CODEN: LUTEDX (TETS-5423)/1-40/(2001) & local 7

Functional testing of GUI based software without using the GUI

Master Thesis by

Nima Davoudi-Kia
Alijan Momeni

Performed at Telelogic AB
Supervisors

Tomas Lundh
Jan Docekal

Department of Communication Systems at Lund Institute of Technology
Supervisor

Per Runeson

March 2001

Lund University, Sweden

Abstract

Today’s software systems usually feature Graphical User Interfaces (GUI). Tools, which help programmers quickly create applications with the GUI, have dramatically improved programmer productivity, which in turn has increased the pressure on testers. This software must be thoroughly tested before each new release.

Today’s GUI testing is mostly done manually, which is costly and time consuming. It would be desirable to fully automate this testing process. By having an automated testing process, it would be possible to significantly reduce the time and manpower needed for testing.

In this master thesis, an introduction of testing issues and the effects of automated testing on testing process is given. Different kinds of testing tools and the internal experience of automated testing of GUI:s are also discussed in this thesis.

This master thesis focuses on how an application with a GUI can be tested and steered without involving the GUI. The goal is also to investigate how automated regression test should be performed so that the effort of keeping the tests up to date is minimized when the functionality of the application changes.

The attempt of testing the GUI without using the GUI, was partly succeeded. This kind of automated testing is strongly depended on the structure of the product being tested. The test object in this thesis was structured in a way, which made the testing method difficult and time consuming.

At the end of this master thesis, a case study is given to show different kinds of obstacles, which can rise during the testing process.

Acknowledgements

We would like to take this opportunity to thank our supervisors, Per Runeson at the department of the communication system at Lund Institute of Technology and Jan Docekal at Telelogic Technologies Malmö AB for their help and support throughout this master thesis.

Thanks also to Tomas Lundh, the originator of this master thesis, and Patrik Rosenqvist for their guidance during the first steps of our work.

Last but not least, we would like to thank Niclas Bauer, Engin Zufer and all the other employees at Telelogic who helped us with various questions.

Malmö, March 2001

Nima Davoudi-Kia and Alijan Momeni

Table of Contents

1 Introduction

1.1 Purpose

Manual testing of GUI(Graphical User Interface)-based software is labor-intensive and not well liked by software testers [Fewster & Graham, 99]. However GUI:s wide range of possibilities for user interaction and the number of control elements (buttons, pull-down menus, tool bars, etc.) have made it popular and useful among today's software systems.Therefore the GUI is target for automation and several tools for computer-based testing of the GUI are already commercially available.

This master thesis is intended to investigate test process performance, automated testing and testing of the GUI-based software at Telelogic, which is the supplier of software development tools for real-time applications.

Many companies have tried to automate the GUI, but testers usually revert to manual testing, because it is easier to maintain the manual test cases. At Telelogic there has been an investigation about how to automate testing of the GUI-based software by using the GUI [Rosenquist & Bruck, 98]. This investigation is not enough because a large amount of maintanence effort is required every time the software is changed. More details concerning this issue are given later in this master thesis.

1.2 Method and Main result

Several investigations, concerning automated testing of GUI-based software by using the GUI, have been done and a wide variety of tools for computer-aided testing of the GUI have been developed. But all these approaches require a large maintenance effort of test cases every time the software is changed or updated.

The main idea behind this master thesis is to investigate the possibility of reducing the maintenance effort by trying to automate testing of GUI-based software without using the GUI. Because it is a new area, in which no serious investigations have been done, no tools are developed or available for this purpose.

At the beginning, it was intended to find a common interface where it should be possible to capture signals between two communicating units, i.e. an arbitrary GUI and the related program code. These signals would, on a later opportunity, be used as input for running the test automatically. This attempt failed because GUI and program were two overlapping units and such an interface was missing.

Another possibility to achieve this goal was to invoke a desired function by writing single instructions into the program code, and compare the outcomes with the input to analyze the result of the automated testing.

This approach was, to some extent, successfully performed but had also some disadvantages, which are discussed later.

Chapter 2 focuses on software testing in general, verification, validation and software engineering environment.

In chapter 3, test and development process at Telelogic is described. Some improvement proposal are also given at the end of this chapter.

Automated testing in general, computer-aided software testing tools and automated testing of GUI-based software by using the GUI are taken up in chapter 4. This chapter is ended by a description of the investigation done in this area including the related conclusions.

Chapter 5 gives a highlight to automated testing of GUI-based software without using the GUI. The performed task is described by the means of a case study with associated results.

At last, in chapter 6, some conclusions concerning test implementation on design are drawn followed by suggestions and future investigations.

2 Background

2.1 What is software testing?

Testing is a process of planning, preparation and measuring, aimed at assessing the characteristics of an information system and demonstrating the difference between the actual and the required status. Activities such as planning and preparation emphasize the fact that testing should not be regarded as a process that only begins when the object to be tested is delivered [Koomen & Pol, 99]. Of course a test case[1] can not be executed before the software has been developed, but it can be designed based on a requirements specification.

According to [Fewster & Graham, 99], the quality of a test case can be described by the following attributes, i.e. how good it is:

- **Effectiveness** Shows the detection ability of a test case to find defects.
- **Exemplariness** An exemplary test case should test more than one thing so that the total number of test cases required should be reduced. It should also pinpoint the found errors if the test tests several things.
- **Cost considerations** How **economical** a test case is to perform, analyse and debug; and how **evolvable** it is, i.e. how much maintenance effort is required on the test case each time the software changes.

A well designed test case requires that these attributes must be balanced one against another. For example, a high measure on the exemplariness which cost a lot to perform, analyse, debug and may require a lot of maintenance, can result in low measure on the economic and evolvable scales [Fewster & Graham, 99].

1.Test case: A set of a tests performed in a sequence and related to a test objective.

2.2 Verification and Validation

Verification and validation ensures that software confirms to its specification[1] and fulfils the need of the customer. Validation makes sure that the system has implemented all the requirements, so that each system function can be traced back to a particular requirement in the specification. Verification ensures that each function works correctly. That is, validation makes sure that the developer is building the right product according to the specification, and verification checks the quality of the implementation.

Verification and validation can be divided into two types of techniques for system checking and analyzing, namely: static and dynamic techniques.

- **Static techniques** These techniques are related to the analysis and checking of the system representations such as the requirements document, design diagrams and program source code. These techniques can be used in all stages.
- **Dynamic techniques** These techniques can only be used when a prototype or an executable program is available and involve exercising an implementation.

2.2.1 The testing process

In developing a large system, the testing process involves several stages.These stages, described in [Koomen & Pol, 99], [Pfleeger, 98] and [Binder, 99] are:

- **Unit testing** Individual components are tested to ensure that they operate correctly. Unit testing is done in a controlled environment whenever possible, so that the test team can feed a predetermined set of data to the component being tested and observe what output actions and data are produced.
- **Integration testing** After unit testing, the next step is to ensure that the interfaces among the components are defined and handled properly. Integration testing is the process of verifying that the system components work together as described in the system and program design specifications.
- **System testing** Before handing over the system to the customer, it is tested by developers or an independent test team in a controlled envi-

1.Test case specification is a document which contains a description of test cases and requirements specification is a document which contains a description of something the system is capable of doing in order to fulfil the system's purpose.

ronment, that should demonstrate that the developed system or subsystem meets the requirements set in the requirements specification.

- **Acceptance testing** This is the process of comparing the end product to the current needs of its end users.The system is tested with data supplied by the system procurer rather than simulated test data. It is usually performed by the customer or end user, where the system is checked against the customer's requirements description.

- Finally the accepted system is installed in the environment in which it will be used; a final **installation test** is run to make sure that the system still functions as it should.

- **Regression test** This kind of test involves executing a predefined battery of tests against successive builds of an application to verify that bugs are being fixed and features / functions that were working in the previous build have not been broken. Regression testing is an essential part of testing, but is very repetitive and can become tedious when manually executed build after build after build. This kind of testing which is becoming more important have to be done throughout the development cycle, as illustrated in Figure 2.1.

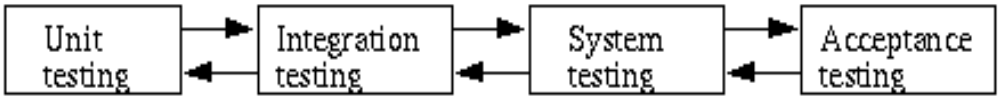

Figure 2.1 Testing process

All these phases should be performed as often as possible. The relation between the regression test and other types of above mentioned tests are illustrated in Figure 2.1. The arrows from the top of the boxes indicate the normal sequence of testing while the arrows returning to the previous box indicate that previous testing stages may have to be repeated, i.e. regression test.

Regression test may also lead to edit existing test cases or create new test cases.

2.2.2 Fundamental testing strategies

Test specification techniques can be divided into two groups: white-box and black-box techniques.

- **White-box** These testing techniques are based on the program code, the program descriptions, or the technical design. Knowledge of the internal structure of the system plays an important role.

- **Black-box** These testing techniques are based on the functional specification and quality requirements. In black-box testing techniques the system is viewed as it will be in actual use. Black-box testing is done without any internal knowledge of the product.

2.2.3 Cost of testing

Software goes through a cycle of development stages. A product is imagined, created, evaluated, fixed, put to serious use and found wanted. The full business, from initial thinking to final use, is called the product's life cycle. According to [Kaner ed al, 99] the product's life cycle involves many stages, but it can be summarized in five basic stages as: Planning, Design, Coding and Documentation, Testing and Fixing, Post-Release Maintenance and Enhancement.

The relative costs of each stage, according to [Kaner ed al, 99], can be summarized, as follows:

Table 2.1: Cost of each stages in percentage of total and Development costs

Stages	total costs	development costs
Requirement Analysis	3%	9%
Specification	3%	9%
Design	5%	15%
Coding	7%	21%
Testing	15%	45%

Table 2.2: Operation and maintenance cost in Production phase

Operation and Maintenance	67%

The tables above shows that maintenance is the main cost component of software. Testing is the second most expensive activity accounting for 45% (15/33) of the cost of initial development of a product. Testing also accounts for much of the maintenance cost, since code changes during maintenance have to be tested too.

Testing, finding and fixing errors in programs can be done at any stages in the life cycle and can be estimated from 40% to 80% of the total development cost. However the cost of finding and fixing errors increases dramatically as development progresses. Figure 2.2 shows that the later an error is found, the more it costs to fix.

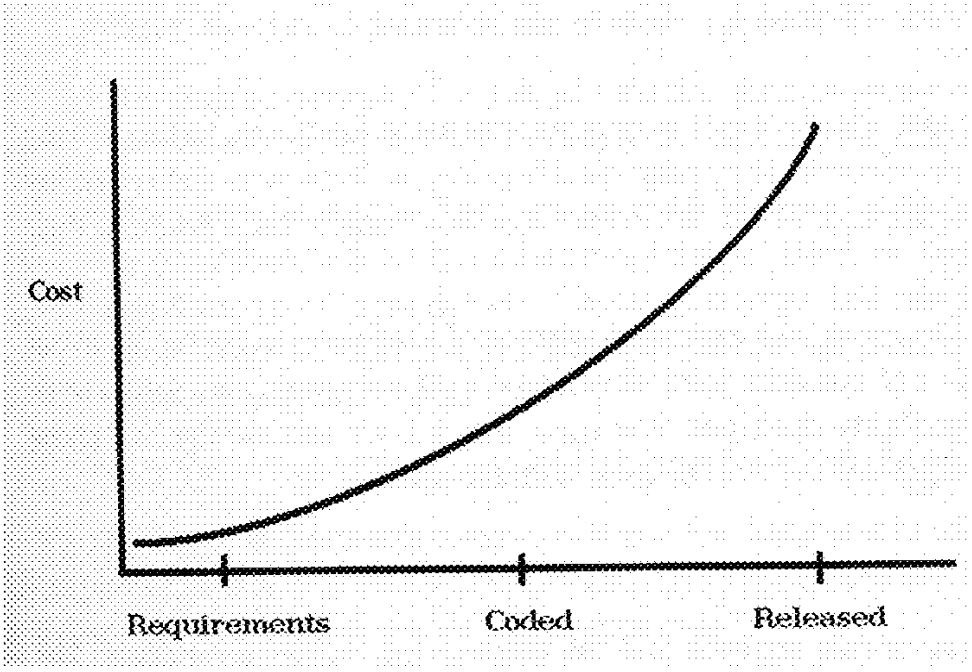

Figure 2.2 Cost of finding and fixing software errors according to [Kaner ed al, 99]

Changing a requirements document before the code has been written will cost less than changing it after the code has been written, since the code must be rewritten. Defect fixing are much cheaper when programmers find their own faults. They do not have to explain the defect for anyone else, and there will not be any communication cost. Fixing a defect before releasing a program is also cheaper than sending new disks, or even a technician, to each customer.

2.3 Software Engineering Environments

A software engineering environment (SEE) is a set of hardware and software tools, which can act in combination in an integrated way to provide support for the whole of the software process from initial specification through to testing and system delivery.

The SEE may be considered as a set of services, which are used by the facilities, which provide end-user support. Services may be provided by the platform on which the SEE is executing or by an environment framework.

The platform on which an SEE runs is called its host system. In some cases, the software developed using SEE will run on the same platform but, in many cases, it will be delivered for some target system, which may have a completely different architecture and operating system. According to [Sommerville, 95] host-target development is necessary for the following reasons:

- In some cases, the application software under development may be for a machine with no software development facilities. This is most likely in the case of real-time systems for special-purpose computers. These computers may not even have an operating system but only a simple real-time executive.
- The target machine may be application-oriented and not well suited to supporting software engineering environments, e.g. game consoles.
- The target machine may be in use and dedicated to running a particular application and this must be given priority over software development.

According to [Whittaker, 00] testers must identify and simulate the interfaces, which a software system uses and enumerate the inputs, which can cross each interface. There are four different interfaces, which can be described as follow.

- **Human interfaces** These interfaces are communication links between people and software. Most prominent is the GUI but older designs like the command line interface and menu-driven interface are still in use.
- **Software interfaces** These interfaces are about how software uses an operating system, database, or runtime library. These application provide services, which are modelled as test inputs. Testers should check not only the expected but also the unexpected services.
- **File system interfaces** These kind of interfaces exist whenever software reads or writes data to external files. Developers must write lots of error-checking code to determine if the file contains appropriate data and formatting. Testers must build or generate files with content, which is both legal and illegal, and files, which contain a variety of text and formatting.
- **Communication interfaces** These interfaces allow direct access to physical devices and require a communication control. To test such software, testers must be able to generate both valid and invalid protocol streams. Testers must gather and submit to the software under test many different combinations of commands and data, in the proper packet format.

Testers must understand the user interaction that falls outside the control of the software under test, since the consequences can be serious if the software is not prepared to cover such exceptional cases. Each application’s unique environment can result in a significant number of user interactions to test.

3 Test Process at Telelogic

3.1 Introduction

Telelogic, with more than 1300 employees, is a supplier of software development solutions for real-time applications. Based on dedicated, standardized languages and notations, Telelogic's tools, Telelogic Tau (consisting of the UML Suite, the SDL Suite, the TTCN Suite, SCADE and Logiscope) and ObjectGeode can provied an effective solution in the field of integrated analysis, design, implementation, testing and debugging of real time software systems.

Editors, related to the suites above, can be classified as graphical and text editors. The graphical editors, which all have very similar graphical interface, are used to create and edit diagrams according to their corresponding graphical notation. The Text Editor differs from the graphical editors and is used to create and edit text documents.

Testing is a vital part of all software and hardware development but it is often regarded as a necessary evil: it is considered as a difficult and uncontrollable process, which takes too much time and money. Unfortunately, in many cases this opinion is justified with regard to the fact that testing accounts for between 25 and 50% of the total budget [Koomen & Pol, 99]. At Telelogic, the test time is estimated to be about 20% of the total development (calendar) time [Rosenquist & Bruck, 98].

Testing at Telelogic, to a great degree focused on the functionality of the product, is performed by testers and developers. Therefore planning of a successful test process and keeping the release date requires an insight into the development process. That is the reason why we first begin with a concise description of the development process at Telelogic.

The test object, Telelogic Tau, is an integration of in-house developed COTS (Commercial Off-The-Shelf) components for the development of real-time software and all tests are run on Unix and Windows platforms.

3.2 The development process at Telelogic

Telelogic delivers two releases of their product per year. The product with its components is evolved in releases, with each release including new and improved features that should ensure that the vendor stays ahead of its competitors. This goal is reached by using a six months long development process, which based on [Rosenquist & Bruck, 98] and [Regnell ed al, 98] is described below.

3.2.1 Requirement Engineering ProcEss At Telelogic (REPEAT)

REPEAT is a specific industrial Requirement Engineering (RE) process which manages requirements throughout a release cycle. The actors involved in REPEAT are:

- **ReQuirements Management Group (RQMG)** This group manages requirements and make decisions on which requirement to implement. RQMG includes department and projects managers.
- **Issuer** Any employee at Telelogic, usually from marketing & sales or customer support, who submits a requirement to the management group RQMG can act as an issuer.
- **Requirements Team** This team analyses and specifies a set of requirements and consists of persons participating in implementation, testing, marketing & sales and customer support.
- **Construction Team** A group of developers who design and implement a set of requirements.
- **Test Team** A team that is responsible for the system and release test.This team consists of project-hired students and the release manager.
- **Expert** A person, typically a developer, who is assigned to analyse a specific requirement
- **Acceptance Group** This group, consisting of department and project managers, decides on the releasability of the product.
- **ReQuirements Data Base (RQDB)** All requirements are stored in this in-house-built database system which has a web-interface that can be accessed by Telelogic employees from a multi-continent intranet.

REPEAT is divided into five overlapping phases:

- **Collection** This phase, which lasts six months is made by an issuer that fills in a web-form and submits the requirement for storage in the data base(RQDB). New requirements can be accepted into an ongoing requirement process at any time. Requirements are described using natural language and has a summary name, an explanation of why the

requirement is needed and an initial priority provided by issuer. The priority, i.e. an assessment made by issuer, is on a scale from one to three as shown in Table 3.1.

Table 3.1: Levels of priority

Level	Priority
1	Allow to impact on-going construction.
2	Incorporated in the current release planning.
3	Postponed to later release.

When a new requirement has arrived, the management group(RQMG) first reads it to see if it is detailed enough; if not it is returned to the issuer for more detail.

- **Classification** In this phase, management group(RQMG) assigns a requirement (with clarified description) to an expert who classifies the requirement by providing it with a rough estimation of its cost, i.e. how long time it would take to implement, and impact, i.e. how many component that need to be changed. The estimation should take less than 30 minutes, otherwise the expert should recommend the management group(RQMG) to initiate a pre-study where the requirement can be decomposed into smaller requirements that is easier to classify. The expert also reconsiders the priority and may recommend management group(RQMG) to change it. Both cost and impact are given on a scale from one to five as illustrated in Table 3.2.

Table 3.2: Cost and impact levels

Level	Cost	Impact
1	Less than 1 day	One component
2	Less than 5 days	A few components
3	Less than 5 weeks	Less than half of all components
4	Less than 3 months	More than half of all components
5	More than 3 months	Nearly all components

- **Specification** The goals in this phase are: to select which requirements to implement in the current release, to specify the selected requirements in more detail and finally to validate the Requirements Document (RD) as an output of this phase. RD consists of a selected-list, a detailed specification of all selected documents, and a not-selected-list

including the requirements that are postponed to the next release. The selected-list is, according to the selection-rule, divided into a must-list, with 70% of the total resources (i.e. a 30% risk buffer dealing with optimistic effort estimations) and a wish-list with 60% of the total resources. RQMG has the responsibility of sorting the selected requirements into priority order using the priority, cost, impact and effort estimations.

- **Change** As mentioned above, new requirements can be accepted into an ongoing requirement process at any time. If incoming requirements with high priority are suggested to impact the current development process, RQMG may decide to introduce this into the must-list. But these actions often alter the selection-rule and thereby cause project delay. To avoid such problems, RQMG has the possibility to de-select a set of requirements amounting to the same effort as the new requirements and put them into wish-list.

- **Verification** In this phase, the implemented requirements in the selected-list are compared with the requirement specification using a requirements-based testing method. Development teams also inform which requirements have been implemented, possible changes to the requirements and known risks. When the implementation is correct with respect to RD, the new release is delivered to marketing & sales and the implemented requirements enter the applied state.

3.2.2 Implementation Process

When the specification phase is completed, the construction team has to put the idea and the requirements into practice. Now we are ready to enter a new phase, which is called implementation phase. This phase affects mostly the construction and test team. A short description about what construction and test team do follows:

- **Construction team** They have the responsibility to design and implement the requirements until code stop[1]. After the code stop, no new functionality will be added. Every four weeks the construction team should deliver a report, which is called status report. In this report they should inform the managers if they are compliant with the time plan. They should also state which resources have been used and if there exists any possible risk regarding the construction and time plan.
- **Test team** Test teams work begins with updating the test specifications (test cases in different documents) for the new functionality. They should also correct errors in the test specifications. Test teams also per-

1.A fixed date beyond which no new functionality should be added.

form system and acceptance test, which are described in the forthcoming subsection

3.2.3 Telelogic Tau Testing

The testing of Telelogic Tau is divided into three stages: unit, system and release test. How these three testing levels are performed, is discussed in more detail below. Before discussing the performance of these three testing stages in more details, it would be a good idea to give a short description about handling of found defects throughout the test process.

During system and release test all found defects are reported by anyone. All defectreports are stored in a web-based database. This database is the formal communication channel between testers and developers. There are several issues that should be sent with the defect report. These issues are Platform, Version, Tool, Summary name and the Severity of the defect, which can be minor, non-critical, critical or catastrophic.

Statistics about found defects are collected at the end of every week. The statistics are classified by severity of the number of found defects and the number of defects that are left to correct. These statistics are presented graphically and are available for all employees.

- **Unit test** Unit test is usually performed by developers to verify that new implemented functions are working, as required. Test data from old defects will be used for a more regressive testing. As mentioned, test team updates the test specification and in some cases writes new test cases during this period.
- **System test** This period begins when there is less than one month left to code stop[1]. During this period the test teams will run through the test specifications. Most of this testing is done manually by project hired students. However, there are parts of Telelogic Tau, with text-based user-interfaces, i.e. the Simulator, Validator, Analyzer, Code Generator and TTCN Link, which are mostly tested automatically by using batch-scripts and file comparators. These interfaces will be tested by data files, which contain errors that should be detected and data files, which have supported identification of defects in earlier versions.

  When defects are reported, at first they will get the status *Assigned*. When the developers have solved the defect they will change the status to *Solved*. They will also run through parts of unit testing that has to do with the defect. Developers can also change the status of a defect to *More info* or *No action* in case they need more information or if some one else has reported the same defect. When the status of a report changes to Solved the defect reporter should verify it. Now the status

1.A fixed date beyond which no new functionality should be added.

of the report will change to *Verified* or *Not Verified*. Developers should also report new found defects during solving other defects.

- **Release or Acceptance test** At Telelogic the acceptance test is called release test. The verification of the solved defects continues in this stage. The verified defects are tested again and if they are corrected they will be closed. The test team also performs regression testing during system and acceptance test. Regression testing is performed by running free tests either by following some parts of the test specification or by testing generally the important parts of the tool. During the acceptance test the developers are not allowed to solve any defects without the project manager's permission. In this way the risk of adding new defects is reduced. At the end of this stage, all defects with the severity catastrophic or critical should be solved. The acceptance group decides if the product is ready for release or not. This decision is based on remaining defects in the database. After burning the product on the CD, the test team performs an installation test to make sure that product performs as it is required. After installation testing, the test team updates the test specification again.

3.3 Conclusions

Even when a system is developed with an egoless approach, developers sometimes have difficulty removing their personal feelings from the testing process.Thus, an independent test team is often used to test a system [Pfleeger, 98]. At Telelogic, this demand is mostly fulfilled by the Test Team, which mainly consists of project-hired students, who are inexperienced. The benefit with this approach is the fact that an inexperienced tester is more disposed and objective to question functionality, program behaviour and find more defects. Found defects are stored in the defect database, which provides a defect tracking system. These defects can be used to make conclusions about testing and program quality.

As stated in [Rosenquist & Bruck, 98], testing GUI-applications or regression tests during a long period makes the tester unattentive. Therefore it is a good idea to automate those test cases. On the other hand, automation of the testing process is not only desirable, but in fact is a necessity given the demands of the current market.

4 Automated testing

This chapter is a review of Computer-Aided Software Testing (CAST) Tools and describes automated testing of GUI-based software by using GUI according to [Rosenquist & Bruck, 98].

Chapter 5 is focused on automated testing of the software without using GUI.

4.1 Introduction

"Automated testing" means automating the running of tests currently in use by using a suitable test tool. The most efficient use and purpose of automated test tools is to automate regression testing. This means that you must have or develop a database of detailed test cases which are repeatable, and this suite of tests is run every time there is a change to the application to ensure that the change does not produce unintended consequences.

Automating test affects only how economic and evolvable they are. Once implemented, an automated test is generally much more economic. In addition, the benefits, among others, concerning automated testing are: regression tests are run often and quickly, better use of resources, testing attributes, e.g. Graphical User Interface (GUI), which are not always easy to verify manually[1] [Fewster & Graham, 99].

Despite these benefits, test automation has its limitations. Automated testing does not replace the need for manual testing because there will always be some testing that is much easier to do manually or is very expensive to automate. Test automation does not improve effectiveness and may limit software development. And test tools has no imagination, i.e. a tool is only a software. A test tool can only obediently follow instructions but a human tester can also use his or her creativity and

1.A GUI object may trigger some event that does not produce any immediate output. A test execution tool may be able to check that the event has been triggered, which would not be possible to check without using a tool.

imagination to improve the tests as they are performed, either by deviating from the plan or preferably by noting additional things to test afterwards.

As noted above, manual testing of a Graphical User Interface (GUI) is difficult and time-consuming, but its wide range of possibilities for user interaction and the number of control elements (buttons, pull-down menus, tool bars, etc.) made it popular and useful among today's software systems. Therefore GUI is target for automation and several tools for computer-based testing of GUI are already commercially available.

4.2 Computer-Aided Software Testing (CAST) Tools

4.2.1 Introduction

A tool is an instrument or automated system for accomplishing something in a better way. This "better way" can mean that the tool makes us more accurate, more efficient, or more productive or that it can enhances the quality of the resulting product [Pfleeger, 98]. Within the automated testing area there are several kinds of Computer-Aided Software Testing (CAST) tools, which depending on the activities the tool is associated, can be divided into different classes. These classes according to [Kit, 95] are:

Tools for reviews and inspections

These tools assist in performing reviews, walk-throughs and inspections of requirements, functional design and code. The types of tools required for support of reviews and inspections are [Kit, 95]:

- **Complexity analysis tools** These tools help to identify high risk, complex areas. The cyclomatic complexity metric is based on the number of decisions in a program, which is important to testers, because it provides an indication of the amount of testing necessary to practically avoid defects.
- **Code comprehension tools** This kind of tool help us to understand unfamiliar code. It can be used to identify areas that should receive special attention, such as areas to inspect.
- **Syntax and semantic analysis tools** These tools perform extensive error checking to find errors that a compiler would miss. These tools are language dependent and can analyse code, maintain a list of errors, and provide build information.

Tools for test planning

A test plan provides the foundation for the entire testing process and it defines resources and schedule of testing activities. The types of tools required for test planning are:

- **Templates for test plan documentation**
- **Test schedule and staffing estimates**
- **Complexity analyzer**

Tools that identify complex areas can also be used to locate areas that should impact planning for additional tests based on basic risk management. The biggest help usually comes from IEEE/ANSI Standard for Software Test Documentation, which describes the purpose, outline, and content of the test plan. Even if test-planning tools don't eliminate the need to think there are still many companies, which have found it useful to simply have someone enter the outline for the test plan.

Tools for test design and development

After test planning we will enter into a new process which is called test design process. In this process all test approaches in the test plan will be more detailed. The test design process identifies and prioritizes the associated test cases. Test development is the process of translating the test design into specific test cases. It should be mentioned that even in this stage there is not a lot of help from the test tools, specially, mental process of the test design. The types of tools required for test design and development are:

- **Test data generator tools** This kind of tools automates the generation of test data based on a user-defined format. These tools are useful when large quantities of test input data are needed.
- **Requirements-based test design tools** These tools are for those who desire disciplined approach, is used to design test cases to make sure that the implemented system meets the formally specified requirements document.

Two other tools in this area are **Capture/playback** and **Coverage analysis** tools. A detailed description about these tools is given in section for test execution and evaluation tools.

Test execution and evaluation tools

Test execution and evaluation is the process of executing test cases and evaluating the result. These tools can be used to automate the running of selected test cases for execution and measuring the effectiveness of the effort. Automated test execution tools are essential for handling the very large number of test cases that must be run to test a system thoroughly. The types of tools required for test execution and evaluation are:

- **Capture/playback** These tools capture test input, data and actions including keystrokes and mouse activities and are able to perform an automatic replay, so that the tests can easily be repeated at a later time. This frees the tester from having to manually re-run tests over and over again. Discrepancies are reported to the team, and the captured data help the team trace the discrepancy back to its root cause. The problem in using capture/playback tool is that certain stability is required to automate these tests. A keyword in a well-automated test is good maintainability. It should be possible to adjust an automated test with relatively little effort for a changed application and recorded test cases are in general not easily maintained. The other problem with this tool is that it is very expensive and time-consuming to use.

- **Coverage analysis tools** These tools measure which parts of a product under test have in fact been executed and which parts have not been covered and therefore need more tests. The measurement can be carried out at a module level or at a subsystem level. For keeping track of the coverage information, these kinds of code run the source code into a preprocessor. The fact that the new source is bigger than the old one leads to growing size of our object module. However, even structural test coverage of 100% does not guarantee that testing was complete.

- **Memory testing tools** This tool's ability is to detect memory problems. Errors can be identified before they become evident in production and can cause serious problems. Memory testing tools tend to be language and platform specific. The tools in this category are usually easy to use and reasonably priced.

- **Simulators tools** Simulator tools simulate the operation of the environment of the system to be tested. It is used to test software which is too expensive, dangerous, or even impossible to test in the real environment, for example testing the control software of an airplane or a nuclear reactor. These are the only practical method for testing when software interfaces with uncontrollable or unavailable hardware devices.
- **Performance tools** Performance tools determine the performance capabilities of the system. These tools make it possible to create, control, and analyze the performance testing of client/server applications.

4.2.2 The tool selection

For making automated testing cost-effective or improving the quality of the software product, special attention should be paid to the choice of test tools that best fit

the testing requirements, because of the fact that the test case execution in automated testing may involve starting the test tool. Following selection criteria, which are derived from discussing the experiences of [Fewster & Graham, 99], [Pfleeger, 98] and [Rosenquist & Bruck, 98] should be taken into account when choosing a test tool.

- **Compatibility** Choose a test tool that will enable you to automate testing in your organisation, e.g. the tool must support the platforms available in the organisation. There are also a number of requirements, which must be identified first, so that there is something to evaluate the candidate tools against. The test tool must have all the critical features needed especially for test result validation and for managing the test data and scripts. It should also support facilities to set up test cases as e.g. creating directories or removing files.
- **Maintainability** An important aspect to investigate is how easy it will be to maintain the test case in proportion to software changes. Maintenance effort can be reduced not only by means of a well-defined process but also by the test tool functionality, e.g. some tools may be less susceptible to your frequent types of software change than others, which will make the basic script editing easier.
- **Learnability - Usability** It should not take a long time to master the tool. This criterion can be more or less important depending on the way in which the tool is used in the organisation. For example, if a certain group in the organisation will continuously use the testing tool, it will not be an important disadvantage if it takes some time for the tool to be mastered. The tool should be easy to use or its features should not be cumbersome and difficult. Documentation, scripting language and the interface of the testing tool are the factors that can affect this criterion.
- **Reliability - Continuity** Make sure that the tool works without failure because it can sometimes happen that the tool is not well-tested by the vendors. The tool should not interrupt the test execution for every detected minor discrepancy.

4.3 Automated testing of GUI-based software by using the GUI

Creating test cases for automated testing, in this case with GUI, requires a test tool and a test object, which in this case is Telelogic Tau. The way of selecting these tools, creating test cases and collecting metrics as an outcome from the automated testing are summarised below.

In this section we will concentrate on [Rosenquist & Bruck, 98], which is the only reference available concerning automated testing of software with GUI at Telelogic.

4.3.1 Test Tool Selection

The MSC editor (version 3.4) was chosen as the test object. The best test execution tools for GUI's seemed to be capture/playback tools since the goal of the investigation was to focus on GUI test problems. In order to choose a good capture/playback tool, three kinds of these tools have been compared with each other. These tools are TestWorks from Software Research Inc., Vermont Creative Software's HighTest and Mercury Interactive, Inc.'s WinRunner/XRunner.

Before choosing one of these tools, some tests have been created on Microsoft Notepad. While creating the tests and executing them with the aid of the test tools, some considerations had been taken into account, namely: the platform support, test case preparation, maintainability, error recovery, test management, debugging, learning curve and reliability/usability. The results of evaluation of the test tools are summarized below.

The functionality provided by TestWorks was very limited and Mouse-movement could not be ignored. One of the other problems with this tool was that it was not well documented.

HighTest was well documented and the functionality was much better than the TestWorks, but the major problem with HighTest was that error reports were not easy to analyse and it could not support the UNIX platform.

The best alternative was WinRunner/XRunner, which provided an extensive functionality and was well documented. WinRunner/XRunner's error reports made it easier to analyse and locate errors; and it also supported all required platforms. This tool was the most suitable choice for Telelogic but it was also the most expensive one among these tools.

4.3.2 Creation of automated test cases

While trying to create the test cases for version 3.4 of the MSC editor in Telelogic Tau by means of WinRunner, some problems were raised. The first problem was encountered when creating the test cases that would check that menu items are selectable or hazed according to the current state of the editor. These test cases did not work as intended and the reason for this was not provided by [Rosenquist & Bruck, 98].

Another problem that occurred when creating tests for the menu item. For doing update of a single table easier, instead of editing a number of test cases when the keybindings were changed, the capacity of WinRunner was intended to be used to store information (in this case the keybinding and short-cuts of menu items) in a table. That was impossible using library functions for entering key stroke provided by WinRunner and therefore keybinding tests were added to the test cases for the menu items created subsequently without using a table.

By the time more and more test script code were entered by hand instead of capturing most actions. This has the advantage of test cases becoming more flexible and

that it is possible to create test cases not only for program behaviour that works at the time, but also for scenarios that do not work on the current version of the application.

Furthermore test cases were not created for the pop-up menus in the editor. It would probably take some time to create the function needed to access pop-up menu items because there was no support for this in WinRunner at the time.

4.3.3 Metrics

Metrics collected, while managing the case study, were *time* to create, run and update test cases; and also the *number of found defects*. The corresponding metrics for manual testing had already been collected before the case study was initiated.

- **Time**

  The estimated time is shown in Table 4.1.

Table 4.1: Time spent on various tasks

	automated tests (man-hours)	manual tests (man-hours)
creating tests for v 3.4	100	-
running tests on v 3.4	1	6
updating test for v 3.5	9	ca 8
running tests on v 3.5	4	4

It should be noted that creating automated test cases for version 3.4 was based on the existing test specification for manual testing. But on the other hand testers were not used to work with a test tool, therefore it took extra time to create the automated test cases.

It is more difficult to evaluate the outcome of automated test cases than the manual one. The running time for automated test cases concerns almost test tool run time whereas automated tests require only a few minutes in order to be started. The time to update manual test cases for version 3.5 was estimated since the test specification was completely restructured. The updating time for automated test cases could possibly be less than 9 hours if the test cases were designed differently.

Table 4.1 shows that the maintenance effort is much lower than the cost of creating new test cases. Because of MSC editor's instability, the running time had increased for automated test

cases and decreased for manual test cases, since automated test cases had to be started one by one, instead of being executed in one sequence.

Another problem was that it took longer time to run automated test cases if defects were detected, but in manual testing it was easier to skip actions that could not be performed.

- **Number of found defects**

One of the most interesting issues when automating tests is a comparison of the found defects between automated and manual tests. During manual testing of version 3.4 of the MSC editor, 30 defects were found but only 7 of those defects was expected to be detected by automated test cases. The rest of the defects had to do with the integration with other tools or defects, which were not covered by automated test cases. Of those expected 7 defects only 3 defects were detected, but 6 other defects, which were not expected, had also been found by automated test cases. The number of found defects can be summarized in the Table 4.2.

Table 4.2: Number of found defects

total number of defects found during testing v 3.4	30
number of defects found during testing v 3.4 that might be found by the automated test cases	7
number of known defects found by automated test cases	3
number of new defects found by automated test cases	6

The reason of why those 7 defects were not detected might be that the test cases were not carefully created or they were not as thorough as they could be.

Those defects which were detected by automated and not by manual test cases were mostly minor defects, such as inconsistent titles of message boxes or the title bar of the editor window sometimes not being updated properly, but it should be mentioned that this kinds of defects are almost impossible to detect by manual test cases.

4.3.4 Conclusions

Below, the fact that introducing test automation make it possible to improve the current testing process is described with respect to the outcome from [Rosenquist & Bruck, 98].

- Automated tests are executed (six times) faster than manual tests can be performed, so that more time can be spent on the remaining manual tests.
- It was not proved that the cost-advantages with automated testing can be shown already after second regression test since the introduction of test automation was based on test cases for manual testing. But from Telelogic's point of view, the automation effort will repay itself at least by the end of the second project if regression tests are performed on each new build during the test phase.
- Calculations based on the time to create and run tests versus the number of found defects will lead to the conclusion that manual tests find defects with less effort than automated tests. But according to [Rosenquist & Bruck, 98], the time to update and run test cases versus the number of defects would be a better measure, since the time to create test cases becomes less significant with every test run and update; and in consequence, the relationship between the number of defects and automated or manual testing is the opposite.

5 Automated testing of GUI-based software without using the GUI

5.1 Introduction

As it was described in chapter 4, an investigation concerning automating testing of GUI-based software with using the GUI has been done. The main problem with this kind of test approach is the large amount of maintenance effort every time the software is changed. Another problem is that the size of drawing area changes every time the product is started. It makes the testing of drawing area difficult.

To avoid the above mentioned problems, this chapter is intended to investigate the possibility of automated testing of GUI-based software without using the GUI, i.e. by means of program codes. It is also desirable to determine: requirements that the test object must fulfil and to find out possible techniques to perform the test automation.

5.2 Telelogic Library Overview

To perform this task, a detailed knowledge about the relations between different editors and the structure of the library with included Directories was needed.

The Telelogic Library Overview is distributed over several directories and sublibraries. This structure has evolved over a number of years. The library is an integral part of Telelogic Tau and can be divided in four different layers. The frameworks represent both an infrastructure and a common code base that allows maintaining and developing certain aspects of the software in very controlled ways. Table 5.1 shows the structure of the Telelogic library.

Table 5.1: Telelogic Library Overview

Library	May Use
Bases	---
Application	Bases
Framework	Bases and Application
Editor	Bases, Application and Framework

- The first library, which is called *Bases* library is partly a repository for all C code, without regard to logical position within the framework. This library is also the foundation C++ library, which is intended to contain foundation classes, non-graphical OS independent abstraction layer, and general framework classes that only contains mechanism, instead of containing adoptions to the needs of particular tools in Telelogic Tau. This library needs to be stable and well documented.
- The second library, called *Application* library, contains module, functions and classes that implement application-level functionality. The reason of putting code in this library is that more than one application needs to access it. This library is also intended to provide an OS-independent abstraction layer to graphical programming.
- The third library, *Framework* library, provides support for writing editor applications, i.e. the framework supplies methods implementing the standard menu options the user expects to find in the menu bar in an editor. It makes these menus very easy to implement, since it provides callbacks for all the common menu items. This library is common for all editors except the SDL editor.
- The fourth library, which is called *Editor* library, contains the specific codes for the desired editor. All codes that are not common for other editors will be placed in this library, for example the content of an editor like menus or symbols. This library is of great importance for this master thesis because it is target for the test implementation.

In order to understand how these libraries interact with each other an example is given. This example also shows the course of event when doing a menu choice.

By choosing a menu, a function in the Application sublibrary is activated. This function invokes **WinMenuChoice** function in the Framework sublibrary. **WinMenuChoice** finds out the page related to the editor in which the menu was selected and invokes the function **MenuChoice**(window*, menuItem) in the editor

sublibrary. This example will be described in more details in the Case Study later in this chapter.

5.3 Choice of the test object

For selecting an appropriate test object from Telelogic Tau, it was preferred that the test object should have a simple structure with few menus, which should make it easier to get started and trace the possible obstacles.

Among editors in Telelogic Tau, two probable candidates for this purpose were Message Sequence Chart Editor (MSCE) and High-level Message Sequence Chart Editor (HMSCE). The HMSCE was more suitable because its program code was more familiar to us.

After choosing the HMSCE as test object, the next step is to give a short description and investigate the functionality of Menus, Dialogs and Drawing areas in HMSCE.

5.3.1 Menus

There are three kinds of menus, which are described below.

- **Pulldown menus** These menus are activated from a menu bar in an application window and allow the user to choose some actions to be performed. In most cases the application itself creates the desired number of menus in the menu bar. As a menu is created, the application supplies it with a unique integer number, which can be used to identify the menu in other calls and callback.
- **Popup menus** Popup menus are very similar to the pulldown menus. The menus can be defined for a drawing area and will appear when the right mouse button is pressed. The user has the possibility to choose a desired submenu among other submenus in a popup menu.
- **Option Menus** These kinds of menus are used to let the user select among a finite set of values, as an alternative to using a radio group or a text area. The user can select a new value in the menu, which will replace the previous value when the menu is unposted.

  If no history has been set for an option menu, the first item in the menu is the default current item.

5.3.2 Dialogs

A dialog box provides an exchange of information or dialog between the user and the application. In general there are two kinds of dialogs, i.e. Modal and Modeless

dialogs. A Modal dialog locks the underlying window until the user first handles the dialog, whereas a Modeless dialog does not lock the underlying window.

From a testing point of view dialogs can be categorized as setting/option and message dialogs. These dialogs may be either Modal or Modeless.

- **Setting/option dialogs** These dialogs can be divided in two different groups. In the first group, it is possible to choose the desired number of toggle, radio or ordinary (OK, Cancel, etc.) buttons. Figure 5.1 shows Window Option dialog of this group.

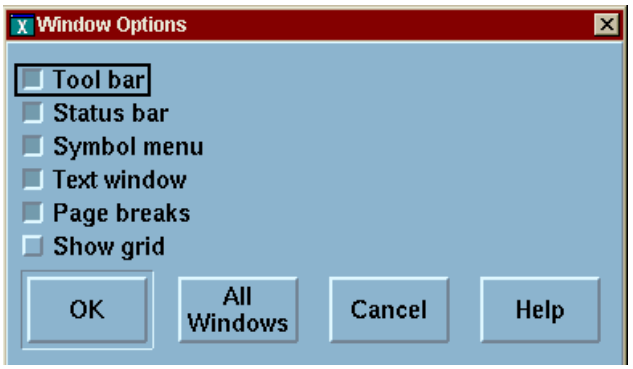

Figure 5.1 Window Options dialog.

The other group of setting/option dialog is almost as the same as the first kind. But the difference is that in this group the buttons are fixed, i.e. it is not possible to change the number of existing buttons. Figure 5.2 shows Print Setup dialog, which belongs to this group of setting/option dialogs.

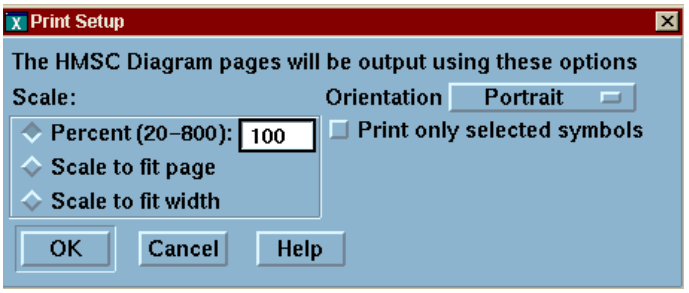

Figure 5.2 Print Setup dialog.

- **Message dialogs** These dialogs can actually be classified as two different types of dialogs, but from a testing point of view they can be considered as one type of dialog. Message dialogs are fixed in advance and often pops up to inform the user that something has been changed or wrongly done. Message dialogs have usually only one button and the user does not have any choice than confirming the content of the message. Figure 5.3 shows Help About of this kind of dialogs.

Figure 5.3 HelpAbout.

There are also message dialogs with several buttons, but they are still fixed dialogs, which informs something is changed and the user has the possibility to accept or regret the changes. Warning dialog, in Figure 5.4, pops up when the user chooses to revert a diagram.

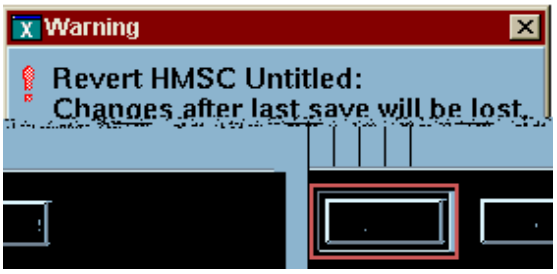

Figure 5.4 Revert Diagram dialog.

5.3.3 Drawing area

When handling an object in a drawing area, it is important to investigate the effect of invoking a function on the object. A well designed test case, aimed for automated testing of drawing area in an editor, should enable the tester to capture all these invoke-actions, which would be used as input for running automated testing.

Meeting this demand requires that the test case should mainly cover some significant operations. These operations may, among others, be: add, move, remove, create and copy an object. Figure 5.5 shows the drawing area of a HMSE.

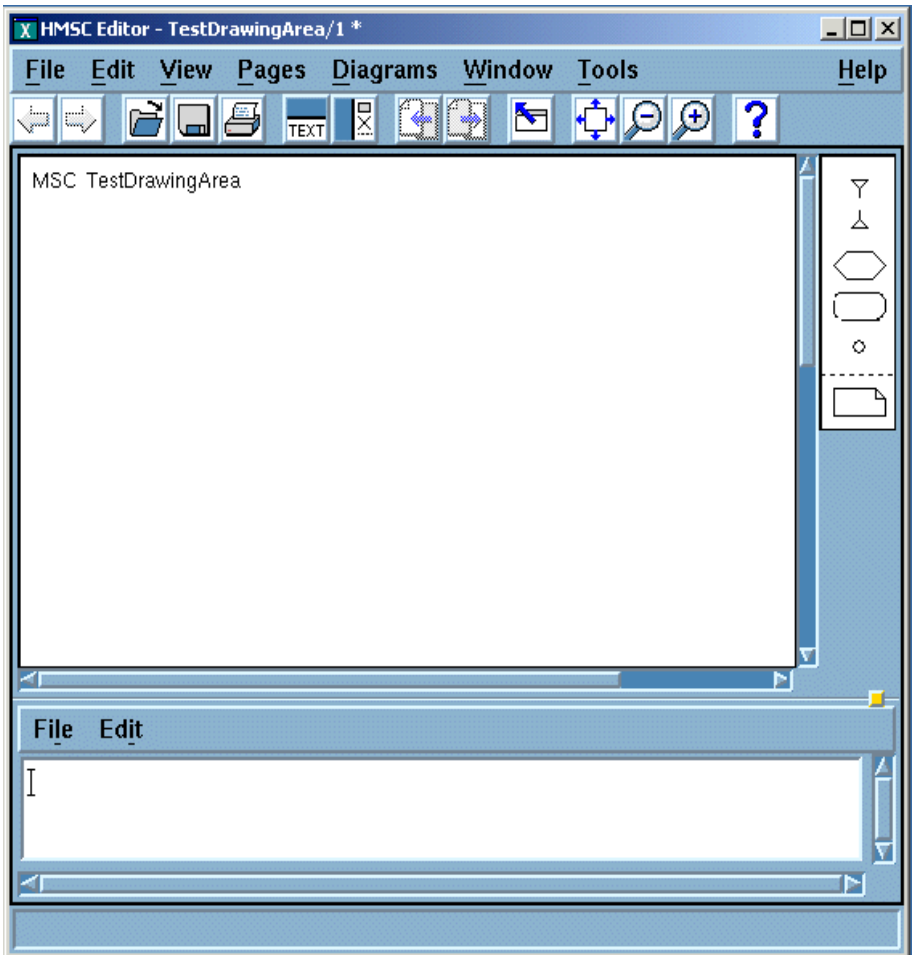

Figure 5.5 Drawing area of HMSCE.

5.4 Case Study

5.4.1 Introduction

The objectives of this master thesis are to investigate how test automation of GUI-based software can improve the test process, what kind of obstacles exists in the current product that makes the automation difficult and how these obstacles can be removed in future products.

In order to understand how test automation works and how the possible obstacles can be detected, a test case has been implemented. The preparation, execution and the result of test implementation will be discussed in this section.

5.4.2 Planning

Before getting a good start in software testing, a detailed knowledge about the software in question and its functionality is of great importance. So the first step was to investigate the structure of the Telelogic library, as shown in Table 5.1, and the interfaces between the sublibraries.

The idea behind this investigation was to find an interface, where it shuold be possible to capture all the signals, with which the Graphical User Interface (GUI) and program logic communicate with each other, as shown in Figure 5.6.

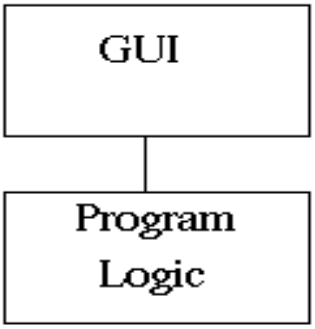

Figure 5.6 The desired relation between GUI and Program Logic.

These signals should, on later occasions, be used when testing Menus, Dialogs and Drawing areas.

During further studies of Telelogic library, it was found that the GUI and the program code were two overlapping units, see Figure 5.7, so that the desired interface did not exist.

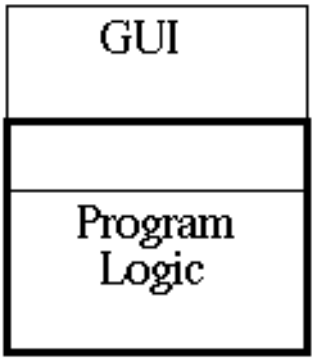

Figure 5.7 The real structure between GUI and Program Logic at Telelogic.

The lack of the interface mentioned above made it impossible to manipulate and steer all components by the means of just one interface. So, that was obvious that the Menus, Dialogs and Drawing areas had to be tested separately.

5.4.3 Execution

For capturing signals there are many different tools, among which capture/playback are considered to be the most suitable tools in this connection. These tools are mostly intended to test GUI-based software by using the GUI while the task was to perform the testing without using the GUI and therefore these tools could not contribute to the performance of the task. Another approach was writing and putting macros[1] into a relevant part of the program code.

As it is seen from **Table 5.1**, the Telelogic library can be divided into four sublibraries. The goal was to write and put macros into the Application sublibrary containing program codes, which are common for all types of editors. The advantage was that it would be possible to test and steer an arbitrary editor from just one interface, but it was found that this approach was time-consuming because not only the program codes would be written on both Unix and Windows operating systems but also for a simple invoke-action you have to change many other programs in this sublibrary.

To avoid this complexity, the Framework sublibrary seemed to be a better alternative than the Application library. In this sublibrary there is a function called **WinMenuChoice**, with which you can invoke another function in an desired editor in the Editor sublibrary. Invoking editors from this sublibrary requires, apart from writing macros, lots of changes in the existing program codes.

To reduce the work and time needed, it was decided to choose the Editor sublibrary as a test platform, where the macros would be implemented to steer and manipulate an desired editor.

This sublibrary contains all kinds of editors, among others HMSC editor. By the reason of simplicity, menus were chosen as the test object. In this editor there is a function called **MenuChoice**(window*, menuItem), which performs the choice of menu. With the aid of macros the **HMSCPage** class including **MenuChoice**(window*, menuItem) function, was redefined as **Original HMSCPage** class. Furthermore, to be able adding a macro to the **MenuChoice**(window*, menuItem) function, a new **HMSCPage** class, which inherited all functions from the previous class, was created. **Figure 5.8** illustrates the relation between the classes.

1.A single instruction in programming language that results in a series of instructions in machine language.

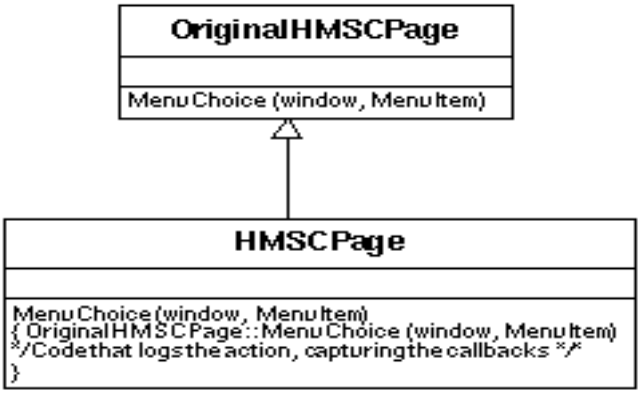

Figure 5.8 General structure of creating a new class for capturing callbacks.

During the test implementation, more than one window could be opened simultaneously. Hence it is important that the both menu and the activated window, in which the menu was selected, are pointed out in the form of two unique integer numbers.

By invoking the **MenuChoice**(window*, menuItem) function, the identity of the menu was provided by menuItem parameter.

The window parameter, a pointer, could just return an address related to the current window and this address was changed every time the program was started. But from automated testing point of view, the identity of the activated window was important. For solving this problem, some modifications were done in a list in the Framework library so that the list was able to find out the identity of number.

The two unique integer numbers were captured and printed by the **fprintf()** function. The appearance of the saved data, which were intended to be input to a replay function, is shown in Figure 5.9.

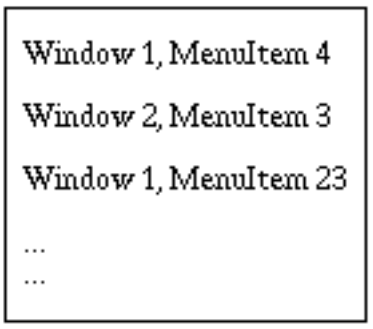

Figure 5.9 The captured data of menu choices.

To replay the captured data, a program was written in the Editor library. With the aid of the function **fscanf()** the input file was read. At the same time, the identity of the window was identified by a pointer, which was created for this purpose.

In order to start the replaying function an extra menu choice, Playback menu, was added to a menu list. By choosing the Playback menu, the saved data above was replayed and stored in another file, which was compared with the input to check the correctness of the result.

5.4.4 Results

The comparison between the input and output files of the capture/playback functions clearly showed that automated testing of menu choice was possible but it is far from being recommended as a testing technique for the current product. Although the idea of this master thesis was to investigate testing of menus, dialogs and drawing areas but for the lack of time and obstacles, concerning the structure of the current, it was not easy to make a test implementation for dialogs and the drawing area. The obstacles mentioned above and the disadvantage of this kind of testing are discussed in the chapter 6 *Conclusions*.

6 Conclusions

6.1 Testing Implementation on design

In this chapter some motivation regarding decisions made in this master thesis and the conclusion that has been drawn are presented.

As it was mentioned in chapter 5, only menu choices were tested and test implementation for dialogs and drawing areas was not done. The callbacks from menu choices were captured and replayed by macros. Using the same method for dialogs and drawing areas is also possible but with regard to the structure of the existing product, it is too difficult and time consuming.

From a testing point of view there are three types of dialogs, i.e. two setting/ options and one message dialog. Because of the absence of a common interface for the dialogs, the dialog classes belonging to these three kinds of dialogs would be tested separately. This means that the callbacks can not be captured by a simple macro and there will be lots of changes, which in turn will affect several files.

When it comes to drawing areas, in addition to the testing problems of dialogs, there are other issues, which should also be taken into account; that is the size of the drawing windows and the usability of captured callbacks.

The major problem is that the size of the page is usually changed every time the product is started or when opening a window while other windows are opened. It can result in a difference between the symbol's previous, in the form of captured callbacks, and current position so that the captured callbacks can not be used as inputs to a playback function for running the test automatically. It is also interesting to find out the way in which the moved symbol can affect the drawing area by scrolling the page.

The usability of callbacks means if the captured callbacks, when invoking a function, can directly be used by a replay function for automated testing or must be prepared in advance for this purpose.

A common problem is platform dependency of some actions, e.g. for some kinds of dialogs, the test must be implemented on both Unix and Windows platforms.

Another problem is the maintainability of the old test data. If a menu is added or removed in the next product the old captured data is useless, because the new menu items are not as the same as the items in the previous product. This maintenance problem can be solved by keeping track of the all menu items, which is not a good idea when lots of menu items are involved. This problem can also be solved by performing the test manually at the first time.

It is obvious that these kinds of test implementation demand lots of time and should be implemented by experienced programmer and performed by experienced testers with a good knowledge of the system and library overview.

It is also very difficult or maybe impossible to test the GUI without using the GUI in the current product at Telelogic.

6.2 Suggestions and Future investigation

Testing of GUI:s can be done in several ways depending on the structure of the GUI and the program logic. There are many software programmer, which are interested to automate the testing of the GUI-based product without using the GUI, but there has not been any investigation in this area. This kind of testing can be difficult, especially when it involves a drawing area.

The idea to capture the callbacks is problematic, due to the non-existent common interface in the current product. To be able to test the GUI-based software without using the GUI, a first step could be to separate the GUI from the program logic as it is shown in Figure 6.1.

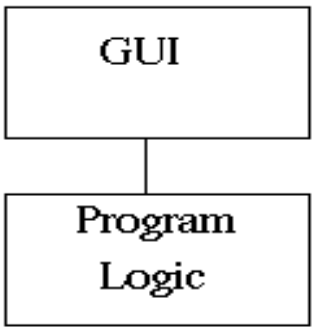

Figure 6.1 The desired relation between GUI and Program Logic.

This separation should give the desired interface, i.e. the common interface. With regard to the structure of the current product, it is a good idea to create this interface in the application library. This approach can make the testing of menu and some dialogs much easier.

The common interface makes it possible to capture all data that can be used as the input for a replaying function. The replaying function can be script written code, which activate the applications by using the captured data. Correctness of the result can then be controlled through comparing the input to and the output from the replay function, Figure 6.2 shows the desired system from a testing point of view.

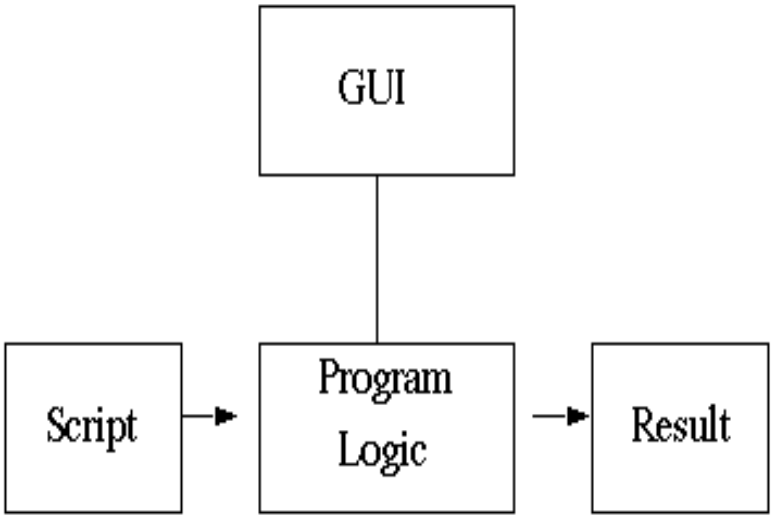

Figure 6.2 A desired system from testing point of view.

Using the captured signals, through the suggested interface, as input to a replaying function for testing of the drawing areas requires a lot of changes in the structure of the current product.

Another idea may be to write more code in the program logic and make GUI:s size smaller as it is, as shown in Figure 6.3.

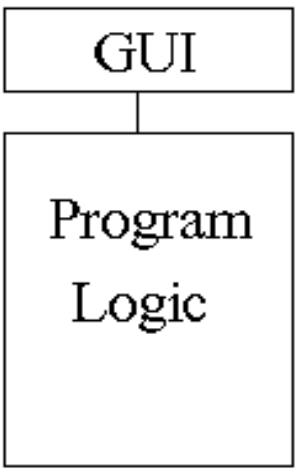

Figure 6.3 An overview from a structure of a big Program Logic and small GUI.

However, the approaches mentioned above are intended to minimize the amount of the work with the GUI during automated testing of the GUI-based software. The benefit is that it would be much easier to test the program logic than the GUI [Beck,99].

References

[Fewster & Graham, 99] Mark Fewster & Dorothy Graham, *"Software Test Automation"*, Addison-Wesley, 1999.

[Koomen & Pol, 99] Tim Koomen & Martin Pol, *"Test Process Improvement"*, Addison-Wesley, 1999.

[Pfleeger, 98] Shari Lawrence Pfleeger, "*Software Engineering"*, Practice Hall, Inc., 1998.

[Binder, 99] Robert V. Binder, *"Testing Object-Oriented System"*, Addison-Wesley, 1999.

[Sommerville, 95] Ian Sommerville, "*Software Engineering"*, Addison-Wesley, fifth edition, 1995.

[Rosenquist & Bruck, 98] Patrik Rosenquist & Kristina Adelswärd Bruck, Master Thesis: *"Automated Testing Of Software With GUI"*, 1998.

[Kit, 95] Edward Kit, "*Software Testing* (in the real word)", Addison-Wesley, 1995.

[Regnell ed al, 98] Björn Regnell, Per Beremark, Ola Eklundh, "*Requirements Engineering For Packaged Software*", 3(2): 121-129, 1998.

[Whittaker, 00] James A. Whittaker, "What Is Software Testing? And why Is It So Hard?", 70-79, IEEE Software January/ February 2000.

[Kaner ed al, 99] Gem Kaner, Jack Falk, Hung Quoc Nguyen, "*Testing Computer Software*", John Wiley & Sons,inc , second Edition, 1999.

[Beck, 99] Kent Beck, “*extreme programming*”, Addison-Wesley, 1999.