small number of times. As it is very desirable to perform regression testing after making changes to the software, it is clear that automated testing has a high payoff.

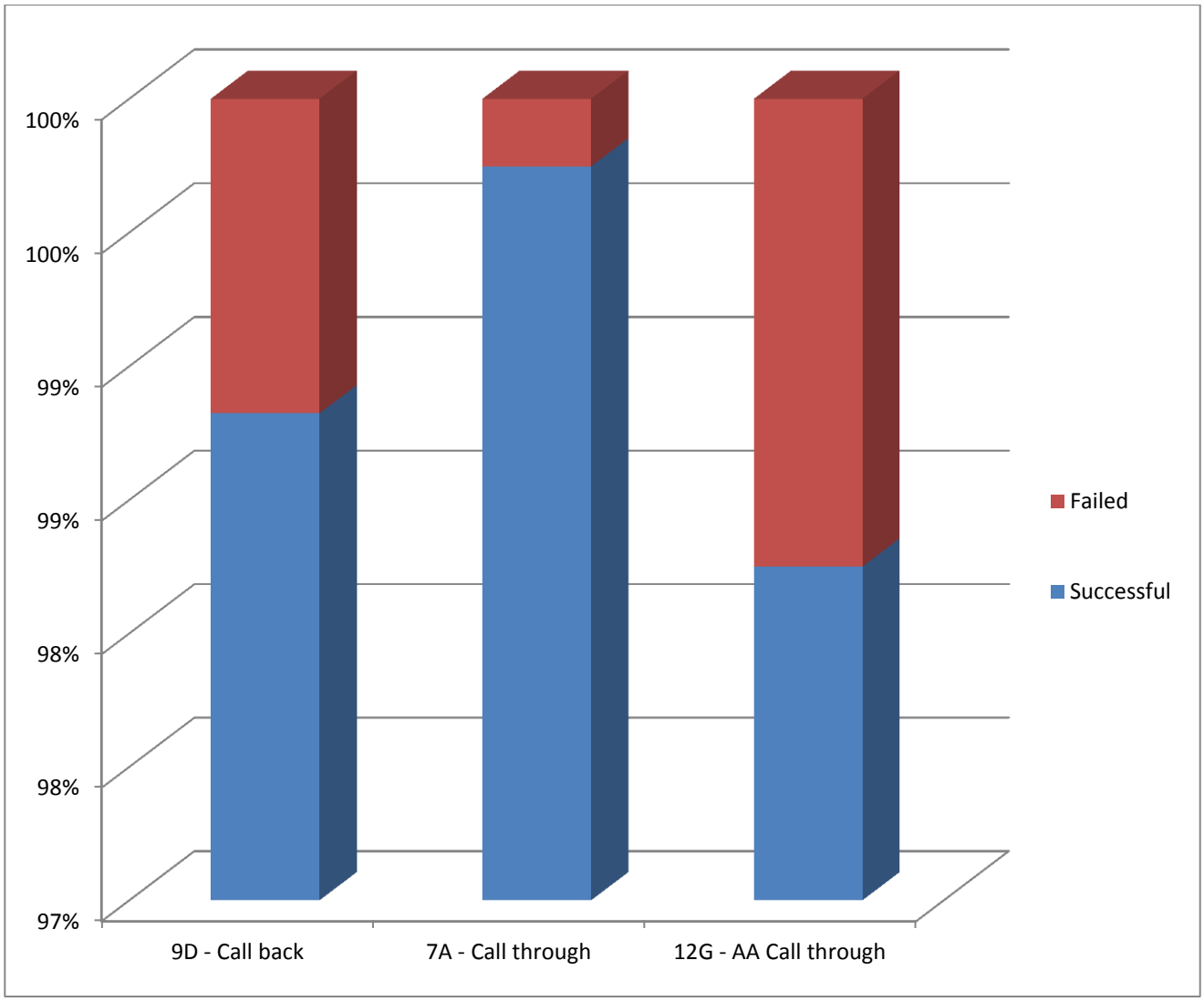

Figure 8: Test results presented as a bar chart.

The results are shown in Table 4 and plotted in Figure 8. The plot has confidence interval 0.978 at confidence level 0.95. The variance is expected to be less than 0.01. The testing includes the sum of all testing of both Nokia models E66 and E72.

Table 4: Test results

Test case	Successful	Failed	Number of tests
9D - Call back	84	1	85
7A - Call through	394	1	395
12G AA Call through	393	7	400
Total	871	9	880
Rate	**0.99**	**0.01**	1

9.5 Proposed solution

It can be argued whether it is really feasible to automate mobile application testing, as this testing is often based on use cases. However, the results have shown that a good framework for testing includes tracking status and results. Test automation is also necessary when load and stress testing the application.

9.6 Acceptance

The original goals of this master's thesis project were to evaluate testing methods, write specifications, and finally execute and evaluate the testing method on different phones. It was also necessary to implement an infrastructure for handling the test reports and to facilitate dealing with any issues reported.

As for the first goal, the testing methods were evaluated and a suitable method was chosen that could deliver the desired solution according to the project's requirements. After this test specifications and templates were written that covered the necessary testing and automation of steps. These tests were then executed with success and the result was indeed positive; the automation was not as difficult as expected and running the tests was easy.

Overall, most of the tests were successful (as shown Table 4) and manual handling of the device under test was not necessary when testing the application. Furthermore, automation of testing reduced the time spent testing by the developers. Finally, an issue tracking system was implemented to receive reports from the analysis system implemented as part of the solution. The chosen software was easy to integrate with the analysis tool, and could also facilitate other parts of the development cycle, such as keeping track of mile-stones, application versions, and general defect tracking.

10 Conclusions

This chapter will summarize the conclusions of this master thesis project as a whole, and also suggest some future work and improvements that could be made. The chapter concludes with some reflections on the project.

10.1 Conclusion

I am quite satisfied with the project as such and I feel that the project's goals were met. Despite the results described earlier, testing applications on mobile devices is not straightforward, and I cannot claim to have shed any light upon how to lessen this complexity. As the technological revolution continues and the devices and software become more and more complicated, testers will struggle to keep up. It is my belief that both testing and development will be more and more built around use cases in the future, thus test cases will evolve together with the development of the application. However, automation of these test cases will not be easy with the increasing use of touch screens and physical input (such as gestures).

To others who venture into the testing arena and especially those considering automated testing, I think the path to success is to find a niche for themselves. Because the set of all possible devices is so large, it is important to find a subset of devices that are most relevant. I would also recommend that testing be conducted through a computer to lessen the manual interaction and workload.

If I were to do a similar project again, I would have demanded a clear set of requirements to start with, and then used these requirements to identify software that I could build upon. Possibly I would have avoided logging and looked at more direct methods of retrieving test output to analyze. A clear reason to avoid logging is that it requires that the logging functionality be built into the code. This adds to the code's complexity and also introduces a risk that there is a change in the behavior when the logging is turned off.

10.2 Future work

Since there are lots of tests it is useful if the other side of the call is a SIP soft phone connected to a local SIP server. A SIP server already exists in the testbed PBX and could make testing much more efficient. Also, it is possible to minimize the costs of calling from the PBX with the current setup, as the costs for calls from the trunk/PBX are quite large each month and this cost should be minimized.

As described in section 9.1 there are many parts of the testing process that could be further automated, these steps should be automated as part of future work. Symbian is a finished chapter for Nokia; however, there are similar applications for Android, Windows Phone, and Apple's iOS that could benefit from similar investigative studies and testing. Also, the logging process could be extended to make it clearer when looking for the cause of error. Timestamps should definitely be introduced in the logs for this purpose. The analysis tool could also be adapted to take delays and other test environment issues into account.

10.3 Reflections

This thesis project provides guidance and advice to mobile application developers and testers when planning and executing automatic tests on mobile devices. The proposed solution points out the possibilities for testing, especially outside of large corporations. The benefits of these solutions and conclusions can be meaningful to both small scale developers and other who venture into the mobile market in terms of the potential economic benefits.

As the market demand rises, corporations and entrepreneurs will benefit from automation of testing when striving to deliver the best service to the end user without needing any feedback, or doubtfully ethical anonymous information, originating from the user's device. During the work with this project, only internal phone numbers have been logged and the application is very restrictive when collecting user data.

When introducing logging into an application it is important to remember the issues associated with maintaining the privacy, since if a malicious attacker were to enable logging they could collect considerable private information. OptiCall Solutions' aim is to achieve the Symbian Signed criteria for the application, both as a sign of user friendly design, but also to protect the privacy of the end user. Much work is left to be done within this area; the most important next step is to integrate this mobile application with SIP systems.

It is my clear opinion that OptiCall Software greatly benefitted from the project outcome and learnt many things in the process. A test strategy was introduced, and the test cases were formalized. Also, the analysis tool made it easy to automate testing and validation of results. Finally, the issue tracking software installed will continue to benefit the company as they now can get a better overview of defects and results.

References

1. **TechCrunch.** [Online] 2012. http://techcrunch.com/2012/05/11/this-is-what-developing-for-android-looks-like/.
2. Press Releases. *Symbian.org.* [Online] 02 04, 2010. http://www.symbian.org/news-and-media/2010/02/04/symbian-completes-biggest-open-source-migration-project-ever.
3. **Sun, Tao.** *Developing a Mobile Extension Application: OptiCaller Application and Provisioning System.* Kista : KTH ICT, October 2009. TRITA-ICT-EX-2009:177.
4. **Kumar, Shiva.** An Effective Handbook for Implementing Test Strategies. *Think Business Networks.* [Online] July 2001. pg.5. http://5676430411356704223-a-onestopsoftwaretesting-com-s-sites.googlegroups.com/a/onestopsoftwaretesting.com/home/public-osst-files/AnEffectiveHandbookforImplementingTestStrategies.pdf.
5. Defining a Test Strategy. *Software Testing Times.* [Online] April 2010. http://www.softwaretestingtimes.com/2010/04/defining-test-strategy.html.
6. **IEEE.** IEEE Standard Glossary of Software Engineering Terminology. s.l. : IEEE, 1990. p. 74.
7. Software Testing. *Wikipedia.* [Online] 2010. http://en.wikipedia.org/wiki/Software_testing.
8. Test Cases. *Testing Excellence.* [Online] December 7, 2008. http://www.testingexcellence.com/test-cases/.
9. **Berger, Bernie.** The Dangers of Use Cases Employed as Test Cases. *Test Assured Inc.* [Online] 2001. http://www.testassured.com/docs/Dangers.htm.
10. **TestAssured Inc.** Dangers of using use cases as test cases. [Online] 2003. http://www.testassured.com/docs/Dangers.htm.
11. **Pan, Jiantao.** Software Testing. *Carnegie Mellon University.* [Online] Spring 1999. http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/.
12. **Chandran, B and Pai, I.** Testing Techniques for Mobile Applications - MangoSpring Technology Pvt Ltd. *IndicThreads.com Conference on Software, Pune, India.* [Online] 2010. http://www.slideshare.net/indicthreads/indic-threadsqualityconference2010-testingtechniquesformobileapplicationsfinal.
13. **Williams, Laurie.** Testing Overview and Black-Box Testing Techniques. [Online] 2006. http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf.
14. Wikipedia. [Online] http://en.wikipedia.org/wiki/Fuzz_testing.
15. **IEEE.** *IEEE Standard 610.12 IEEE Standard Glossary of Software.* 1990.
16. Wikipedia. [Online] http://en.wikipedia.org/wiki/Boundary_testing.
17. **Parekh, Nilesh.** Software Testing - Stress Testing. *www.buzzle.com.* [Online] 2000. http://www.buzzle.com/articles/software-testing-stress-testing.html.
18. **Andrews, James H.** *Testing using Log File Analysis: Tools, Methods and Issues.* Honolulu, USA : IEEE Computer Society, 1998. Proceedings of the 13th IEEE Conference on Automated Software Engineering. pp. 157-166. ISBN: 0-8186-8750-9.

19. **SmartBear Software.** Why automated testing? [Online] 2012. http://support.smartbear.com/articles/testcomplete/manager-overview/.
20. **Liu, Yike.** *WCDMA Test Automation Workflow Analysis and Implementation.* Kista : KTH ICT, 2009. TRITA-ICT-EX-2009:6.
21. **Holtz Betiol, Adriana and de Abreu Cybis, Walter.** *Usability Testing of Mobile Devices: A Comparison of Three Approaches.* [ed.] M.F. Costabile and F.Paternó. Brazil : Springer, 2005. In Proceeding of INTERACT 2005. Vol. Lecture Notes in Computer Science 3585, pp. 470-481.
22. **Li, Kanglin and Wu, Mengqi.** *Effective GUI Testing Automation: Developing an Automated GUI Testing Tool.* s.l. : Sybex, 2004. ISBN: 0-7821-4351-2.
23. **Kervinen, Antti, et al.** Model-Based Testing Through a GUI. *Formal Approaches to Software Testing.* Berlin : Springer, 2006, Vol. 3997, pp. 16-31.
24. **Bsquare.** Countdown. *Products: Countdown.* [Online] 2010. [Cited: November 7, 2010.] http://www.bsquare.com/countdown.aspx.
25. TestVerb Technology (TVT) datasheet. *TestQuest* . [Online] 2010. http://www.testquest.com/download.cfm?oid=4325&product=3487.
26. UserEmulator. *Symbian developer wiki.* [Online] 2010. http://developer.symbian.org/wiki/index.php/UserEmulator.
27. STIF Test Framework. *Symbian Developer wiki.* [Online] 2010. http://developer.symbian.org/wiki/index.php/STIF.
28. **Digia Plc.** UsabilityExpo. *Digia.* [Online] 2010. http://www.digia.com/C2256FEF0043E9C1/0/405001348.
29. Remote Phone Management User Guide. *Nokia RDA.* [Online] Digia Plc., April 8, 2010. http://apu.ndhub.net/userguide/ch01.html#d4e12.
30. Mobile Application Testing. *DeviceAnywhere.* [Online] 2010. http://www.deviceanywhere.com/mobile-application-testing.html.
31. Symbian Signed End User Statement. *Symbian Signed.* [Online] 2010. https://www.symbiansigned.com/app/page/EndUserStatement.
32. **Jokela, Jutta.** *Evaluation of Testing Software for Symbian OS/C++ and Series60.* Helsinki : EVTEK Polytechnic, 2005.
33. **Cell Telecom.** *Cell Telecom home page.* [Online] 2010. www.cell-telecom.com.
34. **Zhang, Yingjun and Andrews, James H.** *General Test Result Checking with Log File Analysis.* 7, s.l. : IEEE Computer Society, July 2003, IEEE Transactions on Software Engineering, Vol. 29, pp. 634-648. ISSN 0098-5589.

Appendix A: Test Strategy

A.1 Test Strategy Document

This document is a tool for testing the OptiCaller client on mobile phones running Symbian. The document describes the steps in the process with prerequisites, objectives, the process of testing and the tools used.

Objectives: To test the OptiCaller client program and make sure it is stable. The approach used is manual functional and acceptance testing.
The only accepted outcome is 100% for each model with the latest OptiCaller client and firmware.

Preconditions: Symbian 9.1+ phone, UserEmulator, OptiCaller client with special logging activated, usb/datacable or Bluetooth connection.
Requirements for automatic testing: Number to be called is latest in call log. Settings correct and correct call method chosen.

Testing environment: Install both programs and insert the settings used in testing (CT, CB etc.). Test scripts located in {phonemem}/TestScripts

Tools: UserEmulator and the auto-test website

A.2 Functional Testing

Test Table v1 Name of tester Phone model Date /
20__

Nr	Case Description and Goal	Initialization	Time (h)	Expected Results/ Acceptance Criteria	PASS/ FAIL	Issues and faults
1a	Install OptiCaller	Install OptiCaller from USB or Bluetooth.		OptiCaller icon in applications folder, install success		
1b	Run OptiCaller	Start OptiCaller using the icon in Applications.		OptiCaller starts		
2a	Get license	Insert provision settings and click “Get License”.		Product gets licensed.		

2b	Change a setting	Change a setting entry anywhere, e.g. “Call Back 2”		Settings changed		
2c	Revert Settings	Click the button Options > Setup > Restore Settings to revert		Settings reverted to original…		
3a	Turn on Transparent mode and exit	In Options > Setup > Operational Settings. Select Transparent Mode: ”On”. (default) and exit with the Exit button.		AA icon displayed and OptiCaller in the background.		
3b	Turn off Transparent mode and exit	In Options > Setup > Operational Settings. Select Transparent Mode: ”Off”. (default) and exit by clicking Options > End Program.		OptiCaller doesn’t run in background anymore.		
4a	Screen orientation check	Rotate the phone 90 degrees with OptiCaller open.		OptiCaller looks the same but wide.		
4b	Icon orientation check.	Exit the program by choosing Exit in the (right) menu. Still 90 degrees rotated.		Icon moves to lower part of horizontal view.		
5a	Localization check	Change the phone language. Restart phone. (for WiMo change language in Options > Setup > Language)		Language of OptiCaller changed.		
5b	Auto Start test	Options > Setup > Operational Settings, Set Auto Start to On. Restart phone. (Symbian only)		OptiCaller starts automatically and the menu comes up.		

6a	Direct Call, select in menu	In Options > Setup > Operational Settings. Select call mode: “Direct Call”.		AA icon changes to DC		
6b	Direct call functionality	Perform an outgoing call. Answer on both ends (to make sure the call was set up.)		Call is setup and you can talk to each other.		
6c *	Direct Call and OptiCaller.	Perform an outgoing call without answering on the target phone.		Call signal goes through.		
6d	Making sure you can abort a call in DC	Perform an outgoing call and hang up immediately.		Call hung up.		
7a	CT, select in menu	In Options > Setup > Operational Settings. Select call mode: “Call Through”.		AA icon changes to CT		
7b	Make sure CT is working on both ends	Perform an outgoing call (and make sure it is interrupted by the software. Answer on the receiver side.)		Call through, dial tone heard and call setup. Showing correct phone nr.		
7c *	Make sure CT works.	Perform an outgoing call (and make sure it is interrupted by the software.) Hang up after dial tone is heard.		Call interrupted by server, dial tone heard after <30s.		
8a	Attempt to abort CT during initiation.	Try to abort the call with red button during call initialization.		Call hung up.		
8b	Attempt to make second call while CT is initializing.	Try to setup a second call while CT is initializing call.		Nothing happens.		
8c	Attempt to abort CT during DTMF.	Try to abort the call with red button during DTMF signaling.		Call hung up.		

8d	Attempt to make second call during DTMF sequence.	Try to make a second call during DTMF signaling.		Nothing happens.		
9a	Select Call Back	In Options > Setup > Operational Settings. Select call mode:"Call Back".		AA icon changes to CB		
9b	Select HTTP	In Options > Setup > Call Back Settings. Select Call Back Method: HTTP(S).		HTTP chosen and used when performing CB.		
9c	Test Call Back (HTTP) call functionality	Perform an outgoing call, answer on both sides		Call is interrupted by software, received by PBX within 60 s		
9d *	Test Call Back (HTTP) Call setup	Perform an outgoing call, hang up after dial tone is heard.		Call setup ok, functioning without errors.		
9e	Test Call Back (HTTP) call setup control	Perform an outgoing call, don't answer when called back.		The PBX doesn't call the receiver.		
10a	Select SMS	In Options > Setup > Call Back Settings. Select Call Back Method: SMS.		SMS chosen and used when performing CB.		
10b	Test Call Back (SMS) Call functionality	Perform an outgoing call, answer on both sides		Call is interrupted by software, received by PBX within 60 s		
10c	Test Call Back (SMS) call setup	Perform an outgoing call; hang up after dial tone is heard.		Call setup ok, functioning without errors.		
10d	Test Call Back (SMS) call setup control	Perform an outgoing call, don't answer when called back.		The PBX doesn't call the receiver.		
11a	Select HTTP again	In Options > Setup > Call Back Settings. Select Call Back Method: HTTP.		HTTP standard.		

11b	Attempt to abort CB during init.	Try to abort the call with red button during call initialization.		Call hung up successfully.		
11c	Attempt to make second Call while CB is initializing.	Try to setup a second call while CB is initializing call.		Second call created.		
11d	Attempt to abort CB during incoming call.	Abort the call with red button on incoming call from PBX.		Call dismissed. (normal behavior)		
11e	Attempt to make second call during incoming call.	Try to make a second call while getting an incoming call.		Second call created.		
11d	Attempt to abort CB during call.	Try to abort the call with red button when performing outgoing call setup after you have answered.		Call dismissed.		
11e	Attempt to make second call during outgoing call setup.	Try to make a second call after you have answered.		Nothing happens.		
11f	Generate generic error msg.	Attempt to make a Call back to a number that doesn't exist.		You receive a generic error message from server.		
12a	Select Always Ask	In Options > Setup > Operational Settings. Select call method:"Always Ask".		The Call Method icon changes to AA		
12b	Check that Always Ask works.	Perform an outgoing call.		The Always Ask dialogue is displayed. Nothing else.		
12c	Call cancellation before AA dialogue.	Abort the call with red button just before the AA dialogue is displayed.		Call dismissed.		
12d	Attempt to make second call during AA dialogue.	Try to make a second call when AA dialogue is shown.		Nothing happens.		

12e *	AA and DC	Make a call and choose DC. Hang up after callee starts ringing.		Direct Call performed. SIM nr displayed on callee side.		
12f *	AA and CB	Make a call and choose CB. Answer and hang up after callee starts ringing.		CB performed. PBX number displayed on both sides.		
12g *	AA and CT	Make a call and choose DC. Hang up after callee starts ringing.		CT performed. PBX nr displayed on callee side.		
13a	Edit White List	Edit the White List and add a number XX.		Number added		
13b	Select CT	In Options > Setup > Operational Settings. Select call mode: "Call Through".		CT Selected		
13c	Test White List function.	Perform an outgoing call to the defined number XX		A direct call is made.		
14a	Menu while in Call	Make a call and open up OptiCaller while in a call. CB/CT/DC doesn't matter in the testing scenario.		Call Services "menu" is displayed.		
14b	Test Call Services	Use one of the Call Services. Hang up.		Correct tones are sent.		
15a	MEX number	Select the correct MEX access number.		Number changed.		
15b	Edit MEX	Go to MEX Settings, add an entry such as Lunch ***23*3/*T*/#**		Entry added with correct properties.		
15c	Use MEX	Choose (click on) an entry in the MEX List. Enter a substitution for /*T*/ (such as 800)		Correct DTMF tones sent.		

(*) preferably automated when stress testing.

A.3 Build (Regression) Testing

Objectives: Tests to make sure that no errors have resurfaced due to changes introduced in the latest build.
Tests **7c, 9d, 12f, 12g**, and **12e, 6c** are preferably run after each new version is released.
All these tests are automated and are the main functions of the program.

TRITA-ICT-EX-2013:13

www.kth.se