AUTOMATED GUI TESTING OF LOW-RESOURCE EMBEDDED SYSTEMS

BY

JESPER GAARSDAL
201201700

JACOB EMBORG SØNDERSKOV
20082675

MASTER’S THESIS
IN
TECHNICAL INFORMATION TECHNOLOGY

SUPERVISOR: STEFAN HALLERSTEDE
Aarhus University School of Engineering
June 2, 2014

Jesper Gaarsdal
201201700

Jacob Emborg Søunderskov
20082675

Stefan Hallerstede
Supervisor

FACULTY OF SCIENCE
AARHUS UNIVERSITY
Abstract

Since the coming of age of smartphones, the general perception of embedded GUI applications has changed significantly. While the resources available are reduced in order to decrease development costs, the applications are expected to deliver a GUI that lives up to the standard set by modern smartphones. The addition of increasingly sophisticated GUI applications increases the need for structured test automation. Embedded GUI testing however, especially on low-resource devices, is a field of limited research.

Through this thesis, the challenges of GUI testing on low-resource embedded devices are explored and addressed in the conceptual design of a GUI testing framework. The design has been devised by combining lessons learned from related work in embedded test automation as well as state of the art GUI testing frameworks. The design is validated by implementation of a proof-of-concept prototype evaluated through a standardized process.

The embedded applications targeted by the GUI testing framework are based on the lightweight GUI framework TouchGFX and produce a code size in the range of tens of kilobytes.

The result of this thesis is a feasible conceptual design validated through a prototype capable of embedded GUI testing on low-resource devices. The conceptual design produces minimal overhead in terms of resource consumption, while automating simple as well as complex test scenarios on a TouchGFX application. Furthermore, the design presented in this thesis provides a well founded basis for future work within this field.
Acknowledgements

First, we would like to thank our supervisor Stefan Hallerstede for his tireless attention and advice during this thesis and Master’s degree.

Additional thanks go to Jesper Hedegaard and Søren Snehøj Nielsen from Mjølner Informatics A/S. The TouchGFX framework, developed by Mjølner, has provided a great environment for trying our hypothesis, and the input and support from Jesper and Søren has been invaluable.

Finally, we would like to thank family and friends for their patience and support during this work.
Table of Contents

Abstract i
Acknowledgements iii
Table of Contents v

Chapter 1 Introduction 1
  1.1 GUI-based embedded systems 1
  1.1.1 Challenges of embedded software development 1
  1.1.2 Embedded software testing 2
  1.2 Motivation 2
  1.3 Thesis work 3
  1.3.1 Goals 3
  1.3.2 Approach 5
  1.3.3 Scope 5
  1.4 Reading guide 6
  1.4.1 Terms and abbreviations 6
  1.4.2 Structure 6

Chapter 2 GUI Testing of Embedded Systems 9
  2.1 Overview 9
  2.2 GUI testing 9
  2.2.1 Principles of GUI testing 10
  2.2.2 Automated GUI testing 11
  2.2.3 Considerations of automated GUI testing 15
  2.3 Embedded GUI Test Automation 16
  2.3.1 Challenges of embedded GUI test automation 17
  2.4 TouchGFX 18
  2.4.1 Application layer 18
  2.4.2 TouchGFX core 20
  2.4.3 Hardware Abstraction Layer (HAL) 22

Chapter 3 An Automated GUI Testing Framework 25
  3.1 Overview 25
  3.2 Assumptions 25
  3.3 Related work 26
  3.3.1 Work by Jacobs et al. 26
  3.3.2 Work by Lin et al. 27
# Table of Contents

3.3.3 Work by Wu et al. .................................................. 28
3.3.4 Comparison of related work ...................................... 28
3.4 Designing a GUI testing framework ................................ 29
  3.4.1 The overall design ............................................. 29
  3.4.2 Robot Framework ............................................... 31
  3.4.3 Test library .................................................... 32
  3.4.4 TestGFX on the host-side .................................... 33
  3.4.5 Dynamics of the host .......................................... 36
3.5 Discussion .......................................................... 39
  3.5.1 Comparison to related work ................................... 39
  3.5.2 Comparison to thesis goals ................................... 39

Chapter 4 Test Automation on Low-Resource Embedded Systems ... 41
  4.1 Overview ............................................................ 41
  4.2 Assumptions ........................................................ 41
  4.3 Related work ....................................................... 42
    4.3.1 Work by Kervinen et al. .................................... 42
    4.3.2 Work by Loft ................................................ 43
    4.3.3 Comparison of related work ............................... 44
  4.4 Designing embedded test automation ................................ 45
    4.4.1 The target design .......................................... 46
    4.4.2 TestGFX on the target-side ............................... 46
    4.4.3 TouchGFX integration .................................... 48
    4.4.4 Dynamics of the target ................................... 51
    4.4.5 TestGFX communication ................................... 54
  4.5 Discussion ........................................................ 55
    4.5.1 Comparison to related work ............................... 55
    4.5.2 Comparison to thesis goals ............................... 56

Chapter 5 Validation of an Automated GUI Testing Framework ... 57
  5.1 Overview ............................................................ 57
  5.2 Evaluation process ............................................... 57
    5.2.1 Quality properties ......................................... 58
    5.2.2 Quality requirements and measurements .................. 58
  5.3 Validation of GUI testing on low-resource embedded systems ... 62
    5.3.1 Proof-of-concept prototype ............................... 62
    5.3.2 Test environment .......................................... 72
    5.3.3 Results ..................................................... 77
  5.4 Discussion of results ........................................... 79

Chapter 6 Concluding Remarks and Future Work ......................... 81
  6.1 Overview ............................................................ 81
  6.2 Results achieved .................................................. 81
    6.2.1 Host design ................................................ 82
    6.2.2 Target design ............................................. 83
    6.2.3 Evaluation ................................................. 84
  6.3 Future work ....................................................... 86
    6.3.1 GUI survey tool .......................................... 86
    6.3.2 Record and playback ..................................... 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.3</td>
<td>GUI framework abstraction</td>
<td>86</td>
</tr>
<tr>
<td>6.3.4</td>
<td>GUI event synchronization</td>
<td>86</td>
</tr>
<tr>
<td>6.3.5</td>
<td>General improvements</td>
<td>86</td>
</tr>
<tr>
<td>6.4</td>
<td>Personal learning outcome</td>
<td>87</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Parallel research and development</td>
<td>87</td>
</tr>
<tr>
<td>6.4.2</td>
<td>An engineering perspective</td>
<td>88</td>
</tr>
<tr>
<td>6.5</td>
<td>Final remarks</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td><strong>Bibliography</strong></td>
<td>89</td>
</tr>
<tr>
<td></td>
<td><strong>Appendices</strong></td>
<td>93</td>
</tr>
<tr>
<td>A</td>
<td>Terminology</td>
<td>95</td>
</tr>
<tr>
<td>B</td>
<td>Implementation of the Proof-of-Concept Prototype</td>
<td>97</td>
</tr>
<tr>
<td>C</td>
<td>Results of the Prototype Evaluation</td>
<td>119</td>
</tr>
<tr>
<td>D</td>
<td>Overview of the Thesis Work</td>
<td>127</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter introduces the context of this thesis and presents its background and goals. The approach we have taken during our thesis work is also introduced along with the scope of the thesis.

1.1. GUI-based embedded systems

Before the age of the smartphone, the term *Embedded System* was often associated with headless devices wired into large electrical systems or circuits and micro controllers connected to a button panel. A smartphone is a pocket-sized complete system of user interfaces, sensors, and other hardware. They are rich on resources in terms of memory and processing power and are capable of running general purpose software. Embedded systems are rarely complete but rely on external systems. They are designed for a specific purpose and are generally constrained on resources. Smartphones are thus considered to be somewhere between embedded systems and general purpose computers and have greatly influenced the development of embedded systems [39]. Embedded Control Units (ECU) with touchscreen monitors and rich Graphical User Interfaces (GUI) are becoming increasingly common, which promotes the development of graphics frameworks for these devices.

While embedded graphics frameworks, such as Qt\(^1\), existed before the smartphone, Android, and iOS in particular have changed our expectations of embedded GUI applications and given birth to many new technologies.

The focus the smartphones have brought on user friendly and efficient GUIs, running on yet smaller devices, has provided a wide range of technologies for creating sophisticated user interfaces with touchscreen support for small embedded systems.

1.1.1 Challenges of embedded software development

Unlike desktop and laptop computers an embedded system is dedicated to a specific purpose. This enables the use of smaller platforms with limited resources minimizing size as well as production costs. This limitation introduces several concerns. Limited memory limits the code size, which in turn constraints system flexibility and extensibility. The limited memory in combination with limited processing power also reduces performance of internal and external communication [22]. Resource limitations are not the only challenge however. As embedded systems integrate software and hardware, the physical device plays an integral part of software design. Physical systems

\(^1\)http://qt-project.org/
are naturally concurrent and often real-time, which makes performance and temporal metrics an essential part of embedded software development.

A common application of embedded systems is in safety- or mission critical systems, making reliability a priority to a much higher extent than for general purpose software. This condition, in combination with the inherently concurrent environment, is a key challenge of embedded software development [20].

1.1.2 Embedded software testing

Embedded systems are often produced in large quantities, and once they reach the customer they can be difficult to update or patch. This makes testing, whether it be manual or automated, an important aspect of embedded software development.

Testing embedded software has much in common with general purpose software testing. The aim is to detect defects in a system and reduce risk, where risk can be defined as the expected cost due to operational failures [16]. The differences exist in the challenges described in Section 1.1.1 and in the fact that embedded software may be run on a vast number of different platforms. This heterogeneous environment hampers the creation of generic technologies for embedded test automation, making the common approach an *ad hoc* mix of manual testing and custom-made frameworks for test automation of a specific application or platform [18, 49]. Desktop OS-based simulators enable the use of proven tools that exist for general purpose software, but they often fail to simulate the resource- and temporal constraints of the target platform, which makes them viable for only the strictly functional test cases [19, 30, 40].

1.2. Motivation

The amount of research made within the field of GUI testing has increased in recent years. The advent of smartphones and touchscreen-based apps presented a need for better GUI test automation [48]. Manually performing the same test cases on many different devices and after every major change is a tedious task.

Research in the field of embedded test automation has also seen an increase. As dictated by Moore’s law, yet smaller systems are becoming increasingly more powerful, and the tasks demanded from embedded systems today are both complex and versatile [43]. Relying only on manual testing is for the most part infeasible [19]. The methods and tools available for testing the GUI of smartphones however, are not well suited for general embedded systems. Many have strong ties to their respective OS and most work under the assumption that they have the memory capacity and processing power of a smartphone.

The motivation for this thesis is the lack of research and technologies for testing the GUI on embedded systems with limited resources. TouchGFX², developed by Mjølner Informatics A/S, is a framework for developing touch-based GUI applications on low-resource embedded systems. It is also the GUI framework on which we will build our proof-of-concept prototype. Including an OS, such as FreeRTOS [3], TouchGFX will consume in the range of 21-40 KB ROM (not including graphics data, text strings etc.) and 11-35 KB RAM, depending on application size. Building upon existing research within automated testing on low-resource devices along with proven methods and tools for GUI testing in a general purpose software environment, we believe that we can provide a solution capable of performing elaborate GUI testing on these TouchGFX-based devices.

²http://mjolner.dk/services/embedded-solutions/touchgfx/
1.3. Thesis work

1.3.1 Goals

Personal goals
Our personal goals for this work are:

1. To improve our skills and knowledge in GUI testing of embedded systems, test automation in a low-resource environment, and in the design of a GUI testing framework.

2. To improve our ability to research a scientific subject from different perspectives and to contribute to the scientific community.

Scientific goals
This thesis is based on the hypothesis, that we can perform automated GUI testing on low-resource embedded devices. In relation to this, the goals of this thesis are to design a framework capable of:

1. performing automated GUI testing of embedded applications running on a remote target device or in a simulated environment.

2. automating simple as well as complex test cases on a low-resource device.

3. evaluating and reporting on test cases executed remotely.

In order to validate our framework against the goals set for this thesis work, an evaluation process based on the ISO/IEC 25000 set of standards for software quality requirements and evaluation is used [5, 12]. From the quality model of the ISO/IEC 25010 standard [8], five quality properties, presented in Section 1.3.1.1, are chosen and for each of these properties a set of requirements and measurements, based on the ISO/IEC 25030 and 25020 standards respectively [6, 7], are specified. In order to validate the measurements a proof-of-concept prototype of the framework is developed.

This prototype is used to execute a test suite on a TouchGFX application, both developed for this purpose. The validation of the measurements provide the results of the evaluation process. For each of the quality properties the results of the measurements are presented in a normalized bar chart, described in Section 1.3.1.2.

The evaluation process, prototype, test environment and results are all described in further detail in Chapter 5.

1.3.1.1 Property description and justification

The framework is evaluated on five properties selected from the ISO/IEC 25010 product quality model. These properties and the reasoning behind them are presented below:

Functional suitability The degree to which a product provides functions that meet the stated as well as implied needs under specified conditions. Sub-properties are appropriateness (the product provides an appropriate set of functions for specified tasks and user objectives) and accuracy (the product provides the specified results with the needed degree of precision and accuracy).

This property is important as it covers the core ability of the framework to implement the test cases necessary to properly test the Application Under Test. As a result, it also covers
Chapter 1. Introduction

the maintainability of the test cases, in terms of robustness against change in the Application Under Test and the ease with which test cases can be extended or altered.

**Performance efficiency**  The degree to which a product meets requirements for performance relative to the amount of resources used under specified conditions. The sub-properties are *time behavior* (response- and processing times as well as throughput rates) and *resource utilization*.

The target domain of our framework is low-resource embedded platforms and as such, resource utilization is an important property to evaluate. In test automation, time behavior is an important property as latency and delay of events may yield different results [30].

**Compatibility**  The degree to which a product can interoperate with other products and perform its required functions while sharing hardware or software platform. The sub-properties include *co-existence* and *interoperability*.

For any test framework the ability to co-exist with the Application Under Test is a very important property. For embedded test automation in particular, the ability of the test framework to run on the target platform under resource constraints without influencing the execution of the Application Under Test is crucial.

**Maintainability**  The degree to which a product can be effectively and efficiently modified by the intended personnel. Sub-properties include *modularity*, *reusability*, *analyzability* (the impact of change in a product can be assessed or deficiencies and causes of failures can be detected), *modifiability* and *testability* (criteria for a product can be established and tests to determine whether those criteria have been met can be performed).

Where functional suitability covers the maintainability of the test cases, this property covers the maintainability of the test framework, in terms of the sub-properties stated above. Another important aspect of this property is the ability to extend the functionality and environment support provided by the framework.

**Transferability**  The degree to which a product can be transferred from one environment to another. Sub-properties include *portability* and *adaptability* (the product can be adapted for different environments, including scalability of screens and data transactions).

For embedded software this is an important property due to the large selection of heterogeneous platforms. For a GUI testing framework, adaptability in particular is relevant as GUI objects may change size or position depending on the size of the monitor.

### 1.3.1.2 Property rating

For each of the five quality properties a set of requirements are specified and for each requirement a set of measurements are taken. The quality requirements and measurements are described in Section 5.2.2.

The measurements are all simple statements that may be answered with a yes or a no and the results of the evaluation are presented in a stacked bar chart, where each property is represented by a bar. Each bar contains a stack of successful measurements and a stack of failed measurements making up 100% of the measurements taken for that property. This method of presentation is chosen as it emphasizes the contributions of the succeeded and failed measurements to the total score of a quality property.

The results of the evaluation are presented in Section 5.3.3.
1.3.2 Approach

The solution presented in this thesis is designed through a series of steps in two concurrent branches, as illustrated by Figure 1.1. The branches are a result of a decision to partition the test framework on a host computer and a target device. This design decision is described in detail in Chapter 3.

![Figure 1.1: Flowchart of the thesis approach.](image)

**Research** During this step, scientific research is performed within the field of embedded GUI test automation. The research made, along with an analysis of existing methods and technologies, serves as a feasibility study of our thesis goals and will aid in the development of a prototype for the validation of our design.

**Elaboration** In this step, the GUI testing framework is designed based on the output from the research along with a study of the enabling framework, TouchGFX, and hardware platforms.

**Evaluation** For this step, a prototype is produced in order to validate the design through static analysis as well as experimentation using a test suite executed on a TouchGFX application.

In the initial phase of the thesis work the primary objective, automated GUI testing of low-resource embedded systems, is divided into two problem domains: GUI testing remote applications and low-resource embedded test automation. For each branch the above steps are performed dependent on the other as they make up the parts of a single solution.

1.3.3 Scope

The scope of this thesis is limited to automated GUI testing of low-resource embedded devices running TouchGFX applications in a FreeRTOS OS or in a simulated environment utilizing the SDL library³.

The level of GUI testing performed as part of the framework evaluation is purely functional, while GUI performance testing is considered in the design of the framework.

As the front-end of the solution Robot Framework⁴ has been selected and the reasoning behind this decision is covered in Chapter 3. The development of a user interface for a test automation tool, including the scripting language used in test specification, is thus considered out of scope for this thesis.

The limitation to TouchGFX applications and the decision to use the Robot Framework mean that quality properties from the ISO/IEC 25010 standard regarding *operability* and *reliability* will not

---

³http://www.libsdl.org/
⁴http://robotframework.org/
be part of the framework evaluation performed in this thesis. *security* is likewise considered out of scope.

The design of the GUI testing framework presented in Chapter 3 relies on structured state models generated by a GUI survey tool. The design and implementation of this tool is considered out of scope, although a possible implementation is discussed in Chapter 6.

### 1.4. Reading guide

#### 1.4.1 Terms and abbreviations

**Emphasis:** Concepts, words, or sentences with special relevance are emphasized in the text by the use of an *italic* typeface.

**Terminology:** Significant terms and abbreviations used throughout this thesis are listed in Appendix A.

Throughout this thesis the terms System Under Test (SUT) and Application Under Test (AUT) will be used depending on the scope of the test entity in question. Furthermore, the terms host and target will be used for a connected PC and SUT respectively. The relationship between these terms is illustrated by Figure 1.2.

![Figure 1.2: The composition of a System Under Test including the Application Under Test](image)

#### 1.4.2 Structure

This thesis is divided into six chapters and four appendices. Figure 1.3 shows an overview of the chapters and appendices with arrows indicating suggested reading order.

**Chapter 2:** This chapter contains the general theory, primary concepts and relevant technologies for the thesis.

**Chapter 3:** This chapter presents an automated GUI testing framework for remote test execution. It begins with a presentation of assumptions and related work followed by a description of our contribution and ending with a discussion of the work presented.

**Chapter 4:** This chapter will, extending upon the solution presented in Chapter 3, present a solution for automated GUI testing on low-resource embedded devices. It begins with a presentation of assumptions and related work followed by a description of our contribution and ending with a discussion of the work presented.
Chapter 5: This chapter validates the framework design established in chapters 3 and 4, including a description of a proof-of-concept prototype.

Chapter 6: This chapter concludes the thesis, summarizing and discussing the results. Areas of future work are also presented along with our personal learning outcome.

Appendix A: This appendix presents the terminology used in this thesis.

Appendix B: This appendix contains the details of the implementation of the proof-of-concept prototype presented in Chapter 5.

Appendix C: This appendix contains the details of the results of the evaluation process presented in Chapter 5.

Appendix D: This appendix presents an overview of the thesis work in terms of the work distribution between the two authors and a rough outline of the progression of the thesis work.
Figure 1.3: Overview of the thesis structure.
GUI Testing of Embedded Systems

This chapter presents existing concepts and technologies used for achieving the thesis goals presented in Section 1.3.1. The contents of this chapter serve as background material and basis for the automated GUI testing framework presented in chapters 3 and 4.

2.1. Overview

This chapter describes the background theory as well as the state of the art technologies necessary in order to understand the thesis work and the motivation behind it. Section 2.2 presents the core concepts of GUI testing and Section 2.3 presents principles and methodologies specific to embedded GUI test automation.

2.2. GUI testing

Software testing can be performed on different levels and the levels involved depend on the nature of the software. Figure 2.1 presents a hierarchy for GUI application testing. The first level of testing is unit testing, which verifies the smallest testable elements in a system. Next is integration testing, where the interfaces between components are verified. At the highest level of testing in this hierarchy is GUI testing, where user events are exercised on the complete, integrated system.

As with general purpose software testing, the goal and purpose of GUI testing is to detect defects - not to show their absence [11]. The aim of test case specification is then to expose any defects present in the System Under Test (SUT). The purpose of a GUI is to facilitate user interaction and because of this, GUI testing is often performed in a manual, ad-hoc fashion [10]. However, as a system matures and the GUI becomes
more complex this kind of manual exploratory testing becomes inadequate. A structured approach
towards GUI testing has to be introduced.
This section will review principles and methods of structured GUI testing as well as approaches
used in existing GUI testing tools.

2.2.1 Principles of GUI testing

GUI testing is quite different from testing the business logic of software. It requires the ability
to recognize GUI objects, automate events, provide input, and validate graphical representations.
For this reason, GUI testing should be separated from other testing efforts [34].
As with business logic testing, GUI testing may be performed on different levels from validating
low-level graphical operations to simulating events on a finished product, where the first is often
performed by the GUI framework vendor [40].

2.2.1.1 A GUI testing procedure

A structured GUI test consists of the same four building blocks used for business logic testing;
*specification, execution, validation,* and *reporting,* as presented in Figure 2.2.

![Figure 2.2: Overview of a software testing procedure.](image)

1. **Specification** For any structured test effort a test specification must be written. In order to
do this one must first determine what to test, what level of testing will be performed and
what kind of coverage will be pursued [38]. Next, the format of the test specification must be
determined. For manual testing user scenarios are the common approach, but for automated
testing there are several other methods, some of which are described in Section 2.2.2.

2. **Execution** Based on the specification test execution is performed. Test execution may be
manual or automated, using specialized software capable of influencing the SUT. For a GUI
test this includes simulating user input or injecting GUI events into the AUT.

3. **Validation** The purpose of testing is to validate that the SUT behaves as expected. Following
the test execution, results and the current state of the system must be validated against
the expected results defined in the specification. For manual testing this is straightforward,
but for automated testing this requires advanced methods, described in Section 2.2.2.

4. **Reporting** In the case of automated testing it is crucial that the software executing the tests
is able to report back to the developers. In a Continuous Integration (CI) environment,
tests are often executed during nightly builds, and when assessing the results the developer
should from the report not only be able to recreate a failing scenario, but also be able to
deduce the time and the place in the source code, where the defect occurred.

Error reports in GUI testing are similar to those found in business logic testing, except that
they may also include a screenshot of the GUI in its faulty state.
As mentioned earlier, it is still quite common to perform manual GUI testing. Both exploratory and structured manual testing can be valuable, but relying solely on manual testing for complex GUI applications is infeasible. The event- and state-space increases rapidly and regression testing by means of a test suite, large enough to achieve the coverage necessary to establish the targeted quality, has to be automated [37].

### 2.2.2 Automated GUI testing

Some of the core principles of software engineering are the reuse of assets and automation of repetitive tasks. Regression testing is based on repetition, and automation can not only save resources spent on testing, it can also increase the overall effectiveness of regression testing as CI environments can execute tests automatically after each commit.

One of the concerns of automation in general is the amount of time spent on writing and maintaining the automation software compared to the time spent manually performing the task. This section will present several methods to automated GUI testing and while they are all resource intensive to develop, the effort spent on maintaining a test suite differ significantly.

Section 2.2.2.1 presents five levels of GUI testing frameworks and Section 2.2.2.2 presents a model-based approach to GUI testing.

#### 2.2.2.1 Levels of GUI testing frameworks

When developing a GUI testing framework the method of test specification must be considered as it is crucial to the design and use of the framework. This has lead to five levels of GUI testing frameworks, also described as evolutionary stages [21, 23, 26], based on the method of specification. These five levels are: *record and playback*, *scripting*, *data-driven scripts*, *actionword- and keyword-scripting*, and *scriptless*.

**Record and playback**

Record and playback is a method for which the test developer *records* the manual execution of a test case. The recording can then be replayed by the test framework and the behavior can be evaluated against the recorded *correct* behavior.

The recorded scenario must be persisted and a common approach is to generate a script, which can then be modified by the test developer. For this reason, record and playback is often used together with script-based or even scriptless GUI testing frameworks [26].
While this approach is very easy to use, it has several limitations:

- **High complexity and redundancy** in the generated scripts is common, as it shares the same problems as other code-generation technologies. Several test cases may share several steps that could be reused in order to increase maintainability, but instead test developers have to manually maintain each individual script [26].

- **Vulnerability to change** is a problem usually associated with record and playback frameworks. The GUI may be the most volatile component in a software product, and while it is not the only approach, many tools still utilize coordinates of GUI objects. Even a small redesign of the GUI can break entire test suites, which must then be repaired [4, 23, 33].

- **Timing issues** are common when relying on record and playback [1, 26, 27, 30]. The time it takes for a GUI object to load can change significantly between executions and while many frameworks support waiting for a certain element to become visible, the element may still not be properly loaded.

While relying solely on record and playback has several shortcomings and issues, it can be a valuable feature for a script-based or scriptless GUI testing framework. Scripting languages used for GUI testing can be complex and developing test cases in scriptless frameworks is often very time consuming. The ability to generate even a draft of a test case, by recording, can greatly reduce time and effort for the test developer.

**Scripting**

Script-based frameworks are based on the concepts of Domain-Specific Languages (DSLs). A DSL is a programming language of limited expressive power specific to a certain domain, in this case GUI testing. The limited expressive power is one of the greatest advantages of using a DSL as it provides abstraction, simplicity, and makes it more difficult to write something that is wrong [15]. Script-based frameworks provide high flexibility and depending on the scripting language used, they can also integrate well with business logic testing environments.

Script-based frameworks often provide great modifiability and reusability and integrated into a CI environment, they can be a valuable asset to the testing effort. However, in teams with designated testers without programming experience, script-based frameworks are less useful. The scripting languages used can be complex and require knowledge of the internals of the SUT, which defeats the purpose of designated testers.

The script-based method greatly resembles the playback half of a record and playback framework, as seen in Figure 2.3, and a recording tool is then just an extension in order to increase usability for designated testers. It is difficult however, for a test script generator to utilize the modifiability and reusability of script-based approaches [48].

**Data-driven scripts**

Data-driven scripts leverage the reusability of script-based testing in order to execute the same scripts but with different input and expected results. Input and expected results are, unlike in script-based frameworks, not hard-coded in each test case but separated into data tables. Test cases are then generated based on sets of data rather than on test scripts.

Data-driven scripting can, by abstracting the test data from the test implementation, alleviate the usability issues of script-based testing. As illustrated in Figure 2.4, by relying on developers to implement modules of testing logic, designated testers can create test cases in a more scriptless fashion [23].
Another benefit to this approach is that it enables software testing methods such as combinatorial testing, equivalence class partitioning and boundary value analysis. These methods can uncover many defects that would be difficult to detect otherwise [41].

**Actionword- and keyword-scripting**

Actionword- and keyword-scripting, or Action-Based Testing (ABT), is another approach that extends on script-based testing [9, 26]. While test data is not necessarily abstracted from test implementation, actionword- and keyword-scripting is also based on data-driven testing as it further utilizes the modularization of test functionality. Actionword- and keyword-scripting enables the test developer to create hierarchies of reusable test modules. Keywords are identifiers for low-level modules, such as “Click button <name>”, relying on parameters from higher-level modules. Actionwords are identifiers for high-level modules that may specify entire user scenarios. Actionwords thus consist of keywords or other actionwords, resulting in a tree-based hierarchy.

The main benefit of this approach is reusability and thus maintainability, as simple changes to the GUI only require changes to parameters or low-level keywords [37]. Similar to data-driven testing, this approach provides abstraction, which can help remedy the usability issues of script-based testing, as low-level keywords can be implemented by developers and utilized by designated testers in the formulation of actionwords and test cases, as illustrated by Figure 2.5. The main concern of this approach however, is the time spent on implementing a sufficient set of keywords [23].

**Scriptless**

A scriptless GUI testing framework is in many cases a GUI on top of an actionword- and keyword-script framework based on the same tree hierarchy of test modules. These frameworks retain the reusability and maintainability of actionword- and keyword-scripting with increased usability for designated testers without programming experience or knowledge of internals of the SUT [21, 23]. The frameworks include a set of built-in low-level keywords such as “Click button <identifier>”, where the identifier is a logical name for a component used during test specification. From these innate keywords, designated testers build a hierarchy of modules by drag and drop and by specifying parameters. As illustrated by Figure 2.6, designated testers are thus able to create and maintain an entire test suite without relying on developers.

As it is assumed that users of script-based frameworks have some knowledge of the internals of the SUT, the identification of a specific GUI object is often based on an internal component id. Scriptless frameworks are designed for designated testers and the recognition of GUI objects is...
therefore often more intelligent and based on more than one attribute. Some scriptless frameworks support test-driven development through the use of logical component names as mentioned above. This abstraction of GUI objects enables test cases to be specified independently of the SUT and later the logical components can be bound to the physical components by a GUI survey tool. These tools often collect several attributes of a GUI component and associate these with a logical identifier [26]. Such attributes may include an internal id, any text visible on the component, a location, and even size. Relying on several attributes for GUI object recognition, referred to as heuristics-based GUI object recognition, the tests are much more robust against change, which increases maintainability of the test scripts [21, 33].

2.2.2.2 Model-based GUI testing

A system model is a description of the behavior or structure of a system. It provides abstraction and is often utilized during the early phases of development of complex and critical systems. Models are simpler and easier to validate than the systems they describe, which has led to the creation of model-based testing [42]. Depending on the form of the model used, model-based testing may include several different activities such as test case generation, system state validation or system design validation. Test developers will often focus on models describing system behavior, enabling design validation or test case generation [9, 45]. Most often this entails using a formal modeling language such as the Vienna Development Method (VDM) or Z notation [13, 46]. Such models can be validated through discrete mathematics and possible test cases can be generated
from a finite-state automaton, describing the relationships between states, actions and transitions [34, 37]. Figure 2.7 gives an example of a state model including five possible states (S0-S4) interrelated by ten transitions (T0-T9).

![State Model](image)

Figure 2.7: Example of a state model used in model-based testing.

Model-based GUI testing is mostly concerned with test case generation and state validation and may utilize both behavioral and structural models for this. Test cases may be generated from behavioral models, such as Figure 2.7, where each state may be a specific screen or window. Each screen may be described by a structural model of GUI objects, enabling validation by simple comparison of the expected hierarchy of GUI objects given the transitions that has occurred and the actual hierarchy presented by the AUT.

Model-based testing is quickly becoming a popular approach to test automation. While it can be complex and expensive to develop, it promises a higher maintainability than other methods [27]. Test cases can be automatically generated, covering every possible state and state transition, and the state of a GUI application is easily expressed and validated through a model.

### 2.2.3 Considerations of automated GUI testing

This section will shortly sum up the important considerations and issues raised throughout Section 2.2.2. Some new considerations, such as GUI performance testing, will also be introduced.

**Test case generation** This concept was introduced with record and playback frameworks in Section 2.2.2.1, where test cases are generated from user sessions recorded by a tool. Test case generation can be automated even further however.

Some tools are capable of automatically generating entire test suites, providing excellent coverage with very limited effort as presented with model-based testing in Section 2.2.2.2, where test cases are generated from a behavioral model describing possible states and transitions. However, while these tools can greatly decrease the overall testing effort, they can also produce significant overhead and redundancy in test scripts [10].

Further theory and research in automated test case generation is considered out of scope for this thesis. When comparing script-based and scriptless testing, it is important however, to keep in mind that script-based frameworks are generally much easier to extend with such features.
**Reusability** Due to the volatile nature of the GUI, tests will always require some measure of maintenance. Reusable test modules can greatly reduce the maintenance effort. Most script-based and scriptless frameworks have natural support for reusable components [23]. Test case generation tools, such as record and playback, have difficulty reusing generated code, resulting in redundancy and poor maintainability [26].

**Synchronization and timing schemes** Several record and playback and some script-based frameworks rely on simple sleep methods to wait for the state of the GUI to settle [26]. Fixed sleep durations are a very unreliable approach and more intelligent synchronization schemes are necessary.

The challenge of waiting to provide input only when the GUI is in the correct state is referred to as the *temporal synchronization problem*. Some solutions are still based on sleep functions, but will adjust the sleep duration during test execution [30], while other solutions include more intelligent functions capable of waiting for the GUI to be in a certain state.

**GUI object recognition** The maintainability of a GUI testing framework is heavily affected by its ability to recognize a GUI object across releases of an application [33]. Several record and playback frameworks rely on coordinate-based event invocation, e.g., move the mouse to a specific coordinate and click the left mouse-button. Instead of attempting to identify the GUI object that should be clicked, the area that the object has been known to occupy is clicked. Relying on such volatile properties, a simple redesign of the GUI can break an entire test suite [21, 23, 26].

In order to increase maintainability, attributes of a GUI object can be collected or specified at design-time and used during test execution to identify the object in question, as described with scriptless testing in Section 2.2.2.1.

**Test verification granularity** Depending on the format of a test case specification and on the underlying test engine, expected results may be verified at several points in time. Test verification granularity is a measure of the amount of intermediate assertions made during test execution.

While this subject is out of scope for this thesis, it should be considered during framework design whether only the final result of a test case can be verified or assertions can be made in between events [49].

**GUI performance testing** Related to the concept of *perceivable performance* [25] the reaction time of the system should be measurable. Defined as the latency of event handling, excess time taken for a user input to be properly handled can be the source of great frustration and be as devastating to a software product as functional defects [44]. Performance testing is out of scope for thesis.

### 2.3. Embedded GUI Test Automation

A general purpose software application is most often developed and executed inside a standardized OS with several abstraction layers between it and the enabling hardware. This is seldom the case for embedded applications, and as the hardware available can change significantly between devices developing any generic tool or framework for embedded applications is a challenge [29].
Challenges of embedded GUI test automation

The lack of abstraction in embedded software development has made embedded test automation an ongoing field of research. Embedded GUI testing however, is a field of limited research. While the challenges are similar, the concerns regarding GUI testing, described in Section 2.2.3, provide unique issues, presented in Section 2.3.1.

2.3.1 Challenges of embedded GUI test automation

The difficulties of embedded GUI testing are, as with embedded business logic testing, based on the lack of abstraction in the software and the resource constraints of the platform [29, 35].

Heterogeneous environment Embedded platforms come in a wide range of shapes, sizes, and capabilities and although we are able to produce high-end platforms, such as smartphones, embedded products are often designed for a single task and produced in such quantities that smaller and cheaper devices are utilized in order to minimize costs [24]. However, in order to support a large variety of business cases, it is necessary to provide support for several of these specialized platforms.

An AUT that may run on several hardware platforms relying on different implementations of device drivers and other software interfaces is a great incentive for automated regression testing. At the same time it makes it difficult to develop such automation software. The same test cases must execute under different constraints and assumptions. The framework must be sufficiently lightweight as to support the smallest of platforms memory-wise. Intelligent timing schemes, mentioned in Section 2.2.3, are necessary in order to adapt to the possible differences in processing power. GUI object recognition, also mentioned in Section 2.2.3, has to take different screen formats into account, rendering the position and size attributes of GUI objects even more volatile.

Limited resources While it is possible today to create pocket-sized devices capable of performing the same tasks as a small laptop, these are expensive and smaller platforms are therefore used in order to reduce production costs. This means embedded developers must solve the same problems under the same performance requirements but with less code and less processing power. The strain put on the resources that are available can lead to changes in program execution, introduced by latency and events that are missed [30]. With an automated test framework, stress testing could be performed in order to identify such defects.

Tests aimed at detecting defects regarding performance and reliability are best executed on a device as it is difficult to simulate resource constraints [40]. This means the test framework is under even stricter resource constraints as it must execute alongside the AUT with minimal impact on its performance.

High complexity As embedded platforms become increasingly powerful with added features and responsibilities, the associated software increases in complexity. A low level of abstraction and a heterogeneous platform also adds to the complexity of the software, to a point where manual testing is no longer feasible. Testing must be performed in an incremental and controlled fashion [24], and the high complexity along with the volatile nature of a GUI necessitates automated regression testing [35].

Low abstraction Abstraction is a key issue in embedded test automation and it correlates with all of the other issues presented in this section. Resource constraints limit the level of abstraction attainable and limited abstraction leads to an increased complexity. Furthermore abstraction would be a solution for the heterogeneity of embedded devices.
Low abstraction makes it difficult to develop generic tools. Even a common framework for a family of applications may be infeasible due to the number of different hardware platforms that must be supported.

In the case of GUI testing, an abstract representation of GUI elements and events is necessary for test automation [35]. This abstraction does not always exist between the AUT and graphics framework however, resulting in Built-In Test (BIT) code or specialized tools [29].

The four challenges presented above are correlated. A low level of abstraction will result in a higher complexity and limited resources will result in a low level of abstraction, as software abstraction takes up memory. This means that an embedded test framework must consider all four issues in its solution.

Chapters 3 and 4 present our design of an embedded GUI testing framework. Based on related work presented in these chapters and concepts described in this chapter, our solution attempts to overcome the challenges presented here through software design. This design is based on TouchGFX as the GUI framework of the AUT.

2.4. TouchGFX

TouchGFX is a graphics framework that enables the creation of sophisticated GUI applications on embedded systems with limited resources [2]. It is the graphics framework utilized by the AUT for this thesis work. This section will present the TouchGFX architecture including important concepts and methods relevant to our GUI testing framework.

The TouchGFX framework consists of two major parts, the TouchGFX core, and the Hardware Abstraction Layer (HAL), and provides important abstractions between the implementation of the GUI in the application layer and the target platform. This allows the application developer to focus on GUI and business logic rather than target-specific implementation. Figure 2.8 presents the architecture of the TouchGFX framework.

TouchGFX is designed for touchscreen interaction. User input is registered by target device drivers and sampled by the HAL in order to identify a specific touch event, described in Section 2.4.2.4. Upon identifying a touch event the HAL passes it to the TouchGFX core, which in turn invokes appropriate event handlers specified in the application layer.

The following sections will describe each layer of the TouchGFX architecture and discuss how TouchGFX addresses the challenges presented in Section 2.3.1 and how the considerations of automated GUI testing, presented in Section 2.2.3, could be handled.

2.4.1 Application layer

The application layer is responsible for implementing the presentation logic of the TouchGFX application. Presentation logic describes the source code concerned with how application data is presented to the user and how the user events are handled by the application. Additional logic concerned with the acquisition and manipulation of data is typically referred to as business logic. In general GUI development, actions are taken in order to separate presentation logic from the supporting business logic. This is also the case in TouchGFX application development. While employing the TouchGFX core, the application layer must be structured in compliance with a TouchGFX Model-View-Presenter (MVP) pattern, which provides means to structure and separate presentation and business logic.
2.4.1.1 TouchGFX Model-View-Presenter pattern

The MVP architectural pattern provides a structural separation of concerns in the presentation layer of a GUI application. Utilizing modularization, the typical design consists of three interrelated parts: a model, a view, and a presenter, which together constitute a single application screen. Several variations of the MVP pattern exist however, and the relations between the three parts may differ. A common variation is the Passive View MVP pattern [14], described as follows:

- The view holds a structure of GUI elements providing the definition of the GUI. User input is captured by the view and forwarded to the presenter making the view passive towards the input.
- The presenter is responsible for handling the forwarded user input and manipulating the model. Conversely, the model will notify the presenter of changes to the application state and the presenter will update the view.
- The model is responsible for storing and maintaining the state of the GUI application and notifies the presenter of any changes. Additional business logic such as external communication and data processing is implemented by a back-end component. The back-end uses the model in order to update the state of the application.

The TouchGFX MVP variant, presented in Figure 2.9, differs slightly from the Passive View MVP variant. It uses a single system-wide model that allows the application to maintain state across view transitions. This avoids reallocation of multiple and similar models minimizing resource consumption. Another difference between the two MVP variations, introduced by the system-wide model, is the separation of external communication and data processing business logic into...
the Back-end component, shown in Figure 2.8. A single model provides the back-end with a single access point to update the state of the application. It also provides a single access point for resetting the applications internal state, which is valuable for a GUI testing framework.

Figure 2.9: TouchGFX MVP pattern.

2.4.2 TouchGFX core

Besides the implementation of the MVP pattern, the TouchGFX core provides several visual components, or *drawables*, and touch event definitions, presented in Sections 2.4.2.3 and 2.4.2.4 respectively. Furthermore, the core provides means for the definition of transitions and application management, described in Sections 2.4.2.2 and 2.4.2.5 respectively.

2.4.2.1 Views and presenters

An application screen is implemented in two parts: a view and a presenter. While the presenter handles GUI events and information mitigation from the model to the view, the composition of the view is of primary focus when developing a GUI testing framework. Figure 2.10 gives an example of a TouchGFX application screen in a simulated environment.

Figure 2.10: TouchGFX example application in a simulated environment.

A view is composed of a number of visual components. It has an internal root container to which each component is added sequentially. This process forms a linked list of GUI objects, illustrated
Transitions

by Figure 2.11, and the result is a hierarchical object-oriented structure representing the current screen or state of the GUI.

The root container is utilized by our GUI testing framework in order to extract the state of the view. This process is described in Section 4.4.

2.4.2.2 Transitions

Changes between view and presenter pairs on run-time are referred to as transitions. A transition defines how the application switches from one view context to another. This entails deallocation of the current view and presenter and allocation of the new.

Transitions are implemented by transition-event handlers in the application layer, and in our GUI testing framework they are used in order to reset the application to an initial state, which is a crucial feature in a test automation framework. This process is also described in Section 4.4.

2.4.2.3 Visual components

Visual components are GUI objects which can be added to a view definition to constitute an application screen. They are provided by the TouchGFX core as a series of drawables specialized into containers and widgets.

- A container describes a visual component, which may contain widgets and other containers. In a tree hierarchy, a container may be considered a branch node. Notable container implementations provided by TouchGFX include the Container base class, the Scrollable Container and the List Layout.

- A widget describes a visual component with a specialized purpose, that can not contain child elements. In a tree hierarchy, a widget represents a leaf node. Notable widget implementations provided by TouchGFX include a Button, a TextArea and an Image.

As described in Section 2.4.2.1 adding widgets and containers to a view creates a hierarchical two-dimensional linked list. Widgets and containers are specializations of the same abstract class, Drawable, and contain basic information related to GUI objects, such as position, size or visibility.
In addition to this, each derived Widget and Container contains specific content related to its function in the GUI. While the content of a container is a list of child drawables, every widget may have its own type of content relevant to its use, e.g., a TextArea containing textual content. The ability of a GUI testing framework to determine the type of and extract content from drawables is a prerequisite in order to acquire the state of the GUI.

### 2.4.2.4 Touch events

A touch event is defined as an application level abstraction of a user intent produced by physical stimuli to a touch-enabled display. Identifying such events is an important aspect of the TouchGFX framework as each event type maps to a specific event handler. In general TouchGFX supports three types of touch events: clicks, drags and gestures. While the first two are well-known from traditional desktop application interaction, the latter encompasses generic inputs enabling the implementation of more sophisticated interaction such as the flick gesture known from smartphones and tablets.

Each event type is defined by a series of properties relevant to the user action it represents. A click event is defined by its direction, i.e., held down or released, and the coordinates of the interaction on the physical interface. This means that user interaction in TouchGFX is coordinate-based, leaving it to the test framework to provide the proper GUI object abstraction if it is to alleviate the maintainability issues of coordinate-based GUI testing, described in Section 2.2.3. In our GUI testing framework this abstraction is provided via heuristics-based GUI object recognition, described in Section 3.4.4.3.

### 2.4.2.5 Application front-end

The front-end of a TouchGFX application is responsible for the management of GUI resources, the navigation between screens and interfacing to other components including the HAL and the application itself. It consists of concrete implementations of the abstract Heap and Application classes.

- The heap handles resource allocation for the entire application. This is done by identifying the largest view, presenter and transition and allocating the memory sufficient to contain these at the same time. A concrete heap, such as the FrontendHeap shown in Figure 2.8, contains an optional Model along with the concrete application instance. As such, the heap provides a single application object, which manages all of the memory allocated for a TouchGFX application.

- The application base class defines means for lower-level components, e.g., the HAL, to forward identified touch events into the application execution. This makes it possible, through the application instance, to invoke specific touch events on the GUI enabling a test component running in the same context to control application execution. Concrete application implementations, such as the FrontendApplication shown in Figure 2.8, defines available transition-handlers to switch between application screens.

### 2.4.3 Hardware Abstraction Layer (HAL)

The purpose of the HAL is to provide abstraction from platform-specific hardware composition, including device drivers, and serve as a framework engine.
2.4.3.1 Target abstraction

By ensuring abstraction from the specific target, changes to hardware configuration and device driver implementations will not affect the application code, improving robustness against heterogeneous platforms as additional platform support only requires a new HAL implementation. Figure 2.12 shows how the HAL interface enables easy extension of platform support by introducing a new concrete HAL implementation.

When creating a concrete HAL implementation for TouchGFX, the most important aspects are the operating system (OS) and the device drivers.

- The target OS used in embedded development may vary significantly in size and features. As additional features introduce overhead and greater hardware requirements, low-resource embedded development tends to focus on the minimal OS such as FreeRTOS and TinyOS. The recommended approach to TouchGFX is to utilize the FreeRTOS distribution in order to enable low-resource, real-time execution of the GUI application. In essence the FreeRTOS distribution supplies threading facilities and resource management for a multitude of target platforms [3]. However, as there is no concept of multiple processes in FreeRTOS, but rather a scheduled and prioritized execution of multiple tasks, running parallel processes is not an option. This means that a test framework can not be completely separated from the AUT as they must be compiled together and will compete for the processor.

- Target device drivers are software interfaces enabling the use of physical peripheral devices, e.g., Universal Asynchronous Receiver/Transmitter (UART) and Analogue-to-Digital Converter (ADC) components. The TouchGFX distribution provides graphical device drivers for several platforms, but no implementations are included for other peripheral components. Utilization of such components and device drivers are thus subject to application-specific implementation, which may be placed in the back-end.

Provided with the TouchGFX distribution are a Windows and a Linux simulator. The simulators are based on SDL\(^1\), which is a cross-platform library providing access to keyboard and mouse input as well as graphical output. By using SDL in the concrete HAL implementation, a TouchGFX application may run on a computer in Windows or Linux.

2.4.3.2 Framework engine

The HAL is responsible for forwarding events to the TouchGFX core. In this respect the HAL can be said to constitute the engine that drives the TouchGFX framework. Apart from forwarding

\(^1\)http://www.libsdl.org/
touch events, the HAL forwards lower-level tick and vsync events to the application. A tick event is a simple abstraction of the target clock, while a vsync event indicates that the framework has finished processing GUI instructions and rendering the graphics. Knowing when the framework has finished rendering allows the core to postpone further instructions to the HAL until the GUI has reached a stable state. Related to the temporal synchronization problem, described in Section 2.2.3, this is an important feature in a test automation framework as it enables the framework to discern when the AUT is idle and ready for new instructions.

The vsync event is utilized by our GUI testing framework in order to synchronize with the AUT. This process is described in Section 4.4.
Chapter 3

An Automated GUI Testing Framework

This chapter describes the conceptual design of an automated GUI testing framework. The design enables automated GUI testing of embedded applications on low-resource devices. Besides the overall design of the framework, this chapter presents the detailed design of the host framework in a partitioned architecture. An extended design including the Application Under Test is presented in Chapter 4.

3.1. Overview

The framework presented throughout this chapter enables automated GUI testing of embedded applications on low-resource devices. This ability is at the core of the thesis goals presented in Chapter 1, and it is achieved using methods and technologies presented in Chapter 2 along with methods from related work presented in this chapter. The design is validated in Chapter 5. Assumptions made for the design are described in Section 3.2 and related work is described in Section 3.3. The design itself is described in Section 3.4 and is discussed in Section 3.5.

3.2. Assumptions

When using the GUI testing framework, testers without any programming experience may write test specifications, but it is assumed that test libraries, as described in Section 3.4 below, are provided by developers. Test libraries may specify application- and platform-specific configurations, such as using serial communication as the concrete method of communication between the framework and the SUT, which requires knowledge of the setup of test equipment. Test libraries rely on a structured model of the GUI of the AUT, as described in Section 3.4.4.3 below. The test specification may refer to GUI objects such as buttons and textboxes by a logical name which maps to a GUI object in the model. This abstraction of the AUT is used by scriptless GUI testing frameworks, as described in Section 2.2.2, in order to enable test-driven development. Throughout this thesis it is assumed that this model is generated by a tester or developer using a GUI survey tool. In compliance with Section 1.3.3 the survey tool is out of scope for this thesis, but it is expected that the tool will extract and interpret a structure of GUI elements from the AUT similar to how the current state of the GUI is retrieved for validation, as described in Section 3.4.5. The tool can then enable the tester to label each GUI object by a logical name used in the test specification and, upon completion, persist the model in a file or database similar to the use of test databases in scriptless frameworks.
3.3. **Related work**

This section describes and analyzes how others have approached the challenge of embedded low-resource GUI testing. Several solutions are presented and then compared and evaluated against the thesis goals.

### 3.3.1 Work by Jacobs et al.

A solution to automated testing on embedded devices is presented in [24]. The solution is characterized by a partitioning of the framework, presented in Figure 3.1. The test specification, framework engine, and the generated test report are all run on a host machine, while the AUT is running on a target device. The two environments are coupled by a test navigation layer, which provides communication and interpretation of commands and results. The solution is based on the complexity, heterogeneity, and resource constraints of embedded software development.

In order to meet the challenges of complex software, the solution makes a distinction between test specification and test implementation (or navigation). The test specification utilizes spreadsheets with actionword- and keyword-scripting, described in Section 2.2.2, and is based on the software requirements. The test implementation links the actionwords and keywords from the specification to the software interface of the AUT.

The partitioned architecture of the solution serves to meet the challenges of resource constraints and heterogeneity. Test specifications are maintained, interpreted, and executed on a host machine in order to minimize overhead on the target device. The test navigation layer exists in both environments, but the majority of the navigation is implemented on the host in order to reduce target overhead and alleviate the heterogeneity of embedded devices as the amount of device-specific code is reduced.

![Figure 3.1: Overview of the test framework presented by Jacobs et al.](image)

This solution offers several benefits including a reduced code size on the target and a clear abstraction between the test framework and the AUT. The drawback is that it introduces a new set of challenges. The remote execution of test cases requires the ability to interpret commands on the target device and inject them into the AUT. In order for the framework to assert the correct behavior the target must also be able to capture the system state and transmit this to the host. This introduces some computational overhead on the target, which in some cases might affect the execution of the AUT.
3.3.2 Work by Lin et al.

A solution to automated GUI testing of Android smartphones is presented in [30]. The solution, named SmartPhone Automated GUI testing (SPAG), is illustrated in Figure 3.2. SPAG is a record and playback framework, as described in Section 2.2.2, and in order to achieve this it integrates with the open source tools Sikuli [50] and Android Screencast [47].

Sikuli is a GUI testing tool based on Computer Vision, which is a field of study that teaches machines to recognize and understand visual context through sensors and image processing software. Sikuli is a scripting framework and supports testing of any GUI application running on the same machine, as it does not require access to the source code or binaries of the AUT but works solely through image processing of screenshots.

Android Screencast is a tool that enables remote control of an Android device connected to a host computer. It creates a representation of the device GUI on the host allowing control of the GUI using mouse and keyboard. The AUT representation on the host enables Sikuli to automate GUI tests on the Android device and validate the results through screenshots of the representation.

Besides the external tools, SPAG includes methods meant to increase the accuracy of the framework, defined as the success rate of examining an application free of defects. These methods are named Event Batch and Smart Wait.

The Event Batch method creates a batch of GUI events on the Android device, instead of executing each event immediately. The reason for this approach is that some events are connected and sensitive to timing. If the delay between their execution is too great, the test may fail. The Smart Wait method enables the framework to dynamically adjust the delay between sending two GUI events to the device. The adjustment is based on the current CPU load of the device. If the delay between events is too long the execution time of a test suite may increase significantly. If the delay is too short, events may be ignored by the device and the test may fail.

![Figure 3.2: Overview of the test framework presented by Lin et al.](image-url)

While this solution is developed for smartphone GUI testing, some of the methods used may be applied in embedded GUI testing. Sikuli only works with GUI applications running on the same machine and even if remote GUI testing was enabled, the overhead of transmitting a screenshot from the device after every change in the GUI is infeasible on a low-resource embedded device. For purely functional GUI testing in a simulator however, this is a viable approach.

In a partitioned test framework, the synchronization between a host and a target device is a common issue referred to in Section 2.2.3 as the temporal synchronization problem. The Smart Wait method utilized by SPAG is an approach to alleviate this issue. While it is still based on a simple delay of operations, it attempts to adapt to the state of the device without relying on any processing on the device.
3.3.3 Work by Wu et al.

Another solution to automated GUI testing of Android smartphones is presented in [48]. This solution, called the Keyword-Driven framework or AKDT, is illustrated in Figure 3.3. As with the work by Jacobs et al., it is an actionword- and keyword-scripting framework relying on Robotium [51] for test instrumentation and execution.

Test scripts are written in a tabular format where test cases are made up of actionwords and keywords. Actionwords may contain keywords and other actionwords, and actionwords may be specified globally in the script in order to increase reusability. Furthermore data-driven scripting, described in Section 2.2.2, is partly supported as test data can be imported from a separate file but will not result in the automatic generation of test cases based on the data available.

Besides the keywords specified in the test script, actionwords can include keywords from test libraries. Utilities and application-specific functionality can be implemented in test libraries and thus reused by different test scripts.

The benefits of AKDT are similar to the benefits of actionword- and keyword-scripting described in Section 2.2.2. Test scripts are easy to maintain due to the high degree of reusability. Test cases based on high-level actionwords can be written by test personnel without programming experience, utilizing keywords from test libraries provided by developers. This approach also supports test-driven development as high-level actionwords abstract the specifics of the application.

The drawback of AKDT and of actionword- and keyword-scripting in general is the effort spent creating an initial hierarchy of reusable keywords. It may take a significant amount of time before the first test case is implemented and running.

3.3.4 Comparison of related work

The amount of research available within the field of embedded GUI testing is very limited and the related work presented here is not specifically within this field. However, the three solutions presented here each cover some of the challenges and considerations of embedded GUI testing, described in detail in Section 2.2.3.

The solution presented by Jacobs et al. is not designed with GUI testing in mind but with automated functional software testing of embedded systems. It is however designed to meet the resource constraints and heterogeneity that we face in our thesis work. The partitioned architecture is useful for any embedded automation framework, where minimizing the code size and resource consumption on the device is a concern. It is also a very generic solution, in the sense
that it can support any platform with an external communication interface regardless of the OS or programming language used on the platform. The solution presented by Lin et al. is based on GUI testing of Android smartphones and relies on external tools for remote control and image processing of screenshots for validation of the current state. This solution is very reliant upon Android and, in the case of Sikuli, upon a representation of the AUT on the host machine. The Smart Wait method however, is a useful approach to synchronization between a host and a target in a partitioned architecture. It does not implement exact synchronization, but on the other hand it does not rely on information from the AUT. The solution by Wu et al. is also based on Android smartphones. The focus of this solution is on the front-end of the testing framework. It promotes the use of actionword- and keyword-scripting for increased maintainability, extendability and abstraction between the test script and software interfaces.

Each of these solutions include methods or technologies useful in achieving our thesis goals. The partitioned architecture presented by Jacobs et al. offers great performance efficiency due to the reduced resource consumption on the target device. The minimization of code on the target also increases compatibility and transferability. The Smart Wait method presented by Lin et al. promotes several of the quality properties as it increases the robustness against differences on the target platform without relying on information from the AUT. The actionword- and keyword-based solution by Wu et al. offers great maintainability and also functional suitability, as functionality is easily extended and test case specification can be performed by test personnel without programming experience.

3.4. Designing a GUI testing framework

This section describes the overall design of our solution to automated GUI testing on low-resource embedded devices, applying concepts presented in Chapter 2 as well as elements from related work, presented in Section 3.3. This section starts by describing the overall conceptual design, which follows a partitioned architecture, as presented in Section 3.3.1. Afterwards the design of the host-side of the partitioned architecture is described in detail, including important terms as well as the dynamics of the host design. The design of the target-side of the solution as well as the communication protocol between host and target is described in Chapter 4.

3.4.1 The overall design

The embedded GUI testing framework developed in the context of this thesis is based on a partitioned architecture dividing the framework in a host- and target side. The two sides are connected through the back-end of the framework, named TestGFX, including remote communication and interpretation of messages. Figure 3.4 gives an overview of the complete framework. The host is the moderator in the testing scenario and is responsible for storing, maintaining, and executing test specifications as well as reporting the results to the user. The front-end of the GUI testing framework is implemented by means of the Robot Framework, an actionword- and keyword-scripting framework. The Robot Framework is connected to the host-side of TestGFX through a test library. The target is the environment in which the AUT is running and this may be a physical embedded device or a simulator running on the same physical machine as the host. The AUT is based
Chapter 3. An Automated GUI Testing Framework

Figure 3.4: Overall design of the embedded GUI testing framework.

on TouchGFX, an embedded graphics framework described in Section 2.4, which serves as the software interface between the AUT and TestGFX.

The framework is designed such that designated test personnel without any programming experience may write and maintain test specifications. A test specification relies on a model of the AUT as well as a test library. The model is generated by the GUI survey tool mentioned in Section 3.2 and the test library is provided by a developer. The Robot Framework is the engine of the GUI testing framework and executes the test specification. It uses the specified libraries to search for keyword implementations, which in turn use the model as well as TestGFX to execute GUI events on the AUT and validate the state of the GUI. Based on the execution of a test specification, the Robot Framework generates a test report that is presented to the tester.

Figure 3.5 shows the operational flow of the framework. The Robot Framework including test specifications and test reports is described in Section 3.4.2. Section 3.4.3 describes test libraries and Section 3.4.4 describes TestGFX along with the AUT model. Finally, the dynamics of the host-side of the framework are described in Section 3.4.5.

Figure 3.5: Concept of operation of the GUI testing framework.
3.4.2 Robot Framework

The Robot Framework constitutes the front-end of our GUI testing framework. It is an actionword and keyword script-based framework with a flexible script-format, relying on keywords defined in the script or in libraries, enabling easy maintenance and extension of functionalities [28].

A test specification is written in a tabular format using HTML, plain text, or other supported file types, see the Robot Framework User Guide\(^1\) for details. When using plain text, a single text file constitutes a test suite and may contain sections for configurations, test cases, keywords, and more. A test specification written in plain text is presented in Figure 3.6.

```
### Settings ###
Library com.myapp.testlibrary.TestClass
Suite Setup  Setup  COM3
Suite Teardown Teardown

### Testcases ###
A TestCase using SpecificationByExample Style
  Given textbox "myTextBox" says "Hello"
  When user presses "okButton"
  Then textbox "myTextBox" says "Robot"

A TestCase using a free style
  ${text} = get text
  Should Be Equal ${text} "Hello world"

### Keywords ###
Textbox "${textbox}" says "${text}"
  assert textbox ${textbox} ${text}

User presses "${button}"
  press button ${button}
```

Figure 3.6: Example of a Robot Framework test suite written in plain text.

The settings-section may contain references to test libraries and configurations for the entire test suite such as a common setup- and teardown-procedure.

Test cases written in the testcases-section are named in the first row and test steps are specified in the subsequent rows. In a plain text script the test steps are indented by two whitespace characters. Test steps are made up of keywords, arguments, and variables, which are separated by two whitespace characters. When resolving a keyword, the framework will first look in the set of built-in keywords, then any keywords defined in the script, and lastly it will look in any referenced libraries. Keywords are subject to a very loose and forgiving naming convention although the best match has a higher priority. A keyword named “GetStatus” may be matched to both “Get status” and “get_status”, but “Get status” will take priority.

Keywords defined in the script are placed in the keywords-section. The first row of a keyword is the name, including names of expected parameters. Subsequent rows, indented by two whitespace characters, are the steps that constitute the keyword.

The Robot Framework may be given a single test script or a directory of test scripts to execute and when finished it will generate a test report in HTML format. The test report will include information regarding execution times, successful tests, failed tests including any error messages produced and more. An example of a test report is presented in Figure 5.21.

3.4.3 Test library

The Robot Framework is connected to the AUT by test libraries. Several libraries are available online\(^2\), but custom libraries may also be developed, enabling easy extension of a test framework. Custom test libraries may be developed in a number of ways. There is immediate support for \textit{Python} and \textit{Java}, but the Remote library, available online\(^3\), can act as a proxy between libraries on the same or on different machines, using the XML-RPC protocol. Libraries can thus be developed in any environment and programming language with support for this protocol.

Figure 3.8 shows a test library for the test suite presented in Figure 3.6, written in Java.

A Java test library is a simple class with keywords implemented as public methods. Some optional environment variables may be set, such as the library scope. The library scope controls the life-cycle of the library and has three possible values; \textit{“GLOBAL”}, \textit{“TEST SUITE”}, or \textit{“TEST CASE”}. A global library is created once and shared by all test suites in a session of testing. A library with test suite scope is created for each test suite and a library with test case scope is created for each test case.

\(^2\)http://robotframework.org/#test-libraries

\(^3\)http://code.google.com/p/robotframework/wiki/RemoteLibrary
TestGFX on the host-side

```java
public class TestClass {
    public static final String ROBOT_LIBRARY_VERSION = "1.0.0";
    public static final String ROBOT_LIBRARY_SCOPE = "GLOBAL";

    public void Setup(String comPort) throws Exception {
        // Open the COM port
    }

    public void Teardown() throws Exception {
        // Close the COM port
    }

    public void AssertTextbox(String textbox, String text) throws Exception {
        // Identify the textbox and validate its current text
    }

    public void PressButton(String button) throws Exception {
        // Send a button-click event to the AUT
    }

    public String GetText() throws Exception {
        return "Hello world";
    }
}
```

Figure 3.8: Example of a Robot Framework test library written in Java.

In our solution the test library connects the Robot Framework to our back-end, TestGFX, utilizing its classes, and methods to control and validate a remote AUT.

3.4.4 TestGFX on the host-side

TestGFX provides the functionality necessary to remotely test the GUI of a TouchGFX application. TestGFX is designed with extensibility and maintainability in mind, such that minimal effort is required to add support for new embedded platforms or new GUI objects. Due to the partitioned architecture of our GUI testing framework, TestGFX consists of a host- and a target-solution and this section describes TestGFX on the host. The target-side TestGFX solution will be presented in Section 4.4.2.

Figure 3.9 presents the general flow of TestGFX on the host. The test library creates a Command using information about GUI objects retrieved from the AUT Model. The commands are executed through the Core of TestGFX which uses the Transport to pass the command to the SUT. Upon receiving data through the transport, the core will use the Parser to create a model of the actual state of the AUT and this model is passed to the test library in order to validate assertions made on the state.

The design of TestGFX is based on the principles of Component-Based Development and includes the generic core component, which acts as the point of entry for the test library and defines the contract between the library and the AUT in the form of a set of interfaces. These interfaces are implemented by separate components, utilized by the test library. Figure 3.10 gives an overview of the design of TestGFX on the host.

The following sections contain detailed descriptions of the components of the host design. Cursory readers may skip to Section 3.4.5 which presents the dynamics of the host design.
Chapter 3. An Automated GUI Testing Framework

Figure 3.9: Overview of the flow of TestGFX on the host.

Figure 3.10: Design of TestGFX on the host.
3.4.4.1 Core

The primary class of the Core component is the TestClient. It provides methods for the test library to identify GUI objects in order to push commands to the AUT and validate its current state. The TestClient utilizes a GUIObjectIdentifier and a TestExecutor in order to identify GUI objects in the AUT and push commands respectively.

The TestExecutor implements the communication protocol between the host- and target-side of TestGFX. It converts instances of ICommand into binary data that is sent across an instance of ITransport. The specific instances used are instantiated by the test library.

The GUIObjectIdentifier implements heuristics-based GUI object recognition, used by scriptless GUI testing frameworks described in Section 2.2.2. It receives an instance of IModel from the test library and finds the best match to that model in the current state of the AUT.

3.4.4.2 Command

The Command component consists of implementations of the ICommand interface. A command represents an action that can be performed on the GUI of the AUT. Each command contains the information necessary for the TestExecutor to send a message to the AUT that will result in the corresponding action being performed. This approach of letting the concrete command be responsible for its own invocation information is based on the Command Pattern [17].

The Core component has no knowledge of the concrete ICommand implementations. The test library instantiates the commands and passes them to the TestClient. The information required for a command to target a specific GUI object is provided by the test library.

The test library selects the target GUI object from the AUT model generated by the GUI survey tool, mentioned in Section 3.2, and attempts to identify the corresponding GUI object in the current state utilizing the TestClient. The information passed to the command is that of the actual GUI object. Identifying the actual GUI object instead of relying solely on the information provided by the model increases the maintainability of the GUI testing framework by increasing robustness against small differences between the generated model and the actual state, such that the model does not have to be renewed after every change to the GUI.

3.4.4.3 Model

The Model component consists of implementations of the IModel interface. Each model represents a GUI object and as some GUI objects can contain other objects, the state of the AUT can be expressed through a hierarchy of concrete IModel implementations.

In our solution the models are used by the parser in order to express the state of the AUT, but also by the test library in order to activate commands on a specific GUI object, as described in Section 3.4.4.2 above. A model contains information about a GUI object including position, size, content, and child elements. Utilizing this detailed model, heuristics-based GUI object recognition is performed. The recognition procedure compares the model provided by the test library to each element of a model of the current state created by the parser. The result of the comparison is the element of the actual state that best matches the provided model.

When evaluating the compliance of a model of the actual state to a provided model, a percentage value indicating the level of compliance is calculated. The calculation is based on the comparison of properties weighted by their level of significance. The larger the weight the more the comparison of that property will affect the result of the collected comparison of the model. The model with the largest collected level of compliance is selected as the best match to the provided model.
The use of weighted properties is common with heuristics-based GUI object recognition. Properties such as position and content are more unique to a GUI object than size. Position on the other hand is a very volatile property. Weights allow the algorithm to be tailored to the specific properties available in the model.

Besides targeting commands at specific GUI objects, the test library also uses GUI object recognition when validating assertions made to state of the AUT. With the model of the actual GUI object identified, the test library may directly validate any properties it contains.

### 3.4.4.4 Parser

The Parser component consists of implementations of the `IStateParser` interface. The TestExecutor uses an `IStateParser` instantiation in order to interpret the binary state received from the AUT into an object-oriented representation based on the Model component. The specific parser to be used is instantiated by the test library and passed to the TestClient. Along with the Command component, this abstracts the communication format from the Core component.

### 3.4.4.5 Transport

The Transport component consists of implementations of the `ITransport` interface. The TestExecutor uses an `ITransport` instantiation to transmit messages to the target device. The test library is responsible for instantiating the specific transport that will be used, and the specific method of communication is thus abstracted from the Core component.

### 3.4.5 Dynamics of the host

The GUI testing framework is driven by the Robot Framework as it executes a test script. We describe the execution of a script in the framework in three phases: initialization, command execution and state validation.

#### Initialization

When executing a script, Robot Framework will start by resolving keywords. Built-in keywords are matched first followed by any referenced libraries. The last step of the initialization phase is the instantiation of any global libraries, explained in Section 3.4.3. Afterwards the framework will start executing the first test suite in the script. Figure 3.11 shows the initialization phase of our framework including a global library.

The global test library is instantiated before any test suite is prepared for execution. The library is responsible for creating the concrete `ITransport` implementation that will be used and passes it along as it creates the TestClient. Afterwards, the test library initiates a reset of the AUT. The reset is performed in case the test library is not global and the AUT has been left in an unknown state by a prior test suite execution.

#### Command execution

A command corresponds to an action that can be performed on the GUI such as a click or a drag-event with the exception of a reset- and state command sent to reset the AUT or obtain the current state of its GUI respectively. Figure 3.12 shows the process of executing a GUI event command on the host-side of the framework.

In order to execute a command on the GUI of the AUT, the test library first identifies the GUI object affected by the command in the current state of the AUT. This process is referred to as
Dynamics of the host

Figure 3.11: Dynamics of how TestGFX is initialized on the host.

Figure 3.12: Dynamics of how GUI event commands are executed through TestGFX on the host.
heuristics-based GUI object recognition, described in Section 3.4.3. The fragment “get current state” of Figure 3.12 is given by Figure 3.13.
With the information provided by the identified GUI object, the test library creates the command instance and sends it to the TestClient. The communication protocol used between the host and the target is, on the host-side of the framework, implemented by the TestExecutor, which receives the command from the TestClient. The command is then converted to a binary message and sent through the transport created during framework initialization.

State validation
Validating assertions made to the state of the AUT is a crucial feature in a test automation framework.
In order to validate the state of a GUI object, the test library identifies the object in the current state of the AUT by heuristics-based GUI object recognition. The process is the same as in command execution described above. After the test library has obtained the GUI object its state, in terms of internal properties, can be validated directly. Figure 3.13 shows the process of state validation.

![Figure 3.13: Dynamics of how the AUT state is validated through TestGFX on the host.](image)

In order to identify the GUI object to be validated, the current state of the AUT must first be obtained. A state command is created and sent to the target instructing it to send back the state of the GUI. The state is sent across multiple data packets, described in Section 4.4.5, retrieved by the TestExecutor and interpreted by its parser, described in Section 3.4.4.4, before it is returned to the TestClient.
3.5. Discussion

The overall design of the GUI testing framework is partitioned on a host and a target with the Robot Framework as the front-end on the host-side and TouchGFX as the run-time environment on the target-side. With the use of Robot Framework follows actionword- and keyword-scripting as the primary approach to test specification, and the partitioned architecture creates the need for a communication protocol and mechanisms between the host- and target solutions.

3.5.1 Comparison to related work

The partitioned architecture of our framework is based on the work by Jacobs et al. described in Section 3.3.1. Their solution is not directed specifically at GUI testing, but the challenges of constrained resources, complex software, and heterogeneous hardware platforms are the same. The TestFrame engine developed by Jacobs et al. corresponds to our use of the Robot Framework. The use of the Robot Framework does however, provide greater extensibility due to its use of test libraries for keyword implementation. As its back-end, TestFrame has a test navigation layer consisting of host- and target-side interpreters and a communication tool. The tool, named ActiveLink, abstracts the specific method of communication from the TestFrame engine and the interpreters. This abstraction is important for heterogeneous platform support and is achieved in our solution through interface- and component-based development.

The solution presented by Lin et al., described in Section 3.3.2, is directed at GUI testing on smartphones. As with a partitioned architecture, such as the one used in our framework, most of the processing is performed on a host machine. However, unlike our framework, GUI actions are not transmitted to a target platform but executed on a host-based representation of the AUT. Validation of the GUI state is also achieved through this representation of the AUT by the image processing of screenshots taken directly on the host. The tool used to create the remote representation is designed for Android smartphones and while smartphones come in various shapes and sizes, the general hardware and software architecture is a standardized platform. Their solution does not suffer from the challenges of heterogeneous environments faced by embedded test automation frameworks. Their use of Computer Vision seems promising, but without the host-based representation of the AUT our framework would spend a significant amount of resources frequently sending screenshots from the target to the host.

The Smart Wait method also proposed by Lin et al. is a simple approach to synchronization between the host and target. Its primitive nature makes it a lightweight and versatile solution, but for our work it has been disregarded in favor of a solution on the target, described in Section 4.4. Due to the use of Robot Framework our solution is based on actionword and keyword-scripting. The work by Wu et al., presented in Section 3.3.3, also promotes this approach due to the high level of reusability and maintainability it provides. While their solution is also directed at smartphone GUI testing, their use of actionword and keyword-scripting is generally applicable. As with the Robot Framework, their solution, called AKDT, uses test libraries. These libraries include application-specific functions that rely on the Android test automation framework Robotium, used to control and communicate with an Android smartphone. AKDT is thus very similar to our GUI testing framework, where Robotium is replaced by our back-end, TestGFX.

3.5.2 Comparison to thesis goals

In terms of the five quality properties presented in Section 1.3.1, our primary objective is to provide a solution to embedded GUI testing that provides the functionality expected, is resource-effective and is easy to maintain. The solution should be compatible with existing software, in the sense
that contamination of TouchGFX and the AUT should be minimized, and it should be easy to transfer to other platforms.

The overall design and host-side solution presented in this chapter utilizes partitioning in order to reduce resource consumption in the target environment. Another benefit of running most of the solution on the host is reducing the amount of code that could be platform-dependent, increasing transferability and maintainability. Maintainability is further increased by the use of interface- and component-based design as well as the Command Pattern, enabling easy extension of the framework. Furthermore, the heuristics-based GUI object recognition provides robustness against trivial changes to the GUI, compared to a bitmap-based approach relying on image processing [33].

The use of the Robot Framework provides actionword and keyword-scripting with support for data-driven scripts as well. While this approach to GUI testing may provide a steep learning curve for testers without programming experience, the hierarchies of reusable keywords make the test scripts easy to maintain. With the Robot Framework and its reusable test libraries that can contain utility- or application-specific functions, our solution is capable of providing all of the functionalities necessary of a test automation framework with the addition of easy extension.
This chapter describes the target design of the automated GUI testing framework presented in Chapter 3. The target design enables remote execution of GUI tests on multiple, heterogeneous targets; both physical and simulated. Apart from describing the core target design and integration with the TouchGFX framework, this chapter will present the protocol used for communication between the host framework presented in Chapter 3 and the target framework presented here.

4.1. Overview

The target design presented in this chapter enables automated execution of GUI tests on TouchGFX-based applications running on remote heterogeneous targets. Following the overall design, the target design completes the GUI testing framework in order to fulfill the thesis goals presented in Chapter 1.

The design presented here is influenced by certain characteristics of the TouchGFX framework, described in Section 2.4. These characteristics are presented in this chapter and used as assumptions for the enabling GUI framework of the AUT.

The assumptions made for the design are presented in Section 4.2 below. Related work relevant to the target design is presented in Section 4.3. The target design itself is presented in Section 4.4 and subsequently discussed in Section 4.5.

4.2. Assumptions

The target design presented in Section 4.4 is influenced by the constraints and the design of the GUI framework TouchGFX described in Section 2.4. This has led to the assumptions presented below. The context referred to by the assumptions relates to a runtime context where public methods of the GUI framework can be accessed.

While interfacing to the AUT it is assumed that:

- a public abstraction of the Application instance is available to any procedure running in the same context. This abstraction contains the means to invoke GUI events such as click and drag events on the AUT.
methods providing access to relevant visual components and internal attributes are available to any procedure running in the same context. These methods provide the means to extract relevant state from the application GUI.

• methods for synchronizing with the GUI of the AUT are available to any procedure running in the same context. This assumption is related to the temporal synchronization problem described in Section 2.2.3.

• methods for transitioning between views and resetting the state of the AUT are available through the application layer of the AUT to any procedure running in the same context. These methods contain the means to return the AUT to an initial state.

4.3. Related work

This section describes and analyzes how others have approached the challenges of test automation on embedded devices. Two solutions are presented both relating to test automation on embedded devices.

4.3.1 Work by Kervinen et al.

A model-based approach to GUI testing of Symbian smartphone devices is presented by Kervinen et al. [27]. The test framework is partitioned into a host and target module as shown in Figure 4.1. Also, the solution utilizes actionword and keyword-scripting, presented in Chapter 2, as the approach to test specification and execution. The solution focuses on performing parallel GUI testing on multiple mobile devices, however the practical aspect of the solution is applicable to GUI testing of a single AUT running on a single target.

Kervinen et al. present a practical approach to performing GUI tests on remote devices. Based on a mathematical specification, a Test Model is generated by a specification tool, TVT Tools, using actionwords. These actionwords are refined into low-level keywords that realize the interaction with the target. The supporting keywords govern the capabilities of the framework to perform GUI testing and are categorized into five groups: command, navigate, query, control and state verification.
Work by Loft

- **Command keywords** define simple actions to be performed on the AUT, e.g., “click the ok-button”.
- **Navigation keywords** define actions, which require navigating complex GUI elements, such as “select menu item”.
- **Query keywords** enables assertion of the presence of specific texts and images in the application window.
- **Control keywords** are used to manage state, e.g., returning the application to an initial state.
- **State verification keywords** enable the assertion of non-queryable state such as a list item having been selected.

Kervinen et al. argue that supporting the first four groups will be sufficient for most test situations while the state verification keyword is best used in specialized situations, where complex GUI elements are employed.

The low-level keywords are realized by a combination of two general purpose tools, *QuickTest Pro* (QTP) and *m-Test*. QTP is a test automation tool for Microsoft Windows applications, which enables the capture of information about GUI elements as well as applying stimulus to these. m-Test is a Windows application, which enables remote control of Symbian devices. Through target-to-host transfers of bitmaps and subsequent use of a text-recognition algorithm, m-Test can extract text strings present in the target application window. The currently active bitmaps and text identified on the target are made available to the host through the m-Test application. By aiming QTP at the m-Test application, tests are transitively carried out on the target device. Assertion of the GUI state is similarly based on asserting the presence of specific text strings in the m-Test application. As such the solutions GUI test realization is based on the host reflection of the AUT rather than the device. This introduces some latency to the execution of tests as the continuous transfer of bitmaps from target to host is resource expensive.

While the solution is directed towards remote Symbian application testing, the keyword groups described are applicable in GUI testing in general and thus embedded GUI testing as well. The employment of general purpose tools provides the necessary functionality at a low development cost but produces significant latency in test execution. Also, an application-based tool for remote invocation and state extraction must be available rendering the solution unfeasible for general embedded GUI testing on heterogeneous devices.

### 4.3.2 Work by Loft

In his thesis [31] Loft presents a solution for automated integration testing of embedded software running on remote devices. The framework described by Loft is partitioned into a host and a target module connected via an RS232 interface as shown in Figure 4.2.

The approach used by Loft employs a *Remote Procedure Call* (RPC) model to communicate actions from the host to the target device. This is implemented by a host-side API abstraction, which maps to a device-side API. The device-side API includes several *instrumentations* of relevant methods on the target. By employing this design Loft enables the specification of tests on the host using the API abstraction, while the execution is performed remotely through RPC model and device API. Subsequent validation and reporting is performed on the host.

In order to realize RPC, Loft utilizes a lightweight protocol suitable for low-bandwidth RS232 communication. Figure 4.3 shows a packet transmitted by the host in order to invoke an *action* on the device. Further packages are defined which enable request and return of system flags.
Figure 4.2: Overview of the test framework presented by Loft.

and events relevant to assertion of the state of the system and the functionality of event-based peripherals such as the RS232 itself.

Figure 4.3: Invoke Action packet used by the solution presented by Loft.

While the framework presented by Loft is designed for remote integration testing, the principles are useful for remote test automation in general. The approach requires instrumentation to be performed on top of the AUT but it avoids Built-In-Test (BIT) code. Utilizing a standardized communication interface such as RS232 ensures that the approach can be applied to a wide range of devices while the lightweight protocol ensures minimal impact on the execution of the AUT.

### 4.3.3 Comparison of related work

Research focused on realizing automated testing on embedded devices is sparse and typically aimed at either traditional integration and unit testing or GUI testing of powerful embedded devices such as smartphones. By analyzing related work however, generally applicable approaches can be identified addressing the challenges of automated GUI testing of embedded applications described in Section 2.3.1.

The solution presented by Kervinen et al. relies on the use of a general purpose tool, m-Test, for accessing and controlling Symbian devices. The approach supports GUI testing on heterogeneous targets sharing a common denominator, i.e., the Symbian OS. Invocation and extraction is targeted at the denominator rather than the application itself, thus abstracting away the target-specific implementation. The approach to validation by ascertaining the state of a reflection of the AUT is useful for moving the majority of processing onto the host. However, due to the reliance
Designing embedded test automation

upon bitmap transfers from the target to the host and the complex image processing employed by m-Test, the overhead introduced is infeasible for low-resource devices. Furthermore, the use of target-specific tools such as m-Test in the presented approach is practical but infeasible for non-standardized heterogeneous devices as it would require such a tool to be produced for every specific device type which presents significant development costs.

The solution presented by Loft focuses on low-overhead, packet-based communication and directed RPC calls to instrumented application methods. The approach benefits from the simplicity of using the RPC model but requires intrinsic knowledge of the AUT in order to introduce the needed instrumentation. By employing a common transport found on embedded devices, i.e., RS232, Lofts solution is applicable to numerous embedded devices.

Both Loft and Kervinen et al. describe a similar approach to test execution by using host-side abstractions of functionality existing on the target. Loft achieves this through RPC and Kervinen et al. through the employment of host-side keywords as well as m-Test. While they share characteristics in remote invocation, their individual approaches are very different. Kervinen uses a target-specific tool to control the device while Loft uses low-level communication mechanisms to send binary packets. These are in turn interpreted and matched to their corresponding target procedures.

Kervinens solution benefits from low development effort by using an existing tool. However it is limited in terms of efficiency by the computational overhead produced by continuous bitmap transfers. Lofts solution benefits from low computational overhead but is limited by development effort needed to apply necessary instrumentation to the software under test.

With regard to this thesis work, the solution by Loft is better suited for low-resource devices, while the solution by Kervinen et al. presents valuable concepts for GUI testing applications by means of a common denominator, i.e., the Symbian OS.

The communication protocol presented by Loft offers great performance efficiency by minimizing communication time between the host and target. The use of the RS232 interface also provides a high degree of transferability.

By targeting a common denominator shared by multiple devices, the solution by Kervinen et al. offers increased transferability and compatibility on related targets. However, instrumentation as described by Loft would be required. Using an RPC model may increase maintainability of the target framework but would also increase the overall code size.

4.4. Designing embedded test automation

This section presents the target design for the automated GUI testing framework. Representing the embedded portion of the test framework presented in Chapter 3, the target design addresses the challenges of embedded GUI test automation presented in Section 2.3.1 by applying concepts and elements from related work presented in Section 4.3 above.

An overview of the design elements covered in this section is presented in Section 4.4.1. The target design is described in detail in Section 4.4.2. The integration of the target design with the TouchGFX framework is presented in Section 4.4.3 and the dynamics of the target design in relation to TouchGFX are presented in Section 4.4.4. Finally, the design of the communication protocol used between the host and target is presented in Section 4.4.5.
Chapter 4. Test Automation on Low-Resource Embedded Systems

4.4.1 The target design

In adherence with the overall design described in Section 3.4.1, the target design enables remote GUI test execution on heterogeneous target devices. This is achieved by means of listening for and interpreting host commands for subsequent execution on the device. In addition to providing a communication protocol for intercommunication, the target design presents an architecture which supports easy extension as well as integration with the target GUI framework, TouchGFX. Figure 4.4 shows the elements covered in the following sections.

![Figure 4.4: Overview of the GUI testing framework on the target.](image)

4.4.2 TestGFX on the target-side

The target-side of TestGFX is responsible for receiving, interpreting and executing test commands from the host on the target AUT. In order to support high maintainability and extensibility principles of component-based development are used. The target design shown in Figure 4.5 is governed by a generic core component which defines the procedural logic of the target TestGFX framework. The core serves as the main point of entry to TestGFX on the target. The following sections contain detailed descriptions of the components of the target design. Curiosity readers may skip to Section 4.4.4 which presents the dynamics of the target design in relation to TouchGFX.

4.4.2.1 Core

The primary class in the Core component is the TestAgent. The TestAgent provides a single method, “testProcedure”, for repeated execution in an appropriate context. This method is designed to be invoked in the main event cycle of a GUI framework. An event cycle is defined as an iteration of input sampling and processing in the GUI framework. The event cycle of TouchGFX is described in Section 4.4.3. With each event cycle, the TestAgent will listen for and interpret command messages from the host and add them to an internal FIFO queue for later execution. Once per event cycle a single command will be taken from the queue and executed by the TestAgent. The dynamics of this behavior is presented in further detail in Section 4.4.4. In order to receive commands from the host the TestAgent utilizes a ProtocolClient. The ProtocolClient utilizes a Transport instance for low-level communication with the host and a ProtocolInterpreter instance for interpretation of binary messages into Command objects. The
specific Transport instance used is instantiated by the context of the TestAgent as it is platform-specific. The ProtocolClient populates the internal list of commands in the TestAgent, which are continuously executed with each invocation of the “testProcedure”. This process is described in detail in Section 4.4.4.

The TestAgent uses a StateClient in order to retrieve the state of the AUT. The StateClient populates an internal byte array in the TestAgent with state information from the target. The implementation of the state acquisition is defined by the State component, described in Section 4.4.2.4. The TestAgent also relies on an Application Abstraction Layer (AAL) interface in order to reset the AUT. The implementation of the AAL is application-specific and defines a contract between the AUT and the TestAgent. For a TouchGFX application to be TestGFX-enabled it must implement a concrete instance of the AAL, which defines a “Reset”-method for the application. Resetting the AUT is a very important feature for a test automation framework. The AAL interface is described further in Section 4.4.3.4.

### 4.4.2.2 Command

The command component consists of implementations of the abstract `Command` class provided by the TestGFX core. A concrete Command constitutes an instruction parsed from a protocol packet received from the host. Commands are used to perform test related actions on the target. The Command implementations are divided into two conceptual categories: *event commands* and *control commands*. Event commands, e.g., click and drag, represent GUI events to be activated...
on the AUT and control commands, e.g., state and reset, represent high-level test-related actions to be carried out by the TestAgent.

The command abstraction used here is based on the Command Pattern [17]. Every command contains information about its own invocation and is managed based on its Type attribute. The command type refers to the conceptual category mentioned earlier. By discerning the type of the command the TestAgent is able to determine how to execute it. In the case of an event type command a touch event is sent to the application, while a state- or reset type command will prompt the TestAgent to acquire and return the state of the GUI or reset the AUT through the AAL respectively. These processes are described in detail in Section 4.4.4.

4.4.2.3 Protocol

The protocol component consists of implementations of the abstract ProtocolInterpreter class. These interpreters implement the parsing of packets from the host, described in Section 4.4.5.1. The concrete ProtocolInterpreters are directly related to the concrete Command implementations, with the exception of control commands as each command is created by a dedicated interpreter. The protocol component implementation is based on the Interpreter Pattern [17]. This enables easy extension of parsing capabilities as the addition of a new packet type from the host only requires the implementation of a new ProtocolInterpreter and possibly a new Command.

4.4.2.4 State

The state component consists of GUI framework-specific implementations of the abstract State-Client. For the TouchGFX GUI framework a TouchGFXStateClient is provided. The extraction and processing of the GUI state on the target depends on the design of the GUI framework. The implementation of the TouchGFXStateClient, shown in Figure 4.5, is based on the Interpreter Pattern [17] and is designed to enable interpretation of the hierarchy of Drawable implementations found in TouchGFX core described in Section 4.4.3.

4.4.2.5 Transport

The transport component consists of implementations of the abstract Transport class. The Transport implementations are related to available hardware components and drivers for external communication on the target platform. The UART implementation, illustrated by Figure 4.5, implements serial communication via an RS232 interface. The abstraction of low-level communication mechanisms from the core increases transferability by enabling easy extension for heterogeneous targets.

4.4.3 TouchGFX integration

The TouchGFX framework, described in Section 2.4, provides support for specifying presentation logic and realizing graphical output on heterogeneous low-resource devices. As is the case in the solution presented by Kervinen et al., the target design revolves around the common denominator, the enabling TouchGFX framework, instead of the specific AUT. Since no general purpose tools for remote control of TouchGFX applications exist, instrumentation must be added to exercise relevant control on and extract state from the AUT. As mentioned in the work by Loft, described in Section 4.3.2, this requires intrinsic knowledge of the design of the GUI framework. The following sections present the integral parts of TouchGFX related to the solution presented in this thesis.
4.4.3.1 Execution context

Figure 4.6 shows a simplified overview of the TouchGFX application structure, described in Section 2.4. It is divided into four components or layers: application layer, TouchGFX core, the Hardware Abstraction Layer (HAL) and back-end.

![Figure 4.6: Overview of the TouchGFX framework.](image)

Referred to as the framework engine in Section 2.4.3, the HAL produces all the input to the application layer by means of the TouchGFX core. The HAL has a single method, “handleEvents”, which samples events and provides synchronization. A method, “backPorchExited”, performs hardware input sampling and forwards appropriate events to the application layer. Another method, “waitForVSync”, enables the HAL to postpone sampling until the hardware has completed its previous instructions and reached a stable state. A state is said to be stable when framework execution and hardware output has been realized and the application is ready to accept new input. The “waitForVSync”-method is a very important feature of the TouchGFX framework in relation to ensuring proper test execution. It fulfills the assumption for synchronization with the GUI of the AUT presented in Section 4.2.

4.4.3.2 Event invocation

Event invocation concerns low-level manipulation of the AUT by pushing touch events from the HAL and up to the application layer. Utilizing the touch event abstractions for user input, provided by the TouchGFX core and described in Section 2.4.2.4, it is possible to push automated touch events to an application instance, where it will result in the corresponding GUI event being executed on the AUT. The application instance can be accessed statically through the Application class. This is a very important feature of the TouchGFX framework and fulfills the assumption for a public abstraction of the application instance mentioned in Section 4.2. The process of event invocation is described further in Section 4.4.4.

4.4.3.3 State acquisition

When defining the presentation logic, the application layer relies on visual components derived from the Drawable abstraction provided by the TouchGFX core. A Drawable implementation has the immediate subtype of either a Container or a Widget and contains its own active state. Accessing and interpreting the internal attributes of the currently active drawables thus becomes paramount for acquiring the current state of the GUI realized by accessing the drawables through the state component described in Section 4.4.2.4. The process of state acquisition is described further in Section 4.4.4.
4.4.3.4 Application control

Application control concerns high-level manipulation of the AUT. This includes automating the application directly through active method invocation rather than transitively through touch events, as described in Section 4.4.3.2. An important example of application control is the reset of the application to its initial state. In order to reset the AUT, two actions are required: transitioning to the initial view and resetting the system-wide application model described in Section 2.4. Since the TouchGFX framework provides no means of performing these actions, the implementation of a concrete AAL in the application layer is required in order to make the AUT testable.

4.4.3.5 Integration details

The following contains detailed descriptions of the integration of the target design with the TouchGFX framework, presented in Section 2.4.

In accordance with the target design presented in Section 4.4.2, interaction with the AUT is achieved by means of the TouchGFX framework. This section presents the integration of TestGFX with the HAL, the TouchGFX core and the application layer. Figure 4.7 shows the overall associations between the TestGFX and TouchGFX frameworks.

![Figure 4.7: TestGFX integration with the TouchGFX framework.](image)

**Hardware Abstraction Layer**

In order to integrate into the execution context of TouchGFX, described in Section 4.4.3.1, the TestAgent must be added to the HAL. By means of a specialization of the concrete HAL, HAL-TARGET, test-specific code can be injected into the TouchGFX framework without compromising the ability of the HAL to realize graphical output and register hardware input. By introducing the “testProcedure”-method of the TestAgent described in Section 4.4.2.1 to the “handleEvents”-
Dynamics of the target method of the concrete HAL, the TestGFX framework is able to execute alongside the AUT. The details of the TestAgent execution are described in Section 4.4.4.

**TouchGFX core**
In order to achieve event invocation and state acquisition, described in sections 4.4.3.2 and 4.4.3.3 respectively, the command and state components integrate with the TouchGFX core. Event invocation on the AUT is realized by the command component through the use of the Application class and Event hierarchy. In a similar way, state acquisition is realized by the state component through the utilization of the Application class and the Drawable hierarchy. The interaction between command and state components and the TouchGFX core is described in further detail in Section 4.4.4.

**Application layer**
In order to achieve the application control, described in Section 4.4.3.4 the TestGFX core requires a concrete implementation of the AAL interface. As described in Section 4.4.2.1, the AAL defines a contract between the TestGFX framework and the AUT in order to enable test automation of a TouchGFX application. The FrontendAAL provides such an implementation by means of the FrontendHeap found in the application layer. The FrontendHeap, described in Section 2.4.2.5, contains a concrete FrontendApplication and an optional Model object. The realization of the “Reset”-method utilizes direct manipulation of these members in order to return the application to its initial state.

### 4.4.4 Dynamics of the target

The target framework interacts with the TouchGFX framework in order to realize test automation on the target AUT. The major interaction taking place are described through four dynamics: **initialization**, **execution**, **event invocation** and **state acquisition**. As described in Section 4.4.3.4, resetting the AUT involves application-specific implementation in the application layer. An implementation of the application reset is presented in relation to the prototype described in Section 5.3.1.

**Initialization**
The initialization of TestGFX is performed alongside the AUT. A main function initializes the target- and application-specific dependencies of the TestAgent, i.e., the AAL, StateClient, and Transport implementations. Figure 4.8 illustrates the initialization process of the TestGFX framework. Note that main is also responsible for initializing the AUT which is left out for the sake of simplicity. The “execution” fragment is given in Figure 4.9.

**Execution**
After initialization the non-terminating “handleEvent” is invoked. The “testProcedure”-method provided by the TestAgent is invoked after “waitForVSync”, described in Section 4.4.3.1, in order to ensure that GUI events are introduced after the AUT has achieved a stable state. Figure 4.9 illustrates the execution of TestGFX.

The “testProcedure”-method first listens for messages from the host. Upon receiving of a command message, it is interpreted and translated into one or more target Command objects. The number of target Command objects corresponding to a host command differs with the command type. While control commands correspond to only a single target command, certain event commands are interpreted as multiple target commands to be executed sequentially. This behavior is described in detail in Section 5.3.1.2.
Chapter 4. Test Automation on Low-Resource Embedded Systems

Figure 4.8: Dynamics of the initialization of TestGFX on the target.

Event invocation
Event invocation occurs when the TestAgent registers an event command taken from the internal command queue. As described in Section 4.4.2.2, event commands contain invocation details internally in adherence with the Command Pattern. The TestAgent invokes the “execute”-method of the command, which realizes the event invocation on the AUT. Figure 4.10 illustrates how an event command is registered and executed through its “execute”-method.

The realization of event invocation includes the creation of an Event implementation, e.g., ClickEvent. The Event object is sent to the appropriate handle-method contained in the Application instance, e.g., “handleClickEvent”. This behavior is described in further detail in Section 5.3.1.2.

State acquisition
State acquisition occurs when the TestAgent registers a state command taken from the command queue. As described in Section 4.4.2.4, the acquisition of GUI state is subject to the GUI framework-specific implementation of the StateClient interface, i.e., TouchGFXStateClient.

Figure 4.11 illustrates how a state command is registered and how the TestAgent proceeds to obtain the state by means of a concrete StateClient, i.e., the TouchGFXStateClient.

To acquire the state of the GUI from a TouchGFX application the TouchGFXStateClient employs a DrawableInterpreter capable of traversing the base containers child elements. The base container accessed through the Application instance is the root element of the linked list of visual elements in the current view described in Section 2.4.2.1. The traversal of this linked list is described in further detail in Section 5.3.1.2.
Dynamics of the target

Figure 4.9: Dynamics of the execution of TestGFX on the target.

Figure 4.10: Dynamics of the processing of an event command.
Chapter 4. Test Automation on Low-Resource Embedded Systems

Figure 4.11: Dynamics of the processing of a state command.

4.4.5 TestGFX communication

This section presents the communication layer of TestGFX. Figure 4.12 shows the design of the communication layer which is based on the work by Loft described in Section 4.3.2. TestGFX relies on the implementation of a protocol which in turn relies on an implementation of a transport providing abstraction from the low-level communication mechanism used. The protocol defines the messages that are sent through the transport and is described in Section 4.4.5.1 below.

4.4.5.1 Communication protocol

The protocol employed in the communication between the host and target framework partitions is designed to be lightweight and extensible. It is based on a hexadecimal identifier scheme similar to that employed in the protocol presented by Loft. The packets are fixed in size with a 2 byte header and a 22 byte payload. Figure 4.13 shows the basic structure of the packets.

The Type-byte identifies the general packet type, i.e., state, reset or event, related to the command types described in Section 4.4.2.2. Depending on the type, an Id-byte is used to indicate the sub-type or id of the packet. This is relevant for the event type in order to sub-type the specific GUI event to perform, e.g., click or drag. Also, when sending the state of the GUI from the target to the host the amount of data is too large for a 24 byte packet. The state is thus split into several packets and the sub-type indicates the first and last packets of a stream. With the Type and Id identifiers
the protocol is thus easily extended.
In comparison to the work presented by Loft, the use of fixed-size packets introduces some additional overhead to messages that do not utilize the full payload width. Fixed packets are easier to handle for the recipient however, and simplifies the implementation of communication.

### 4.5. Discussion

The target design of the GUI testing framework is based on the constraints presented by the TouchGFX framework as well as related work. Although generic in the core of the architecture, some measure of application- and platform-specific implementation is inevitable in embedded software development and a large part of the target design is based on the assumption of TouchGFX as the GUI framework and runtime environment of the AUT.

In this section the target design is discussed and compared to the related work, presented in Section 4.3, as well as the thesis goals, presented in Chapter 1.

#### 4.5.1 Comparison to related work

The solution by Kervinen et al., presented in Section 4.3.1, achieves abstraction of the target device by utilizing the m-Test tool enabling consistent execution on heterogeneous platforms running a
Symbian OS. This abstraction by means of a common denominator is achieved in our solution by integrating with the GUI framework TouchGFX rather than the AUT. The work by Loft, presented in Section 4.3.2, provides a solution to remote integration testing of embedded applications. The communication protocol used in our solution is largely based on the protocol presented by Loft. The use of a 2 byte header for communication packets supports extensibility of the protocol by the addition of new message types but also by the addition of messages that need to be split into several packets.

4.5.2 Comparison to thesis goals

In terms of the five quality properties presented in Chapter 1, the primary properties of the target design are performance efficiency, compatibility, maintainability and transferability. External communication is resource intensive and in order to increase performance efficiency the target design employs a lightweight packet-based protocol. Transmitting the state of the AUT from the target to the host requires several packets to be sent. The number of packets required however, is reduced by minimizing the size of the state message by using hexadecimal values to represent GUI elements and attributes. This increases the amount of processing required on the host-side of the framework but reduces communication overhead on the target. Furthermore, the packet structure in combination with the Command and Interpreter patterns utilized provides an extensible communication layer increasing the maintainability of TestGFX on the target. On the target-side, TestGFX is structured around the architecture of the TouchGFX framework. By relying on the HAL for GUI event invocation and the TouchGFX core for state acquisition, TestGFX is isolated from the AUT increasing the compatibility of the GUI testing framework. Working under the assumption of TouchGFX as the GUI framework, stated in Section 4.2, the isolation from the AUT also increases transferability as TouchGFX provides a common denominator for heterogeneous platforms.
Validation of an Automated GUI Testing Framework

This chapter evaluates the conceptual design of the GUI testing framework presented in this thesis. The design is presented in chapters 3 and 4. Chapter 1 defines the thesis goals from which a set of quality properties have been selected. These properties form the basis of the validation of a prototype of the GUI testing framework also presented in this chapter.

5.1. Overview

In order to validate the conceptual design of the GUI testing framework presented in chapters 3 and 4 of this thesis, an evaluation process based on the ISO/IEC 25000 set of standards is used. As a part of this process the design is evaluated on five quality properties chosen from the ISO/IEC 25010 product quality model [8], presented in Chapter 1. The evaluation of a property is based on a set of quality requirements each consisting of a set of measurements. The requirements and measurements are based on the ISO/IEC 25030 [7] and ISO/IEC 25020 [6] standards respectively. In order to evaluate the measurements, a proof-of-concept prototype is developed. Based on this prototype, a test suite is implemented and executed on an AUT, both designed for this purpose. The results of the measurements are added to a score for each quality property and these scores, presented in a stacked bar chart, constitute the results of the evaluation.

Section 5.2 describes the evaluation process utilized in order to validate the GUI testing framework. Section 5.3 describes the prototype framework, the test environment used in order to evaluate the measurements and presents the results of the evaluation. Finally, Section 5.4 discusses the results of the evaluation process and the validation of our thesis work.

5.2. Evaluation process

The evaluation process described in this section is based on the ISO/IEC 25000 set of standards. Specifically the ISO/IEC 25010, 25020 and 25030 standards are used. Figure 5.1 shows the use of these standards in the evaluation process.

The quality properties based on ISO/IEC 25010 are described in Section 5.2.1 and the quality requirements and measurements based on ISO/IEC 25030 and 25020 respectively are described in Section 5.2.2.
Chapter 5. Validation of an Automated GUI Testing Framework

Figure 5.1: Software product evaluation based on the ISO/IEC 25000 standards.

5.2.1 Quality properties

The ISO/IEC 25010 standard [8] defines a software product quality model consisting of eight quality properties. As described in Section 1.3.1.1, five of the eight quality properties are chosen for this evaluation including functional suitability, performance efficiency, compatibility, maintainability and transferability. Figure 5.2 gives an overview of the properties and sub-properties included in the quality model used for this evaluation.

For each of the five quality properties a set of requirements are specified based on the ISO/IEC 25030 standard [7]. Each requirement is validated by a set of measurements based on the ISO/IEC 25020 standard [6]. The quality requirements and measurements are described in Section 5.2.2 below. Results of the evaluation are presented in Section 5.3.3 and discussed in Section 5.4.

5.2.2 Quality requirements and measurements

The quality requirements and measurements presented in this section are used in the evaluation of the GUI testing framework on the five quality properties presented in Section 5.2.1 above. Each of the quality properties contains a set of requirements and each requirement contains a set of measurements. The quality requirements and measurements are based on the ISO/IEC 25030 and 25020 standards respectively.

A measurement is a simple statement that is easily asserted to be true or false whereas requirements may be more intangible. The measurements included in this evaluation are asserted by three different approaches however.
Measurements for functional quality requirements, such as the ones specified for functional suitability in Section 5.2.2.1 below, are asserted through execution of the test suite described in Section 5.3.2.3. The tests are executed on the AUT described in Section 5.3.2.2 by means of the prototype described in Section 5.3.1. The AUT is run on a physical device, LPC4357 Developer's Kit\textsuperscript{1}, from here on referred to as LPC4357DK. The LPC4357DK is a representation of a low-resource device fitting the thesis profile. Further details about the LPC4357DK and the prototype can be found in Appendix B.

Measurements for non-functional quality requirements, such as the ones specified for maintainability in Section 5.2.2.4 below, are asserted by static analysis. For some measurements, such as the ones concerning code size, development tools are utilized. For other measurements the results are based on a code review. Appendix C provides further details on the validation of measurements and results.

### 5.2.2.1 Functional suitability

As described in Section 1.3.1.1, this quality property represents the degree to which the GUI testing framework provides a set of functions appropriate for the expected tasks and user objectives and provides the specified results with the needed degree of precision and accuracy.

1. **The framework shall support features expected of an automated GUI testing framework.**

This requirement relates to the sub-property *appropriateness*. The measurements specified in Table 5.1 represent functions that the GUI testing framework should provide. The validation of these measurements is based on the test environment described in Section 5.3.2. The test suite is executed on the AUT running on the LPC4357DK device.

<table>
<thead>
<tr>
<th>Requirement 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1 The framework supports specification of user scenarios on GUI applications.</td>
</tr>
<tr>
<td>1.1.2 The framework supports assertion of the state of a GUI application.</td>
</tr>
<tr>
<td>1.1.3 The framework provides appropriate reporting of the execution of an automated test.</td>
</tr>
<tr>
<td>1.1.4 The framework is able to automate user actions on GUI applications.</td>
</tr>
<tr>
<td>1.1.5 The framework is able to retrieve the state of a GUI application for validation.</td>
</tr>
<tr>
<td>1.1.6 The framework is able to reset a GUI application to the initial state of a testing session.</td>
</tr>
</tbody>
</table>

1. **The framework shall be able to detect and report defects in a GUI application.**

This requirement relates to the sub-property *accuracy*. The AUT, described in Section 5.3.2.2, contains two known defects and the measurement in Table 5.2 specifies the number of defects the framework should report, known or unknown. The validation of this measurement is based on the test environment described in Section 5.3.2. The test suite is executed on the AUT running on the LPC4357DK device.

<table>
<thead>
<tr>
<th>Requirement 1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.1 The framework is able to detect and report the existence of at least two defects.</td>
</tr>
</tbody>
</table>

\textsuperscript{1}http://www.embeddedartists.com/products/kits/lpc4357_kit.php
1.3 The framework shall support easy maintenance of test specifications.

This requirement relates to both sub-properties appropriateness and accuracy. The measurements specified in Table 5.3 are validated by static analysis and by execution of the test environment, described in Section 5.3.2, on the AUT running on the LPC4357DK device.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.1</td>
<td>The framework supports reuse of test scenarios in terms of sequences of user actions and other commands.</td>
</tr>
<tr>
<td>1.3.2</td>
<td>The reference to GUI objects in the test specification is abstracted from the actual state of the AUT.</td>
</tr>
<tr>
<td>1.3.3</td>
<td>The framework is able to recognize GUI objects after insignificant changes to the GUI, i.e., changes to position or size of the GUI object.</td>
</tr>
</tbody>
</table>

5.2.2.2 Performance efficiency

As described in Section 1.3.1.1, this property represents the degree to which the test framework meets requirements for performance relative to the amount of resources used under specified conditions.

2.1 The framework shall have low resource utilization.

This requirement relates to the sub-property resource utilization. The measurements specified in Table 5.4 are validated by a benchmark of the resources consumed by the GUI testing framework when included in the target environment with the AUT and the TouchGFX framework. The resources included in the benchmark are; code size, runtime memory consumption and the maximum load placed on the processor.

The threshold values for code size and memory utilization specified for the measurements in Table 5.4 are based on the lower bounds for internal ROM and -RAM of the TouchGFX framework [2].

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1</td>
<td>The total code size of the framework is less than 20 KB.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>The runtime memory utilization of the framework is less than 10 KB.</td>
</tr>
<tr>
<td>2.1.3</td>
<td>The maximum load on the processor during automated testing by the framework does not exceed the maximum load occurring during manual testing by more than 5% of the total load possible.</td>
</tr>
</tbody>
</table>

2.2 The framework shall be able to perform GUI automation within reasonable time.

This requirement relates to the sub-property time behavior. The validation of the measurements specified in Table 5.5 is based on the test environment described in Section 5.3.2. The test suite is executed on the AUT running on the LPC4357DK device.

The threshold values specified for the measurements in Table 5.5 are based on the time behavior of other GUI testing frameworks for desktop applications [36].
Table 5.5: Measurements taken for the validation of Requirement 2.2

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1</td>
<td>The framework is able to automate a user action on the GUI of the AUT within 2 seconds.</td>
</tr>
<tr>
<td>2.2.2</td>
<td>The framework is able to retrieve the state of the GUI of the AUT within 2 seconds.</td>
</tr>
</tbody>
</table>

**5.2.2.3 Compatibility**

As described in Section 1.3.1.1, this quality property represents the degree to which the GUI testing framework can interoperate with other products, i.e., TouchGFX and the AUT, and perform its required functions while sharing the target platform with the AUT.

**3.1 The framework shall execute GUI events reliably and in deterministic order.**

This requirement relates to the sub-property *interoperability*. The measurements specified in Table 5.6 are validated by execution of the test environment, described in Section 5.3.2, on the AUT running on the LPC4357DK device.

Table 5.6: Measurements taken for the validation of Requirement 3.1

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1</td>
<td>Multiple executions of commands in quick succession are performed by the AUT in the order they are sent.</td>
</tr>
</tbody>
</table>

**3.2 While deployed in the target environment, the framework shall not affect the execution of the AUT.**

This requirement relates to the sub-property *co-existence*. The measurements specified in Table 5.7 are validated by comparison of a manual and an automated execution of the test environment described in Section 5.3.2. Manual as well as automated test execution is performed on the LPC4357DK device.

Table 5.7: Measurements taken for the validation of Requirement 3.2

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>The automated execution of the test suite by the framework provides similar results as the manual execution of the same test suite.</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The automated execution of the test suite by the framework is visually similar as the manual execution.</td>
</tr>
</tbody>
</table>

**5.2.2.4 Maintainability**

As described in Section 1.3.1.1, this quality property represents the degree to which the GUI testing framework can be effectively and efficiently modified by the intended personnel.

**4.1 The classes of the framework shall adhere to the principles of SOLID design [32].**

The measurements specified in Table 5.8 are validated by static analysis of the GUI testing framework.
Table 5.8: Measurements taken for the validation of Requirement 4.1

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1</td>
<td>Classes of the framework have only a single reason to change, or source of change, in accordance with the Single Responsibility Principle.</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Classes of the framework can be extended without alteration to the existing source code, in accordance with the Open/Closed Principle.</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Classes of the framework may substitute their super-types without incurring changes to the overall behavior of the framework, in accordance with the Liskov Substitution Principle.</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Internal as well as external parties relying on classes of the framework do not depend on methods they do not use, in accordance with the Interface Segregation Principle.</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Internal as well as external parties relying on functionality of the framework depend on abstractions rather than implementations of that functionality, in accordance with the Dependency Inversion Principle.</td>
</tr>
</tbody>
</table>

**5.2.2.5 Transferability**

As described in Section 1.3.1.1, this quality property represents the degree to which the GUI testing framework can be transferred from one environment to another. The requirement specified below relate to the target design as robustness against the heterogeneity of embedded devices is crucial for embedded test automation.

**5.1 The framework shall be easy to port from one target environment to the next.**

The measurements specified in Table 5.9 are validated by static analysis of the host- and target framework.

Table 5.9: Measurements taken for the validation of Requirement 5.1

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>No changes are required to the host framework in order to support a new target environment.</td>
</tr>
<tr>
<td>5.1.2</td>
<td>The changes to the target framework necessary in order to support a new target environment are limited to low-level device drivers.</td>
</tr>
<tr>
<td>5.1.3</td>
<td>The target framework is applicable to a new AUT without any changes to the framework required.</td>
</tr>
</tbody>
</table>

**5.3. Validation of GUI testing on low-resource embedded systems**

**5.3.1 Proof-of-concept prototype**

This section describes the prototype framework implemented in order to validate our conceptual design of a GUI testing framework for low-resource embedded systems. As described in Chapter 3, the design of the framework is based on a partitioned architecture consisting of a host environment running the framework engine and a target environment running the AUT. The host and target are connected by a communication protocol described in Chapter 4. Figure 5.3 shows an overview of the design of the GUI testing framework with our contributions highlighted.
The scope of this section is limited to the technologies that have been used and the functionalities that have been implemented for the purpose of proof of concept. The development of the prototype is described in further detail in Appendix B.

### 5.3.1.1 Host implementation

In accordance with the design presented in Chapter 3, the host consists of a front-end realized by the Robot Framework, one or more test libraries, and a back-end, TestGFX.

**Robot Framework**

Robot Framework is the engine of the GUI testing framework and it is responsible for executing test scripts as well as presenting a test report upon completion.

A single test script file constitutes a test suite and several file formats are supported. The format chosen for the proof of concept is plain text. Figure 5.4 shows an example of such a test suite. For the prototype we use RobotFramework-EclipseIDE\(^2\), a plugin for the Integrated Development Environment (IDE) Eclipse that provides a text editor for plain text test scripts with syntax highlighting, code completion etc.

Each test suite created for this prototype expects the test libraries to provide *setup*- and *teardown* methods for the test suite and test cases. The purpose of the test suite setup and teardown is to open and close the connection between the host and the target. The reason for invoking the initialization of communication from the test script is that while the method of communication is defined by the test libraries, the port or address to the target is provided by the test script as several scripts may use the same library but for different targets. The purpose of the test case setup and teardown is to ensure that the AUT is reset between each test case by sending a reset command described in sections 3.4.4 and 4.4.2. An important aspect of automated testing is the independence of test cases such that the order of execution is arbitrary.

Robot Framework is based on actionword- and keyword-scripting and while it is very flexible in its use, the recommended approach, and the approach utilized for this prototype, is to define built-in reusable high-level actionwords from other built-in actionwords and low-level keywords. An example of this is given by the *Specification By Example* styled test case in Figure 5.4. The low-level keywords are implemented by the test libraries and the Robot Framework is thus completely

---

\(^2\)https://github.com/NitorCreations/RobotFramework-EclipseIDE
Chapter 5. Validation of an Automated GUI Testing Framework

```robot
*** Settings ***
Library com.myapp.testlibrary.TestClass
Suite Setup Setup COM3
Suite Teardown Teardown

*** Testcases ***
A TestCase using SpecificationByExample Style
   Given textbox "myTextBox" says "Hello"
   When user presses "okButton"
   Then textbox "myTextBox" says "Robot"

A TestCase using a free style
   ${text} = get text
   Should Be Equal ${text} "Hello world"

*** Keywords ***
Textbox "${textbox}" says "${text}"
   assert textbox ${textbox} ${text}

User presses "${button}"
   press button ${button}
```

Figure 5.4: Example of a Robot Framework test suite written in plain text.

Test libraries

Test libraries for Robot Framework may be developed in a number of ways. For this prototype we have chosen to use Java classes packaged in JAR files as both Robot Framework and Eclipse have native support for Java. Java is also a mature language with a wide range of libraries for various technologies and it has an easy learning curve for TouchGFX developers who are used to C++.

For the proof of concept, a single test library is implemented. In its test suite setup and teardown methods it opens and closes a serial communication channel to the target through an ITransport implementation. In its test case teardown method it sends a reset command to the target. Besides setup and teardown, the test library has methods for validating the content of a TextArea, pressing a button, dragging a GUI object, and validating the position of a GUI object. Pressing a button and dragging a GUI object utilize click and drag implementations of ICommand described in Section 3.4.4.2 respectively.

In order to send commands and validate the state of GUI objects a model structure, described in Section 3.4.4.3, is required. The model structure is assumed to be generated by a GUI survey tool, but for this prototype it is implemented by the test library. Figure 5.5 shows the model structure used in the prototype.

The model structure is implemented as a map dictionary with a logical name of the GUI object as the key and the IModel instance as the value. The keys are the GUI object names used in the test script.

TestGFX

TestGFX, described in Section 3.4.4, is the back-end of the GUI testing framework utilized by the test libraries to send GUI events to the AUT and validate its current state. It consists of several components implemented as Java JAR files. As shown in Figure 3.10, the components include the Core, Command, Model, Parser and Transport.

The Core component, described in Section 3.4.4.1, includes the classes TestExecutor and GUIObjectIdentifier. The TestExecutor is responsible for implementing the communication protocol on
The host. As described in Section 5.3.1.3 below, the communication is packet-based, and the packets from the host are created by the TestExecutor using the information contained in the commands. The packets are sent via an ITransport implementation in the Transport component. This prototype implements a serial communication transport based on the jSSC library\(^3\).

The state data received from the target is compressed using hex-values as placeholders for type- and attribute names as described in Section 5.3.1.3. Before the state can be deserialized by the parser the TestExecutor has to convert the hex-values back to the original identifiers. This conversion is based on the two map dictionaries shown in Figure 5.6.

The GUIObjectIdentifier implements the heuristics-based GUI object recognition described in Section 3.4.4.3. The process first filters a model of the current state of the GUI for objects of the same type as the model to identify. Afterwards, each of the objects of the same type is evaluated against the model. The evaluation compares the position, size, content, and children and returns a value in the interval \([0.0;1.0]\), where 1.0 corresponds to a 100% match. The result of the evaluation of a GUI object is the average of the property comparisons, and the result of the heuristics-based GUI object recognition is the GUI object with the largest evaluation result.

The Command component, described in Section 3.4.4.2, includes a click-, drag- and reset command created and used by the test library. A command is sent to the TestExecutor where it is converted to a binary packet. The packets, described in Section 5.3.1.3 below, include a header specifying the type of the message and a body specifying any arguments necessary. The click and drag commands map directly to click and drag touch events on the AUT and they require a position on the GUI at which to initiate the event. Furthermore, the drag command requires a position at which to end the drag event. The position of a GUI object is provided by the test library by means of the model structure as well as GUI object recognition. Other arguments, such as the length of a drag event, are provided by the test script.

The models used to represent GUI objects are implemented by the Model component and for this prototype they include the models shown in Figure 5.7. The hierarchy of models is based on the

\(^3\)http://code.google.com/p/java-simple-serial-connector/
As described in Section 5.3.1.3, the state of the GUI is transmitted as JSON. The Parser component contains state parsers responsible for deserializing the state and the prototype implements a JSON parser based on the Google Gson library\(^4\). In order to deserialize the JSON through this library, a model structure matching the structure of the JSON elements has to be provided.

\(^4\)http://code.google.com/p/google-gson/
5.3.1.2 Target implementation

In accordance with the design presented in Chapter 4, the target prototype enables remote test execution on a TouchGFX-based AUT.

This section will describe how the prototype is initialized and executed within the context of the AUT. Also, the core features of the target framework, i.e., event invocation, state acquisition and application reset, are described in terms of how the dynamics of the design presented in Section 4.4.4 are realized.

Initialization and execution

To avoid introducing test-specific code to the TouchGFX framework, a test-enabled HAL, HAL-TEST, replaces the default HAL used by the AUT on startup. The HALTEST class extends the base definition of the default HAL with core test framework members, which enables test automation on the AUT. On initialization of the AUT, the HALTEST replaces the default HAL as the framework engine, described in Section 2.4.3.2. This allows events produced by the test framework to be injected as if they were produced manually by the user.

To ensure compatibility with multiple target platforms a compile-time inheritance scheme based on preprocessor definitions is introduced by the HALTEST specification. Figure 5.8 illustrates this inheritance scheme.

![TestGFX HALTEST inheritance scheme.](image)

At compile-time the HALTEST class will identify which target-specific HAL to inherit from based on an available preprocessor definition found in the compiler configuration. Upon compilation to a specific target, e.g., the LPC4357DK device described in Section 5.3.2.1, a preprocessor definition “NXP” is found instructing the HALTEST to inherit from the device-specific HALNXP class.

Included in HALTEST is a TestAgent member and an extension of the default HALs handleEvents method. Essentially the HALTEST handleEvents method will override that of the parent class in order to call the “testProcedure” in the TestAgent. This approach was necessary as the original “handleEvents” was difficult to extend. The extension of “handleEvents” is shown in Figure 5.9.

With the compile time switch of the concrete HAL, the AUT is test enabled.

Event invocation

Event invocation concerns the injection of automated touch events into the AUT. To perform event invocation as described in Section 4.4.4, the prototype identifies the event type, creates an appropriate TouchGFX touch event, and pushes it on the Application instance.

Event invocation is realized through concrete commands, described in Section 4.4.2.2. An example of a command is the ClickCommand. In accordance with the Command Pattern [17], commands have an “Execute”-method which contains all the information necessary in order to invoke
Chapter 5. Validation of an Automated GUI Testing Framework

Figure 5.9: The test-specific extension of the handleEvents-method provided by HALTEST.

```c++
void HALNXP::handleEvents()
{
    while(1)
    {
        OSWrappers::waitForVSync();
        backPorchExited();
    }
}
```

void HALTEST::handleEvents()
{
    while(1)
    {
        OSWrappers::waitForVSync();
        _testAgent.testProcedure();
        backPorchExited();
    }
}

Figure 5.9: The test-specific extension of the handleEvents-method provided by HALTEST.

the corresponding touch event. For the ClickCommand this results in a TouchGFX \textit{ClickEvent} being created and passed to the “handleClickEvent” implemented by the Application instance. Figure 5.10 shows the implementation of the “Execute” for a ClickCommand.

```c++
void ClickCommand::Execute()
{
    ClickEvent::ClickEventType type;
    switch(_type)
    {
    case PRESS:
        type = ClickEvent::PRESS;
        break;
    case RELEASE:
        type = ClickEvent::RELEASE;
        break;
    case CANCEL:
        type = ClickEvent::CANCEL;
        break;
    default:
        break;
    }

    ClickEvent clickEvent(type, _xCoord, _yCoord, _zForce);
    Application::getInstance()->handleClickEvent(clickEvent);
}
```

Figure 5.10: Implementation of the “Execute”-method for a ClickCommand in the target framework.

The \textit{ClickEventType} used in the “Execute”-method shown in Figure 5.10 represents the nature of the touch interaction. Simulating a click event involves the invocation of two separate ClickEvents on the AUT: a \textit{pressed} event and a \textit{released} event. In the development of the prototype it was discovered that invoking multiple events in quick succession, i.e., by forwarding two or more events in the same “Execute”-method, resulted in a faulty state in the AUT. This issue is related to the \textit{temporal synchronization problem} in automated GUI testing, described in Section 2.2.3. In order to address this issue in the prototype an \textit{event invocation control} scheme is introduced.

The event invocation control scheme allows the AUT to process an event before addressing the next while controlling the frequency with which the TestAgent actively carries out test-related actions. This is realized by introducing a command FIFO queue to the TestAgent. Upon receiving a host command packet representing multiple commands to be executed on the target, the TestAgent will queue the corresponding commands and subsequently retrieve and execute the first command. The TestAgent proceeds to do this with each call of its “testProcedure”. While initially intended to be called once per event cycle, the frequency of this call is reduced by a fixed amount in order to ensure that the state of the GUI has settled before forwarding test events. Based on
a simple counter and a modulus, the implementation of this synchronization scheme is shown in Listing 5.1. The value specifying the frequency with which the TestAgent is active can be configured at application startup.

```c++
void HALTEST::handleEvents()
{
    while (1)
    {
        count++;
        OSWrappers::waitForVSync();
        if (count % _config.TEST_PROCEDURE_FREQ == 0)
            _testAgent.testProcedure();
        backPorchExited();
    }
}
```

Listing 5.1: TestAgent activity frequency.

Through the employment of the event invocation control scheme, deterministic test execution of events on the GUI is achieved.

**State acquisition**

State acquisition involves extraction of attributes from the Drawable objects present in the active view of the AUT. As described in Section 4.4.4, the state is acquired by means of a TouchGFX-specific StateClient as well as interpreters related to the Drawable hierarchy found in the TouchGFX core.

The prototype realizes state acquisition by means of the `TouchGFXStateClient` capable of accessing the root container of the current view described in Section 2.4.2.1. A `DrawableInterpreter` is used in order to extract state from the root container and its child elements recursively. Upon completion a text-based representation of the state of the GUI is generated. This process is shown in pseudo-code in Figure 5.11.

The “state”-reference used in the algorithm shown in Figure 5.11 refers to a char array statically allocated in stack memory. The state array is purposefully placed in the stack memory in order to anticipate memory use. Reuse of this array removes the need for sizable reallocations, which could perhaps result in memory leaks when using dynamic allocation found in, e.g., the String library. Also, due to the static nature of the allocation, runtime resource consumption can be calculated, which is crucial for the evaluation of the prototype framework. Once the state array is populated, it is transmitted to the host framework. This process is described in Section 5.3.1.3.

The “getType”-method used in the algorithm shown in Figure 5.11 refers to a method capable of discerning the deferred type of a Drawable object, e.g., Widget or Container. The prototype implements this method by extending the TouchGFX framework. Through polymorphism each Drawable object is expected to implement a “getType”-method. Other changes to the TouchGFX core include access to internal members. The complete list of changes made to the TouchGFX framework are described in detail in Appendix B.

While the use of dynamic memory allocation in the framework is minimal, certain conversion methods in the state interpretation rely on dynamic memory. This introduces a dependency to the
Chapter 5. Validation of an Automated GUI Testing Framework

Figure 5.11: Pseudo-code of the interpretation of state in the target framework.

Doug Lea memory allocator\textsuperscript{5} increasing the code size of the framework significantly. The impact of this dependency to code size is analyzed upon in Appendix C.

Application reset
As described in Section 4.4.2.1, resetting the AUT is subject to some measure of application-specific implementation. For the AUT used in the evaluation of the prototype, described in Section 5.3.2.2, application reset is achieved by overriding the current application model with a copy of the initial model, saved during initialization. Furthermore, the transition method “gotoMainScreen” is invoked in order to change the active view back to the initial view of the AUT. This is realized by the FrontendAAL implemented by the AUT.

5.3.1.3 Host- and target communication
This section describes the realization of the host- and target communication based on the binary packets presented in Section 4.4.5.1.

Command packet
The communication from the host- to the target framework in the prototype involves transmitting command packets over a serial communication interface. The specific command packets supported by the prototype include: click, drag, state and reset. Figure 5.12 shows the transmission of a click command packet to the target.

The command packets are created on the host by the TestExecutor described in Section 3.4.4.1. The Type-byte of a click command packet is set to “EVENT” and the Id-byte is set to “CLICK”. The payload contains the coordinates at which the click event should be initiated on the GUI, separated by a “;” delimiter.

Command interpretation on the target
Command packets are received on the target by the ProtocolClient described in Section 4.4.2.1.\textsuperscript{5}

\textsuperscript{5}http://g.oswego.edu/dl/html/malloc.html
The specific ProtocolInterpreter used in the prototype is based on the Interpreter Pattern [17]. The packet type and id are analyzed in order to forward the packet to the appropriate interpreter implementation. For the click command presented above, the interpretation results in two target-based click command objects, one representing a pressed event and one representing a released event. Each contains an x-, y-, and z-coordinate. The x- and y-coordinate specify the location at which to initiate the event, and the z-coordinate specifies the amount of force applied to the touchscreen interface.

State communication

Upon receiving a state command packet from the host, the target extracts the current state of the GUI from the AUT. The state is serialized by the StateClient which is also based on the Interpreter Pattern. For the prototype the state is serialized into a compressed JSON string which is converted to a byte array and sent to the host. The compression is performed by converting type- and attribute names to hexadecimal values thus reducing the size of the byte array. An example of a compressed JSON string is shown in Figure 5.13.

With the compression the state array is still much larger than the 22 byte payload in a packet. Therefore, the array is split into several packets where the Id-byte represents the status of the state
transmission. Figure 5.14 gives an example of a state transmission.

Figure 5.14: Transmission of the state of the GUI from target to host.

5.3.2 Test environment

In order to validate the functional requirements of the evaluation, described in Section 5.3, a test suite consisting of five test cases is executed on an AUT, both developed for this purpose, by means of the prototype described in Section 5.3.1 above. This section describes the setup of software and test equipment during the execution of the test suite. Section 5.3.2.1 gives an overview of the test environment, while sections 5.3.2.2 and 5.3.2.3 describe the AUT and the test suite respectively.

5.3.2.1 Overview

The test environment consists of a laptop with a Windows OS and the LPC4357DK device connected through a serial interface.

Figure 5.15 shows an overview of the test environment.

In order to execute the test suite, described in Section 5.3.2.3 below, the laptop is installed with the Robot Framework (currently version 2.8.4). The prototype framework, described in Section 5.3.1, is packaged as a JAR file referenced by the test suite.

The test environment includes two possible target environments. The active target depends on the COM port specified by the test suite. The “COM3”-port is connected to the LPC4357DK device by a serial RS232 interface. The device runs the AUT with the prototype test framework set to the “UART0” serial port of the device, described in further detail in Appendix B.

The “COM8”-port is connected to a simulator running on the laptop through a virtual serial bridge. The bridge is created with the Free Virtual Serial Ports tool (currently version 2.2.0). Similar to the LPC4357DK device, the simulator runs the AUT but with the prototype test framework set to the “COM7”-port connected to the virtual bridge.

5.3.2.2 Application Under Test

In order to validate the requirements for the evaluation of the TestGFX framework a TouchGFX application is developed by combining existing example applications included in the TouchGFX distribution. The examples, ButtonExample and DragExample, each present a single screen application demonstrating specific features of the TouchGFX framework. The two application screens, referred to as the button screen and the drag screen, are combined and their individual application models are merged resulting in a single application referred to as the Application Under Test (AUT).

**Button screen**

The purpose of the button screen is to present the capabilities of the application to handle simple touch events, i.e., clicks, and update the internal state of the application.

In the AUT the button screen is the initial screen shown when the application starts. Figure 5.16 shows the button screen of the AUT.

The buttons seen in the left portion of the screen increment and decrement the counter in the text field seen in the right portion of the screen. The button to the far right switches the active screen to the drag screen.

**Drag screen**

The purpose of the drag screen is to present the capabilities of the application to handle sophisticated touch events, i.e., dragging as well as snapping a GUI object to specific area of the screen. Snapping describes a behavior where releasing a dragged GUI object within a specific region of the screen will make it snap to a specific location inside this region. Figure 5.17 shows the drag screen of the AUT.

By pressing the square GUI object seen in the left portion of the screen it may be dragged and dropped into the region labeled “DROP HERE”. Releasing the draggable GUI object within its initial region or the “DROP HERE”-region will make it snap into place. Releasing the GUI object
outside one of these regions will return it to its initial region. The button to the far left of the drag screen switches the active screen to the button screen.

Transitions
As described in Section 2.4, transitions specify how the application switches from one screen to another. A transition is made by a call to a “makeTransition”-method within the TouchGFX framework which requires the presenter and view of the new screen as well as any animations that should be shown during the transition. A transition is constrained to a one-way switch from one screen to another with animations being optional. The prototype implements three transitions: the initial transition to the button screen at application startup, a transition to the button screen after application startup and a transition to the drag screen. The initial transition has no animation while the subsequent transitions perform a sliding effect.

Known defects
A defect describes an unintentional behavior in the execution of the AUT. A defect may result in an unexpected state of the application or in unexpected and incorrect results being produced. Since the AUT is a product of the combination of two existing single screen applications, certain defects
have been introduced due to the transitioning between the application screens. The following user scenarios will result in an incorrect state:

1. **Button screen defect:** At application startup the button screen, shown in Figure 5.16, is active with the decrement-button disabled as the counter is set to “0”. After incrementation the counter shows “1” and the decrement-button is enabled. After transitioning to the drag screen and back to the button screen the decrement-button is disabled while the counter still shows “1”. This defective state is the result of poor state management between transitions. The state of the buttons are not retrieved as the button view is created.

2. **Drag screen defect:** While the drag screen, shown in Figure 5.17, is active, the draggable GUI object is pressed and dragged to the “DROP HERE”-region where it is released and snapped into place. After transitioning to the button screen and back to the drag screen, the draggable GUI object is reset to its initial location. This defect is also the result of poor state management between transitions. The position of the draggable GUI object is not persisted between transitions.

These known defects are used in the evaluation of the functional suitability of the GUI testing framework, as mentioned in Section 5.2.2.1. The test suite, described in Section 5.3.2.3 below, includes test cases designed to fail due to the existence of these defects. The purpose of the failing test cases is to present the capabilities of the framework to detect and report on defects in the AUT.

### 5.3.2.3 Test suite

As described in Section 5.3.1.1, the test suite is based on actionword- and keyword-scripting and is written in plain text with the RobotFramework-EclipseIDE plugin for Eclipse. This section presents the test suite divided into its three components: settings, test cases and keywords.

#### Settings

The settings-section of a Robot Framework test script enables a tester to specify external test libraries to look keyword implementations and test setup- and teardown methods. Figure 5.18 presents the settings-section of the test suite developed for the evaluation of the prototype.

```plaintext
### Settings ###
Library com.prototype.testlibrary.PrototypeTestLibrary
Suite Setup  test suite setup COM3
Suite Teardown test suite teardown
Test Setup   Run Keywords   test case setup validate the button screen
Test Teardown test case teardown
```

Figure 5.18: The settings-section of the test suite implemented for the evaluation of the prototype.

The test library referenced by the test suite is described in Section 5.3.1.1. This library implements both test suite and test case setup- and teardown methods. The “COM3” argument provided to the test suite setup is the serial port connected to the LPC4357DK device, as shown in Figure 5.15. The test setup is specified with the built-in “Run Keywords”-method, which is used in order to execute both the “test case setup”-keyword implemented by the test library, and the “validate the button screen”-keyword implemented as a built-in keyword as shown in Figure 5.20.
Chapter 5. Validation of an Automated GUI Testing Framework

Test cases
Figure 5.19 shows the test cases implemented for the evaluation of the prototype.

```plaintext
*** Testcases ***
Increment and decrement counter
   Assert that "CounterBox" says "0"
   Press "UpButton"
   Press "UpButton"
   Press "UpButton"
   Assert that "CounterBox" says "3"
   Press "DownButton"
   Assert that "CounterBox" says "2"

Transition between screens
   Go to drag screen
   Go to button screen

Increment counter then transition and decrement counter
   Assert that "CounterBox" says "0"
   Press "UpButton"
   Assert that "CounterBox" says "1"
   Go to drag screen
   Go to button screen
   Press "DownButton"
   Assert that "CounterBox" says "0"

Transition to drag screen and drag object
   Go to drag screen
   Drag "DraggableBox" by "228,-46"
   Assert that "DraggableBox" has moved by "265,0"

Drag object then transition and validate new position
   Go to drag screen
   Drag "DraggableBox" by "228,-46"
   Assert that "DraggableBox" has moved by "265,0"
   Go to button screen
   Go to drag screen
   Assert that "DraggableBox" has moved by "265,0"
```

Figure 5.19: The test cases of the test suite implemented for the evaluation of the prototype.

The test cases are designed to utilize all of the functionalities included in the AUT. Two of the test cases, “Increment counter then transition and decrement counter” and “Drag object then transition and validate new position”, are designed to fail due to the known defects described for the AUT in Section 5.3.2.2.

In order to demonstrate the reusability of the test specification as well as to increase the readability for test personnel without programming skills or experience, the test cases utilize high-level actionwords implemented in the keywords-section shown in Figure 5.20.

Keywords
The keywords-section of a Robot Framework test script enables a tester to create new built-in keywords and actionwords with a high degree of reusability provided. Figure 5.20 shows the keywords-section of the test suite developed for the evaluation of the prototype.

The bottom four keywords provided for this test suite are created in order to increase readability. The top four keywords are created in order to demonstrate reusability and may be referred to as actionwords as they provide a higher level of abstraction.
5.3.3 Results

This section presents the results of the evaluation of the GUI testing framework. As described in Section 5.2, the framework is evaluated on five quality properties by a set of requirements and measurements. The measurements, described in Section 5.2.2, are validated by means of the prototype described in Section 5.3.1 above. Some of the measurements are validated by static analysis of the source code of the prototype while others are validated through execution of the prototype in the test environment described in Section 5.3.2. As described in Appendix C, some of the measurements for performance require additional tools. Code size and memory consumption are benchmarked by the *IAR Embedded Workbench* IDE, while the processor load is measured by adding a custom component to the AUT calculating this value.

The execution of the test environment yields the test report shown in Figure 5.21. As described in Section 5.3.2.3 above, the two test cases that fail are expected to do so due to the known defects described for the AUT in Section 5.3.2.2.

The measurements of the evaluation are all simple statements that may be answered with a yes or a no. The results of the evaluation are presented in a stacked bar chart where each quality property is represented by a bar and each bar contains a stack of successful measurements and a stack of failed measurements making up 100% of the measurements taken for that property. The results are thus based on a simple normalization of the measurements, e.g., Functional suitability has a total of ten measurements which makes each measurement correspond to 10% of the bar.

Figure 5.22 shows the results of the evaluation. A detailed description of the outcome of the
Chapter 5. Validation of an Automated GUI Testing Framework

Figure 5.21: Test report for the execution of the test suite as part of the framework evaluation.

Figure 5.22: Results of the evaluation presented in stacked bar chart.
5.4. **Discussion of results**

In order to validate the design of the GUI testing framework presented through this thesis an evaluation process based on the ISO/IEC 25000 set of standards has been introduced. In order to validate the measurements of this process a proof-of-concept prototype of the GUI testing framework has been developed and executed in a test environment created for this purpose. The results of the evaluation show that we, in terms of the evaluation process, have succeeded in creating a GUI testing framework with all of the functionalities necessary. The model utilized by the test library enables heuristics-based GUI object recognition and provides complete abstraction between the test script and the GUI objects of the AUT. The GUI object recognition as well as the AUT reset implemented in the target framework make for significant results for the automated GUI testing of embedded systems.

The results for the performance efficiency of the framework show that we have succeeded in creating a GUI testing framework that is capable of sophisticated remote test automation with low resource consumption. The code size of the target framework is less than the lower bound values specified for the TouchGFX framework, although it could be even smaller. The use of dynamic memory allocation in the state interpreter introduces a library that significantly increases the code size. The measurement of the time it takes to automate a user action exceeds the 2 seconds and fails the validation. This is primarily due to the synchronization scheme implemented in the target framework. The execution of test events is delayed by a fixed value that has to accommodate for complex events, making simple events wait much longer than what is necessary.

The results for maintainability and transferability of the framework show that we have succeeded in creating a GUI testing framework that is easy to maintain and extend with new features and supported platforms. The measurement that fails for maintainability is due to the command abstraction in the host framework. The same interface is used by the test libraries and the core, while the libraries only require methods for instantiation and execution, and the core requires access to information on the interpretation of a specific command. The measurement that fails for the transferability of the framework is due to the implementation of the state reset in the target framework which has to be provided by the AUT.
This chapter concludes on the results achieved in this thesis. The results are comprised of the conceptual framework design presented in chapters 3 and 4 and the evaluation of a prototype framework presented in Chapter 5. The evaluation is based on the thesis goals presented in Chapter 1 as well as the ISO/IEC 25000 set of standards for software quality requirements and evaluation.

6.1. Overview

This chapter concludes on the design of a GUI testing framework for low-resource embedded systems presented in earlier chapters. This includes a discussion on areas where the design can be improved based on the results of the evaluation, as well as personal learning outcomes and reflections on the thesis work.

In Section 6.2 the results achieved are presented and discussed with regard to thesis goals and the theory presented in Chapter 2. Future work, expanding on the capabilities of the framework design and prototype, is presented in Section 6.3. Our personal learning outcomes are described in Section 6.4 and finally Section 6.5 concludes this chapter and the thesis.

6.2. Results achieved

Throughout this thesis we have worked from the hypothesis that we can perform automated GUI testing on low-resource embedded devices. In relation to the hypothesis, the overall scientific goals of this thesis, as presented in Chapter 1, are to design a GUI testing framework capable of:

1. performing automated GUI testing of embedded applications running on a remote target device or in a simulated environment.
2. automating simple as well as complex test cases on a low-resource device.
3. evaluating and reporting on test cases executed remotely.

In order to achieve automated GUI testing on low-resource embedded devices it was first necessary to perform elaborate research on existing methods and state of the art solutions available. In Chapter 2 we present information about the core concepts and principles of GUI testing as well as methodologies and approaches aimed specifically at test automation on embedded devices. This knowledge has proven relevant for developing a framework design satisfying the thesis goals.
The design of the framework consists of a host- and a target design described in chapters 3 and 4 respectively. The decision to use a partitioned architecture was made based on related work on the constraints of low-resource embedded test automation. In order to provide maintainability and extensibility, the host framework has a considerable size and resource consumption that would not fit within a low-resource target device. By partitioning the framework the resource constraints may be lifted from the larger part of the framework.

Based on the host- and target designs a proof-of-concept prototype of the complete framework, described in Section 5.3.1, was developed. Through a standardized evaluation process, described in Section 5.2, the prototype was validated against the thesis goals. The results of the evaluation are presented in Section 5.3.3.

Assessments of the host and target designs are provided in sections 6.2.1 and 6.2.2 below. An assessment of the evaluation performed is provided in Section 6.2.3.

6.2.1 Host design

In Chapter 3 an overview of the complete design of the framework was presented along with a detailed description of the host design. The primary objective of the host design was to provide great extensibility of the framework and maintainability of test specifications.

Robot Framework

The design utilizes the open source GUI testing framework Robot Framework, described in Section 5.3.1.1, as its front-end enabling test specification and reporting. The decision to use Robot Framework was made due to the flexibility of the test specification and the extensibility of the framework. The format of test specification in Robot Framework is very flexible and the use of actionword- and keyword-scripting, as described in Section 2.2.2.1, provides reusability and abstraction that increases the maintainability of test scripts. The support for Java test libraries means that Robot Framework is easily extended with our own back-end for GUI testing of remote embedded devices. The test libraries also increase the extensibility as well as maintainability of the overall solution as application-specific functionality and configurations may be isolated to the test library, so that a test library may be developed for each Application Under Test (AUT). Furthermore, a test script may utilize more than one test library. This means that test cases of the same test suite may execute on different applications on different target devices.

Test libraries

The test libraries utilize a structured model of the AUT containing GUI objects including their position, size, contents etc. The GUI objects in the model are referenced by the test script using logical names providing abstraction between the test specification and the AUT and enabling test-driven development. The model is used in order to target specific GUI objects for the automation of user actions. The GUI objects in model are not used directly however, but are used in the heuristics-based GUI object recognition described in Section 5.3.1.1. This process attempts to locate the actual GUI object based on the information provided by the corresponding model. The purpose is to increase the maintainability of test scripts as they become more robust against minor changes to GUI objects. The combination of test libraries and actionword- and keyword-scripting thus provides expressive power as well as abstraction, which increases functional suitability and maintainability of the framework as well as the test specifications.
**TestGFX**

The back-end, TestGFX, implements the embedded test automation. It is responsible for the communication with the target device or simulator, sending commands, and interpreting the results so that they may be validated by the test library. TestGFX utilizes component-based development in order to increase maintainability and extensibility. A generic core component defines the internal software interfaces and the interface implementations are provided by separate components used by the test library. The core has no knowledge of the interface implementations, which means that they may change without affecting the overall behavior of the framework. As implementations are located in separate components they may also be altered and substituted without affecting any other parts of the framework thus increasing maintainability and extensibility.

**Concluding remarks**

Based on the prototype framework and the evaluation thereof, we conclude that our host design succeeds in fulfilling the thesis goals regarding *functional suitability* and *maintainability* of the test specifications and the GUI testing framework.

The decision to use an existing framework to extend upon was made after it had been decided to use a partitioned architecture. As the front-end is located on a host machine without any resource constraints, the use of existing, proven tools is a feasible solution. A tool capable of performing low-resource embedded GUI testing was not found however, and the extension of an existing tool was considered instead. With actionword- and keyword-scripting as well as Java test libraries, the Robot Framework provides a high degree of extensibility and was chosen for the solution.

The decision to use a model of the GUI of the AUT was based on a decision to perform heuristics-based GUI object recognition as is performed in state of the art scriptless GUI testing frameworks, described in Section 2.2.2.1. In automated GUI testing, the robustness of test scripts against changes to the GUI is a common challenge. Test scripts that break due to an insignificant redesign present poor maintainability and the GUI object recognition was thus a crucial feature to implement.

### 6.2.2 Target design

In Chapter 4 we presented the design of the target framework. The target environment may be run on the same physical machine as the host, through a simulator, or it may be run on a separate target device. The target framework is designed for applications based on the TouchGFX framework and focuses on *performance efficiency* and *compatibility* with TouchGFX and the AUT.

**TestGFX**

Similar to the host the target design utilizes component-based development with a generic core component. The core contains methods to receive instructions from the host and execute them on the AUT in a controlled manner. With a single point of entry, the core of the test framework is introduced into the target environment by the enabling GUI framework, TouchGFX. The context of the test framework and the TouchGFX component responsible for introducing it is the Hardware Abstraction Layer (HAL). The decision to introduce the test framework through the HAL was made based on the related work presented in Chapter 4 and the details of the TouchGFX architecture. The related work describes an approach to remote test automation where instrumentation of a common abstraction shared by heterogeneous devices is targeted [27, 31]. The target framework presented in this thesis uses the TouchGFX framework as the common denominator by means of the HAL. The HAL is responsible for forwarding hardware events to the AUT and subsequently realize graphical output on the target device. Introducing our test component into the HAL context enables the host to act as a new source of events on par with the touchscreen interface.
Chapter 6. Concluding Remarks and Future Work

**TouchGFX automation**
The target framework enables remote automation of sophisticated actions on the AUT. The command and state components integrate with TouchGFX in order to automate touch events and retrieve the state of the GUI respectively. Sending events to the AUT through TouchGFX is not straightforward however, as rapid introduction of events may cause the AUT to enter an unexpected state. Because of this, an event queue was added, from which events are retrieved and executed at a vsync event in the HAL providing synchronization with the rendering of the GUI. This provides deterministic and reliable execution of GUI events on the AUT.

**Communication**
With a partitioned architecture, a means of remote communication between the host and target frameworks is necessary. For the prototype framework an RS232 serial communication interface was chosen due to its wide use in embedded devices. With a protocol based on compressed JSON for reduced overhead split into multiple packets, the RS232 interface has proven sufficient in sending even complex state messages from the target to host. While the protocol is based on related work, the need to transmit the state of the GUI meant that a means to transmit structured data was necessary, which led to JSON. As an alternative to XML, JSON is more lightweight and as type- and attribute names were substituted for hexadecimal values, the overhead of state messages was significantly reduced. Based on related work, the packets of the protocol are fixed in size. This reduces overhead as well as packet meta-information can be reduced to two bytes specifying the id and type of the packet.

**Concluding remarks**
Along with the host, the target design has been validated by evaluation of a proof-of-concept prototype. Based on this, we conclude that our target design succeeds in fulfilling the thesis goals regarding performance efficiency and compatibility. The target prototype implements several features parallel to those found in the host design, such as commands and command interpreters. Besides the remote communication, the target framework integrates with TouchGFX and the AUT and for this reason compatibility and especially co-existence were important properties. However, limitations to the TouchGFX framework complicated the application reset and state acquisition. In order to work around these obstacles without compromising the compatibility of the test framework, extensions to TouchGFX were made as well as some measure of application-specific implementation of the reset-functionality. An example of an extension is the synchronization with the vsync event mentioned above, which is made available by extension of the HAL. Improvement of this synchronization scheme is considered an objective for future work however, as it provides a fixed delay of test execution instead of adapting to the actual length of the event or animation carried out on the GUI.

6.2.3 **Evaluation**

In order to validate the GUI testing framework against the thesis goals, an evaluation process, described in Section 5.2, was used. The evaluation process is based on the ISO/IEC 25000 set of standards for software quality requirements and evaluation [5, 12] and has been used along with the proof-of-concept prototype of the framework.

**Quality properties, requirements and measurements**
From the ISO/IEC 25010 product quality model, five quality properties were selected; functional suitability, performance efficiency, compatibility, maintainability and transferability. The eval-
Evaluation

Evaluation was scoped to these five properties in terms of their appropriateness to the thesis goals. The remaining properties of the product quality model, operability, reliability and security, were disregarded as they were deemed out of scope of the objectives of this thesis work.

For each of the five quality properties a set of requirements are specified based on the ISO/IEC 25030 standard [7]. Each requirement is validated by a set of measurements based on the ISO/IEC 25020 standard [6]. A measurement is a simple statement that is answered with a yes or no and the score of a quality property is, for this thesis work, defined as the percentage of succeeded measurements out of the total number of measurements for that property. As such the score is a simple unweighted normalization of the measurement results.

The measurements are validated by static analysis and benchmarking of the prototype framework as well as the execution of a test suite on a TouchGFX-based AUT, both developed for this purpose and described in Section 5.3.2.

Results
The evaluation results, presented in Section 5.3.3, are very satisfying. The prototype shows perfect results in functional suitability and compatibility, while performance efficiency, maintainability and transferability fail a measurement each.

Requirements for functional suitability include the ability of the prototype to support expected features in terms of automating the GUI of the AUT, finding and reporting defects and providing a high degree of maintainability of test specifications. With the AUT and test suite, the prototype was capable of performing all actions specified in the test suite including assertion of the state of the GUI. Furthermore, the prototype was capable of detecting and reporting on both of the known defects described in Section 5.3.2.2. The requirements for compatibility include the ability to execute tests on the AUT reliably and deterministically while not affecting the execution of the AUT. In order to validate this requirement manual execution of the test suite was performed and compared to the automated execution through the prototype. Results show that the presence of the framework has no significant effect on the execution of the AUT.

The performance efficiency of the framework is quite satisfying, although it fails one of the measurements of the evaluation. The measurements for resource consumption and time behavior of the state acquisition satisfy the specified thresholds, but the time behavior of automated user actions exceeds the threshold of 2 seconds. This may seem strange as both user actions and state acquisition involves sending a single command from the host to the target and state acquisition also includes sending the state result back to the host. This behavior is due to the fixed delay in touch event execution, mentioned in Section 6.2.2 above, and the fact that the user action that was used in the measurement was the click of a button. While a button click is a simple command, it translates to two events on the target: a button pressed event and a button released event. The execution of each of these events incurs the same fixed delay.

The maintainability of the prototype framework is evaluated by code review based on the principles of SOLID design. While the overall maintainability of the prototype is satisfying, the host design fails on the Interface Segregation Principle. This is due to the command abstraction in the host, which provides the same interface to the test library and the TestGFX core, although their use of commands is very different. This is a minor issue however, which could easily be remedied in future work.

The transferability of the prototype, which is also evaluated by code review, fails on a measurement regarding the support of a new AUT. The reset of the AUT is subject to application-specific implementation of the Application Abstraction Layer (AAL) provided by the test framework. While the implementation is simple, this issue should be considered by future work.
Chapter 6. Concluding Remarks and Future Work

6.3. Future work

During the progress of this thesis work several areas for future work have been identified. This section presents major areas for extension as well as improvement of the GUI testing framework.

6.3.1 GUI survey tool

The test libraries rely on a structured model of the GUI of the AUT. In a test specification the test developer may refer to GUI objects such as buttons and text areas by a logical name which maps to a GUI object in the model. For this prototype the model is implemented as a part of the test library. Future work should include a GUI survey tool capable of generating this model from an AUT, possibly through the simulator provided with the TouchGFX distribution. As the functionality of such a tool would be similar to the state acquisition process already implemented in the prototype, we believe that this tool could utilize the TestGFX back-end as is, with another front-end as well as extensions to the TouchGFX simulator.

6.3.2 Record and playback

In future work a record and playback feature for the framework should be considered. The ability to record user scenarios and generate a more or less complete test script can increase the usability of a GUI testing framework significantly. While the generated scripts may require some additional work, test recording may reduce the learning curve for new users and greatly reduce the time spent on test specification for experienced users. As with the GUI survey tool presented above, we believe that a test recording tool may be implemented by extension of the TouchGFX simulator.

6.3.3 GUI framework abstraction

The target design of the prototype is very dependent on the enabling GUI framework TouchGFX. While the design includes abstractions such as the command and a state components, it also depends on a globally accessible application instance providing certain information. Future work could include further abstraction of the enabling GUI framework. One approach could be to employ the Adapter Pattern [17], specifying a “contract” which must be fulfilled by the AUT or the enabling framework in order to enable test automation. This solution is similar to the related work by Loft presented in Section 4.3.2.

6.3.4 GUI event synchronization

The prototype framework implements GUI event synchronization as a fixed delay in test execution. The delay is meant to provide the GUI framework with enough time to process and redraw changes to the GUI. As such, it has to be long enough to support both short click events as well as long transition animations. Future work should include a more intelligent synchronization scheme. If a delay is utilized as in the current solution, it should be capable of adjusting itself to the length of the event or animation.

6.3.5 General improvements

The GUI testing framework has been realized in a proof-of-concept prototype capable automating click and drag events as well as retrieving the state of the GUI. Future work should include
additional user actions such as writing text into a text area by means of an on-screen keyboard. A GUI testing framework should have a high degree of reliability and fault-tolerance in order to provide confidence in the results produced. These properties were considered out of scope for this thesis work but should be considered for future work.

6.4. Personal learning outcome

Working on this thesis has not only increased our knowledge within embedded GUI testing but also improved our ability to conduct research in academia. Through the thesis work, we have gained the ability to achieve a quick overview and extract relevant information from scientific research papers and theses. The ability to quickly assess the relative contribution of a paper toward the research goals is essential. While a field of study have excess of scientific material readily available, only a subset of this may be applicable to the specific case. Assessing the contributions of papers has enabled us to quickly disregard material with little to no value to the thesis work. 

The knowledge attained within the field of embedded test automation will undoubtedly prove useful in future work with embedded systems. Due to an increasing complexity and versatility of embedded systems, test automation is becoming increasingly important in order to provide the level of quality required. Furthermore, through the practical thesis work first-hand experience with the design and realization of a system comprising multiple platforms and technologies has been achieved. With the advent of the Internet Of Things, the knowledge and experience of developing a system of systems with interrelating technologies and remote synchronization are invaluable.

6.4.1 Parallel research and development

During the early research into the field of embedded test automation, several solutions relying on a partitioned architecture were found [24, 27, 31, 48]. Some of these solutions work under the same resource constraints as set for this thesis work, and the partitioned architecture was presented as an approach to significantly reduce resource consumption on the target device. This led to a division of the thesis work in a host and target branch with each of the two authors of this thesis responsible for one of these branches.

Through parallel research into related work and subsequent dissemination of the achieved knowledge, the direction of the design and the thesis as a whole were aligned. Practical thesis work in the branches was likewise carried out in parallel, with each of the authors focusing on his own part of the prototype framework, with the exception of the communication protocol connecting the two parts.

Throughout working on this thesis, daily meetings between the authors have been held in order to align the current work and the expectations for the work ahead. Furthermore, weekly meetings have been conducted with the attendance of the thesis supervisor. This has provided valuable feedback to the planning of the thesis work as well as the contents of this thesis. Through these meetings questions and uncertainties were addressed and clarified, and we learned the importance of dissemination of knowledge, even in small groups.

Working from a hypothesis employing a scientific approach with research, elaboration, and evaluation, has been a new experience. The research made within embedded test automation resulting in the division of the work, and the research made within the parallel branches has generated a large knowledge base which proved essential during the elaboration of the framework design and the proof-of-concept prototype. A standardized evaluation process has also proven invaluable due to the confidence it provides in the results that are produced. Figure 1.1 illustrates the scientific
approach employed throughout this thesis work.

6.4.2 An engineering perspective

During our education towards masters in software engineering, we have trained in solving problems by providing the best solution possible within the given time- and resource constraints. In some cases a solution has to be developed from nothing, while in other cases it may be more feasible to utilize existing technology in order to reduce the cost and the time to market. The ability to integrate with and extend upon existing technologies is thus an important skill which has proven useful in our thesis work.

As shown in Figure 5.3, our contribution to the GUI testing framework is an extension to the Robot Framework, enabling it to perform GUI testing on remote TouchGFX applications. This includes the development of a test library and a back-end providing remote communication and integration with TouchGFX. A large part of our thesis work has thus been research into the specifics of these frameworks in order to provide a design that extends rather than alters the existing technologies.

6.5. Final remarks

The goals of the thesis were to design a GUI testing framework capable of performing automated GUI testing on applications running on low-resource embedded devices and to improve our skills and knowledge on this subject. We are pleased with the results and can with confidence say that our thesis work successfully meets the specified goals.

With a partitioned architecture reducing resource consumption on the target device and by utilizing existing technologies a GUI testing framework capable of remote test automation on heterogeneous targets has been designed. While the host framework provides a generic approach to GUI testing, the target framework integrates with the TouchGFX GUI framework. Through this integration the target framework is able to abstract from the underlying hardware platform with the one exception of device drivers for serial communication, and thus, no changes are required in order to support a new device.

The conceptual design of the GUI testing framework is validated by a standardized evaluation process. For the evaluation, a proof-of-concept prototype has been developed along with a representative test suite and TouchGFX application. The results of the evaluation provides evidence that our framework design is capable of sophisticated GUI testing with high maintainability and low resource consumption fulfilling the primary objectives of this thesis.

Through the thesis work, we have gained confidence in our skills and our acquired knowledge of performing scientific research. We feel confident in our ability to present solutions and communicate designs and concepts based on thorough scientific research.

In conclusion, modern embedded GUI applications are quickly becoming too complex for manual testing. Due to the heterogeneity of embedded devices however, automated test frameworks are often limited to a single family of applications or platforms. State of the art GUI testing frameworks provide a part of the solution but are often targeted at smartphones or similar devices with an abundance of resources available. This thesis presents a solution to automated GUI testing on low-resources devices based on the TouchGFX GUI framework and evaluated in adherence with ISO/IEC standards.

We sincerely hope that the design and approaches presented in this thesis will contribute to the research made within embedded test automation and that future research may incorporate or in any way benefit from our work.
Bibliography


[47] A. Thiel. androidscreeencast - desktop app to control an android device remotely.


Appendices
APPENDIX A

Terminology

This appendix describes terms and abbreviations used throughout the thesis. The contents of this appendix are presented in alphabetical order.

**AUT** Application Under Test. The software application that is currently being tested.

**BIT code** Built-In Test code. BIT code describes test specific code, which is integrated into the source code of the System Under Test (SUT) in order to enable testing of the same.

**CI** Continuous Integration is the practice of continuously integrating pieces of a software product, building it and running automated tests. It utilizes several technologies such as version control systems and build servers.

**Designated tester** A testing professional with limited programming expertise.

**DSL** Domain-Specific Language. A programming language of limited expressive power specific to a certain domain.

**Embedded System** An embedded system is a complete system dedicated to a single, or a few use case(s) within a larger system of systems.

**Framework engine** A software component, which drives a frameworks execution forward through high-level domain abstractions of lower-level hardware events.

**GUI** Graphical User Interface. A UI based on graphics in order to inform the user of the internal state of the system. Relies on touchscreen technology or other external hardware in order to accept user input.

**Host** A PC or similar device defined by increased resource and abstraction capabilities compared to a Target.

**ISO/IEC** International Organization for Standardization and International Electrotechnical Commission. Standards organizations responsible for maintaining, promoting and facilitating standards in the fields of Information Technology (IT) and Information and Communications Technology (ICT).


ISO/IEC 25030  Also, the ISO/IEC 25030 Quality requirements. A standard providing requirements and recommendations for the specification of software quality requirements.

LPC4357DK  LPC4357 Developer’s Kit\(^1\). A physical low-resource embedded device supported by the TouchGFX framework and supplied by Mjølner Informatics A/S for the development of the thesis GUI testing framework prototype.

Regression Testing  To repeat the execution of test cases after changes have been made to the AUT. The aim is to ensure that changes have not introduced new defects in the AUT or other dependent parties.

RS232  A standard for serial communication transmission of data supported by a wide range of embedded devices such as the LPC4357DK.

SUT  System Under Test. Refers to a system that is being tested for correct operation. A special case of a software system test is in an application, which, when tested, is called an Application Under Test (AUT).

Target  A physical or simulated device on which the AUT is executed.

Test-driven development (TDD)  A software development process that relies on first producing an initially failing test case and subsequently implementing code to pass the test.

\(^1\)http://www.embeddedartists.com/products/kits/lpc4357_kit.php
Implementation of the Proof-of-Concept Prototype

This appendix presents a detailed description of the implementation of the proof-of-concept prototype. Furthermore, a description of the use of the prototype is provided.

B.0. Overview

The following sections provide a detailed description of the prototype GUI testing framework presented in Section 5.3.1. Section B.1 describes the implementation of the host framework while Section B.2 describes the target framework implementation. Section B.3 describes the implementation of the communication protocol and finally Section B.4 presents a short manual describing how the prototype may be used.

B.1. Host implementation

The implementation of the host prototype is based on the host design described in Section 3.4. It consists of the Robot Framework, an open source, actionword- and keyword-scripting GUI testing framework, a test library and the TestGFX host prototype.

The following sections present the details of the host prototype implementation. Section B.1.1 describes the development environment and the tools used. Sections B.1.2 and B.1.3 describe the implementations of the test library and TestGFX on the host respectively.

B.1.1 Development environment and tools

During the implementation of the host prototype the open source IDE Eclipse\(^1\) has been used. In order to specify and maintain test suites in the same IDE as the host prototype, the Robot Framework plugin for Eclipse, RobotFramework-EclipseIDE\(^2\), has been used. Figure B.1 shows the host prototype and the test suite opened in the Eclipse IDE.

The open source, actionword- and keyword-scripting GUI testing framework Robot Framework has been used as the front-end of the host framework. The Robot Framework is a Python package

\(^1\)http://www.eclipse.org/downloads/packages/eclipse-standard-432/keplersr2
\(^2\)https://github.com/NitorCreations/RobotFramework-EclipseIDE
Appendix B. Implementation of the Proof-of-Concept Prototype

Figure B.1: The host prototype opened in the Eclipse IDE.

and may be downloaded from the Python web site\(^3\), the version used for this prototype was 2.8.4.

### B.1.2 Test library implementation

The test library, “PrototypeTestLibrary”, is implemented in Java. With the TestGFX back-end it implements keywords used by the test suite. Keywords include test setup and -teardown methods, instantiating the serial communication client as well as TestGFX. Other keywords send user actions to the Application Under Test (AUT) such as the “Press” and “Drag” keywords provided in Listing B.1.

```java
public void Press(String objName) throws Exception{
    ButtonModel button = (ButtonModel)getGUIElement(objName);
    if (button == null)
        throw new Exception("Error: Could not find the object named" + objName + ", the AUT model may be out of date.");
    RectModel clickRect = getObjectClickLocation(button.Rect);
    ClickCommand clickCommand = new ClickCommand(clickRect.X, clickRect.Y, 0);
    testClient.Execute(clickCommand);
}

public void Drag(String objName, int x, int y) throws Exception{
    StateModel guiObject = getGUIElement(objName);
    if (guiObject == null)
        throw new Exception("Error: Could not find the object named" + objName + ", the AUT model may be out of date.");
    RectModel clickRect = getObjectClickLocation(guiObject.Rect);
    ClickCommand clickCommand = new ClickCommand(clickRect.X, clickRect.Y, 0);
    testClient.Execute(clickCommand);
}
```

\(^3\)https://pypi.python.org/pypi/robotframework
Test library implementation

```java
throw new Exception("Error: Could not find the object named" +
        objName + ", the AUT model may be out of date.");

RectModel clickRect = getObjectClickLocation(guiObject.Rect);
DragCommand dragCommand =
        new DragCommand(clickRect.X, clickRect.Y, clickRect.X + x,
        clickRect.Y + y);
testClient.Execute(dragCommand);

private StateModel getGUIElement(String objName) {
    StateModel model = guiElementRepository.get(objName);
    if (model == null)
        return null;
    return testClient.Identify(model);
}
```

Listing B.1: Press and drag keywords implemented in the test library.

The keywords sending user actions to the AUT, as well as those validating its state, use the
"getGUIElement"-method in order to identify the targeted GUI object in the current state of the
AUT. With a logical name, provided by the test suite, this method retrieves the model of the GUI
object and sends it to the TestGFX back-end for heuristics-based GUI object recognition. Listing
B.2 shows the model used by the test library for this prototype.

```java
private static final Map<String, StateModel> guiRepository;
static
{
    TextFieldModel counterField =
        new TextFieldModel(new RectModel(258, 61, 152, 154));
    ButtonModel counterButtonUp =
        new ButtonModel(new RectModel(67, 60, 130, 56));
    ButtonModel counterButtonDown =
        new ButtonModel(new RectModel(67, 158, 130, 56));
    ButtonModel nextButton =
        new ButtonModel(new RectModel(420, 75, 55, 131));
    ScreenModel buttonScreen =
        new ScreenModel(new RectModel(0, 0, 480, 272));
    ButtonModel dragBox =
        new ButtonModel(new RectModel(52, 109, 70, 72));
    ButtonModel backButton =
        new ButtonModel(new RectModel(0, 75, 55, 131));
    ScreenModel dragScreen =
        new ScreenModel(new RectModel(0, 0, 480, 272));
    counterField.Content = "0";
    buttonScreen.Content = new ChildrenModel(counterField,
        counterButtonUp, counterButtonDown, nextButton);
```
Appendix B. Implementation of the Proof-of-Concept Prototype

Listing B.2: The model utilized by the test library in the prototype framework.

```java
dragScreen.Content = new ChildrenModel(dragBox, backButton);

guiElementRepository = new HashMap<String, StateModel>();
guiElementRepository.put("CounterBox", counterField);
guiElementRepository.put("UpButton", counterButtonUp);
guiElementRepository.put("DownButton", counterButtonDown);
guiElementRepository.put("NextButton", nextButton);
guiElementRepository.put("ButtonScreen", buttonScreen);

guiElementRepository.put("DraggableBox", dragBox);
guiElementRepository.put("BackButton", backButton);
guiElementRepository.put("DragScreen", dragScreen);
```

B.1.3 TestGFX host implementation

The TestGFX host prototype is implemented in Java. It consists of the five components: core, commands, models, parsers and transports. The core is a generic component which, through its TestClient class, serves as the primary point of entry for the test library. The commands are the secondary point of entry as they are instantiated by the test library and passed to the TestClient. The transports provide the communication clients, also instantiated by the test library and passed to the TestClient. The parsers are used by the core in order to translate the state received from the AUT and the specific parser to use is instantiated by the test library along with the communication client. The models are used by the parser and the test library in order to express an expected or an actual state of the GUI of the AUT.

The complete TestGFX host implementation, as it appears in the Eclipse IDE, is shown in Figure B.2.
TestGFX host implementation

Figure B.2: TestGFX on the host as it appears in Eclipse.
B.1.3.1 GUI object recognition

In order to send user actions to AUT or receive its state, the test library first attempts to locate the actual GUI object targeted. It passes a model of the GUI object to the TestClient which passes the model on to the TestObjectIdentifier class which implements the heuristics-based GUI object recognition algorithm. A part of this algorithm is shown in Listing B.3, including the main method, “Identify”.

```java
public StateModel Identify(StateModel model, StateModel state) {
    if (model == null || state == null) {
        return null;
    }
    ArrayList<StateModel> matches = getTypeMatches(model, state);
    StateModel bestMatch = getBestMatch(model, matches);
    double level = getEqualityLevel(model, bestMatch);
    if (level > EQUALITY_THRESHOLD) {
        return bestMatch;
    } else {
        return null;
    }
}
```

Listing B.3: Part of the implementation of the heuristics-based GUI object recognition.

The “Identify”-method received the model of the targeted GUI object as well as the state of the AUT. First it identifies all the elements of the state which are of the same type as the model. Afterwards, it identifies the best match from the objects of the correct type and finally, it validates whether the best match is good enough in terms of an equality threshold compared to the equality level of the best match. The best match is found as the object with the largest equality level, which is calculated by comparison of position, size, content and child elements.

B.1.3.2 Command execution

In order to send user actions to AUT or receive its state, the TestClient sends commands through the TestExecutor class. The implementation of the “Send”-method in the TestExecutor is shown in Listing B.4.

```java
public void Send(ITestCommand command) {
    byte[] message = new byte[transportClient.GetPacketSize()];
    message[0] = (byte) testCommand.GetMessageType();
    message[1] = (byte) testCommand.GetMessageIdentifier();
    byte[] messageData = testCommand.GetMessageData().getBytes();
    for (int i = 2; i < transportClient.GetPacketSize(); i++) {
        if (messageData.length > (i - 2)) {
            message[i] = messageData[i - 2];
        } else {
            message[i] = 0x00;
        }
    }
}
```

Listing B.4: Part of the implementation of the command execution.
Target implementation

transactClient.Send(message);
}

Listing B.4: Implementation of the “Send”-method of the TestExecutor.

The methods of the TestExecutor implement the communication protocol in the host framework in
the sense that they create and interpret the messages sent and received from the transport object.
As seen in the “Send”-method, the Command Pattern [17] is utilized, as the command objects
contain their own information pieced together by the TestExecutor.

B.2. Target implementation

The implementation of the target prototype is based on the target design described in Section 4.4
and integrates with the TouchGFX framework version 3.0.0 described in Section 4.4.3. During the
development of the prototype framework, Mjølner Informatics A/S provided access to the source
code of TouchGFX, which enabled us to extend and alter the enabling GUI framework as needed.
The following sections present the details of the target prototype implementation. Section B.2.1
describes the development environments used for the simulated and physical environments and
Section B.2.2 describes the integration with the TouchGFX framework in detail.

B.2.1 Development environments

During the implementation of the target prototype two primary development environments have
been used: Microsoft Visual Studio 2010⁴ and IAR Embedded Workbench for ARM⁵ IDE. Visual
Studio utilizes the simulated environment while IAR Embedded Workbench utilizes the physical
device.

B.2.1.1 The simulated environment

The simulated environment refers to a simulator included in the TouchGFX distribution. A simu-
lator is provided for Windows and Linux and it enables an embedded TouchGFX application to be
executed on a laptop. The simulator relies on the SDL library⁶ in order to register user input and
render graphical output. The simulator eases the initial application development as the developer
does not need to continuously download the application to the device in order to verifying minor
changes to the GUI. Initially the prototype was developed for the simulator and later extended for
the physical device.
Figure B.3 shows the simulator running the AUT, described in Section 5.3.2.2, with the MCU
indicator enabled.

⁴http://www.visualstudio.com/downloads/download-visual-studio-vs#
DownloadFamilies_4
⁵http://www.iar.com/Products/IAR-Embedded-Workbench/ARM/
⁶http://www.libsdl.org/
Appendix B. Implementation of the Proof-of-Concept Prototype

Figure B.3: TouchGFX simulator running the AUT on Windows.

B.2.1.2 The device environment

The physical device refers to an embedded device with support for TouchGFX. In the development of the target prototype the *LPC4357 Developer’s Kit*\(^7\), from here on referred to as LPC4357DK, was provided by Mjølner Informatics A/S. Figure B.4 shows the LPC4357DK device running the AUT.

During the prototype development the LPC4357DK plays the role of the low-resource embedded device referred to throughout the thesis. Table B.1 shows a subset of the resources available on the LPC4357DK device.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cortex-M4/M0 microcontroller, running at up to 204 MHz.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Flash</td>
<td>1 MB internal programmable FLASH.</td>
</tr>
<tr>
<td>Data memory</td>
<td>135 KB internal SRAM memory.</td>
</tr>
<tr>
<td>Connectors and interfaces</td>
<td>Support for RS232 through USB-to-serial bridge on UART #0.</td>
</tr>
</tbody>
</table>

\(^7\)http://www.embeddedartists.com/products/kits/lpc4357_kit.php
The device environment

Figure B.4: Physical LPC4357DK device running the AUT.
Appendix B. Implementation of the Proof-of-Concept Prototype

FreeRTOS

The FreeRTOS OS\(^8\) is used as the OS for TouchGFX when executing the AUT on the LPC4357DK device. The FreeRTOS distribution provides an operating system for real time systems and supports a wide range of heterogeneous platforms.

FreeRTOS provides two main functionalities in relation to the TouchGFX execution: resource control and multi-threading facilities. Resource control describes the current resource consumption of the device and is configured in a platform-specific FreeRTOSConfig header file. This configuration file is used by the FreeRTOS kernel during device startup. Listing B.5 shows an excerpt of the FreeRTOSConfig for the LPC4357DK device.

```c
#define configCPU_CLOCK_HZ ( SystemCoreClock )
#define configTICK_RATE_HZ ( (portTickType) 1000 )
#define configMAX_PRIORITIES ( (unsigned portBASE_TYPE) 5 )
#define configMINIMAL_STACK_SIZE ( (unsigned short) 64 )
#define configTOTAL_HEAP_SIZE ( (size_t) ( 5500 ) )
#define configCHECK_FOR_STACK_OVERFLOW 2
#define configUSE_MALLOC_FAILED_HOOK 1
```

Listing B.5: Subset of the definitions in the FreeRTOSConfig header file

While the FreeRTOS configuration specifies many settings relevant to the application execution, only specific settings have been altered in order to allow the TestGFX framework to run on the device. The `configTOTAL_HEAP_SIZE` definition specifies a maximum number of `words` to be supplied for stack allocation within specified tasks. A word constitutes 4 bytes of memory. A task describes a method to be run by the FreeRTOS threading library which is typically non-terminating. Upon creating tasks the stack allocation is checked by facilities of the FreeRTOS distribution. Attempting to exceed the specified threshold will result in a STACK_OVERFLOW error. In order to ensure that the test framework prototype could run alongside the TouchGFX framework additional stack memory was supplied.

The threading facilities supplied by FreeRTOS are based on single-core executions of tasks via scheduling and prioritizing. As such, task execution is not truly parallel. Initial attempts of parallel execution of the AUT and test framework proved to produce indeterministic behavior. Through further inspection of the TouchGFX framework, the HAL was identified as a more appropriate context for execution.

B.2.2 TouchGFX integration

The target design makes some assumptions relative to the enabling GUI framework TouchGFX. While some of the assumptions hold true, others had to be introduced in the development of the prototype. This section describes the extensions made to the TouchGFX framework and how they relate to the features implemented for the target prototype.

The TouchGFX framework, described in Section 2.4, provides several abstractions for interacting with the AUT. However, as the TouchGFX framework was not initially designed to support integration with a testing framework some extensions had to be introduced. These extensions consist primarily of access to scoped attributes. As the extensions introduce minor overhead to the code size and memory utilization of the TouchGFX distribution, the impact is considered negligible.

\(^8\)http://www.freertos.org/
The initialization and execution processes, described in Section 5.3.1.2, specify how the target framework is initialized alongside the AUT. On system startup, the test framework replaces the concrete HAL of the AUT with a specialized version. Figure B.5 illustrates the preprocessor definition inheritance scheme used by the HALTEST class.

As previously mentioned the concrete HAL object is replaced with a HALTEST equivalent enabled for test automation. By allowing the HALTEST class to inherit functionality from the concrete HAL class, the base class itself is left unchanged. Listing B.6 shows the identification of the concrete HAL to inherit from.

```
#ifdef SIMULATOR
# include <hal_sdl\HALSDL.hpp>
typedef HALSDL CONCRETE_HAL;
#else
# include <hal_nxp/HALNXP.hpp>
typedef touchgfx::HALNXP CONCRETE_HAL;
#endif

class HALTEST : public CONCRETE_HAL
{
public:
  HALTEST(HALTESTConfig& config, DMA_Interface& dma, uint16_t width, uint16_t height, DisplayRotation DisplayRotation = rotate0)
      : CONCRETE_HAL(dma, width, height), _config(config) {}

  void handleEvents(); // Override CONCRETE_HAL.handleEvents().

  void initializeTestAgent(AAL* aal, StateClient* stateClient, TransportInterface* transport)
```

Figure B.5: TestGFX HALTEST inheritance scheme.
Appendix B. Implementation of the Proof-of-Concept Prototype

The implementation header shows how the base class of HALTEST is changed at compile time based on preprocessor definitions. A HALTESTConfig structure is used to specify configurations for the framework execution, e.g., the frequency with which the “testProcedure”-method is called inside the overridden “handleEvents”-method. In a prototype setting this is an acceptable approach. While the simulator presents a non-standard “handleEvent” implementation that of the TouchGFX-enabled devices is identical to this. As such, switching from one target device to another only constitutes replacing the concrete HAL.

B.2.2.2 Event invocation

Event invocation, described in Section 4.4.3.2, concerns low-level manipulation of the AUT by pushing touch events onto the Application instance. To introduce event invocation to the target prototype no extensions to the TouchGFX framework were made. Each event command implementation includes the creation of a single Event object related to the corresponding event type, e.g., a ClickCommand will create a ClickEvent. The ClickEvent is subsequently pushed onto the Application instance, which is globally accessible through the TouchGFX Application class described in Section 2.4.2.5.

In order to control event invocation, which was found necessary in order to achieve deterministic test execution on the target, a FIFO queue was introduced to the “testProcedure”. Listing B.7 shows the details of this method.

```cpp
void TestAgent::testProcedure()
{
    if(initialized == false) { return; } // error.

    if(cmdCount == 0)
    {
        cmdCount = _protocolClient->receiveCommand(cmdArray);
        cmdIndex = 0;
    }

    if(cmdCount > 0)
    {
        switch( cmdArray[cmdIndex]->getType())
        {
```
B.2.2.3 State acquisition

The process of state acquisition, described in Section 4.4.3.3, concerns extraction of the state information from visual components in the current view, and structuring it in an appropriate format for subsequent transfer to the host. In order to perform state extraction it was necessary to extend the TouchGFX framework in the Screen, the Drawable class and the derived Drawable classes.

Screen
In order to enable the TouchGFXStateClient, described in Section 5.3.1.2, to extract state data from the current view, the root Container has to be accessed. Through the Application instance it was possible to access the current Screen object. However, the root Container of this screen was scoped. A simple “getContainer”-method was added in order to access this attribute.

Drawable
While performing state acquisition through the root Container a “forEachChild”-method is used to traverse the Drawable children. While traversing the child objects each Drawable is passed to the DrawableInterpreter, described in Section 5.3.1.2. While each Drawable object contains basic state information, i.e., position, width, height and visibility, they also contain important content related to their type. As the “forEachChild”-method does not support carrying over type information from the Drawable object, another approach to identifying the type was necessary. Through Runtime Type Identification (RTTI) it was possible to assess the type of Drawables on the simulator. However, while RTTI is convenient certain issues appeared on the physical device. First, when compiling to a device, a subset of C++, which does not include the RTTI language feature, is utilized. While it is possible to enable RTTI, the calls to the virtual table are generally considered too resource expensive. Also, due to the content of the table being string-based representations of class names, this approach was deemed too inaccurate and unreliable. A custom type identification scheme was thus introduced to the Drawable hierarchy by adding a pure
Appendix B. Implementation of the Proof-of-Concept Prototype

virtual “getType”-method in the Drawable base class. Each derived Drawable is thus required to implement this method removing the need for RTTI by employing static type checking.

B.3. Protocol implementation

This section describes the details of the protocol implementation. This description includes the packets used and how they are translated by the target and host respectively.

B.3.1 Packet structure

The packets used in the protocol all follow the same 24 byte structure as seen in Figure B.6. A Type-byte indicates the type of the packet, i.e., EVENT, STATE or RESET. The Id-byte is used to indicate the sub-type of a packet. This is mainly used in the event command and state packets described in sections B.3.2.1 and B.3.3 respectively. The remaining 22 byte payload is used to hold data specific to the packet type and sub-type.

Figure B.6: Protocol packet structure.

B.3.2 Command packets

TestGFX on the host-side uses four different packets for sending four different commands to TestGFX on the target-side. Command packets contain a structure recognized by the target framework and is interpreted into specific actions to be performed on the AUT. The command packets used in the prototype are divided into two main categories: event command packets and control command packets.

B.3.2.1 Event commands packets

Event command packets are used to communicate event instructions from the host to the target. Two event command packets are supported by the prototype framework; a click command packet and a drag command packet.

Click command packet
A click command packet, shown in Figure B.7, contains three parameters separated by a “;” delimiter. An X and Y coordinate pair represent the screen position to which the click is applied. A Z value describes the force with which it is applied. The format specified in Figure B.7 represents the amount of characters available to the host to specify the X, Y and Z values used for the click command.

Drag command packet
A drag command packet, shown in Figure B.8, contains four parameters separated by a “;” de-
limiter. The first X and Y pair describes the starting position of the drag event. The second pair describes the end position of the drag event.

The format specified in Figure B.8 represents the amount of characters available to the host to specify the two X and Y pairs used for the drag command.

B.3.2.2 Control commands packets

Control command packets are used to query and control the state of the AUT. Two control command packets are supported by the prototype framework; a state command packet and a reset command packet.

State command packet
A state command packet, shown in Figure B.9, contains a single indicative state byte. When received, the target prototype will extract the state and return it to the host as described in Section 4.4.4. The packets used to transfer the state back to the host are described in Section B.3.3.

Reset command packet
A reset command packet, shown in Figure B.10, contains a single indicative reset byte. When received the target prototype will reset the AUT through the AAL interface as described in Section 4.4.4.
Appendix B. Implementation of the Proof-of-Concept Prototype

B.3.3 State packets

In order to support the streaming of state data, which may include up to 500 bytes of data depending on the complexity of the GUI, three different types of packets are used: state start, state stream and state end, shown in figures B.11, B.12 and B.13 respectively. Each of the packets contains a state data payload. Prior to transferring the state information it is compressed based on a byte scheme described in Section B.3.4.

B.3.4 Compression scheme

A compression scheme is applied to the state data prior to sending it to the host. Upon receiving the compressed state data the host will expand it with a reverse implementation of the same compression protocol. Figure B.14 shows an example of how the compressed state sent from the target is expanded on the host.
B.4. User manual

This section presents how to install the prototype and its dependencies as well as how to run the prototype. The prototype source code is provided with this thesis on the attached DVD disc. The reader is informed that the prototype uses the proprietary TouchGFX GUI framework version 3.0.0 distributed by Mjølner Informatics A/S. A trial license can be requested from the product page.9

B.4.1 Installation

This section provides an overview of the tools necessary in order to run the Application Under Test and TestGFX prototype.

Robot Framework The prototype uses the Robot Framework, which can be downloaded from the project website.10 Instructions for installing the dependencies of the tool, i.e., Python and Jython, are provided in the Quick Start Guide.11

Eclipse IDE In order to develop the Java test libraries associated with the prototype and the Robot Framework, the Eclipse IDE has been used. The newest version of Eclipse can be downloaded from the product web site download section.12 In addition, a plugin for Robot Framework, RobotFramework-EclipseIDE, can be downloaded from its web site.13 This plugin has been used in the development of the test suite for the prototype as it provides syntax highlighting and code completion.

9http://mjolner.dk/oursolutions/touchgfx/
10http://robotframework.org/
11https://github.com/NitorCreations/RobotFramework-EclipseIDE
13http://www.eclipse.org/downloads/
Appendix B. Implementation of the Proof-of-Concept Prototype

**Free Virtual Serial Ports** In order to connect the host to the target prototypes running in a simulator, the Free Virtual Serial Ports (FVSP) tool has been used. This can be downloaded from its web site\(^{14}\).

**TouchGFX** In order to compile the TestGFX target prototype and the Application Under Test, the TouchGFX distribution version 3.0.0 has to be installed. The reader is referred to Mjølner Informatics A/S for acquiring this product.

**Microsoft Visual Studio 2010 IDE** In order to edit and compile the TestGFX target prototype and the Application Under Test for execution in the TouchGFX simulator for Windows, the Microsoft Visual Studio (MSVS) 2010 IDE has been used. A free express version can be downloaded from its product page\(^{15}\).

**IAR Embedded Workbench for ARM IDE** In order to compile and download the TestGFX target prototype and the Application Under Test to the LPC4357DK device the IAR Embedded Workbench for ARM has been used. A free 30 day trial can be downloaded from the product page\(^{16}\).

### B.4.2 Running the prototype

This section describes how to run the Application Under Test along with the TestGFX prototype framework. Installation steps described in Section B.4.1 are expected to have been carried out successfully before continuing.

Section B.4.2.1 describes how to set up the appropriate virtual serial communication ports used for the simulated environment. Section B.4.2.2 describes how to run the Application Under Test in the simulator and Section B.4.2.3 describes how to run it in on the LPC4357DK device. Finally Section B.4.2.4 describes how to execute the test suite on the Application Under Test.

#### B.4.2.1 Virtual serial ports

Prior to running the target prototype in the simulator appropriate virtual serial ports must be configured. Using the FVSP tool a virtual bridge has to be created with endpoints matching the COM ports utilized by the test framework and the Application Under Test. Figure B.15 shows the FVSP tool configured for the COM ports “COM7” and “COM8”.

When the virtual serial communication ports have been set up the Application Under Test can be run in the simulated environment.

#### B.4.2.2 Simulator

To run the Application Under Test in the simulator, the MSVS target prototype solution “TargetPrototype.sln” is opened. Figure B.16 shows the prototype solution including the Application Under Test and the target TestGFX projects.

When the editor has loaded it is verified that a preprocessor definition “TESTGFX” is present in the project configuration. This is done by right-clicking the project and selecting “Properties”. Navigate to “Configuration Properties -> C/C++ -> Preprocessor” and inspect the “Preprocessor

\(^{14}\)http://download.cnet.com/Free-Virtual-Serial-Ports-Emulator/3000-2206_4-10836189.html

\(^{15}\)http://www.visualstudio.com/downloads/download-visual-studio-vs#

\(^{16}\)http://www.iar.com/Products/IAR-Embedded-Workbench/ARM/
Figure B.15: Virtual bridge setup in the Free Virtual Serial Ports tool.

Figure B.16: The Application Under Test and target TestGFX opened in MSVS 2010.
Appendix B. Implementation of the Proof-of-Concept Prototype

definitions”-field. If the TESTGFX definition is not found, it must be added for the Application Under Test to be test-enabled.

Upon verifying the aforementioned settings, the “ApplicationUnderTest” project is started.

**B.4.2.3 Device**

In order to run the target prototype on the LPC4357DK device, a JTAG compiler cable is necessary in order to download the compiled executable. Upon connecting the laptop and LPC4357DK device, the application can be downloaded to the device using the IAR Embedded Workbench for ARM. Figure B.17 shows the Application Under Test project in the IAR Embedded Workbench IDE. Note that the TestGFX and TouchGFX libraries are statically linked to the project. A preprocessor definition “TESTGFX” is defined in the project configuration.

![Figure B.17: The Application Under Test and target TestGFX opened in IAR Embedded Workbench for ARM.](image)

After the application executable has been downloaded it may be necessary to reset the device. When restarted, the application should show on the LPC4357DK touchscreen.
B.4.2.4 Robot Framework

After initializing the Application Under Test on either the simulator or the LPC4357DK device, the test suite can be executed through the Robot Framework and the host prototype. The host solution, including the test library, the TestGFX prototype and the test suite, may be opened in the Eclipse IDE as shown in Figure B.1.

Before the test suite may be executed, the test library JAR file, and other JAR files referenced by the TestGFX prototype, has to be registered in the Robot Framework for it to be able to find the keywords implemented by the library. In order to ease this process a Windows batch script was written. The batch script, shown in Listing B.8, takes a directory as its only parameter and will register any JAR files present in the lib directory within.

```bash
@echo off
set CP=. 
for %%j in ("%~dp0\lib\*.jar") do {
    call :set_cp %%j
    echo %%j
}
set CLASSPATH=%CP%

echo "Test suite is configured..."
goto :eof
:: Helper for setting variables inside a for loop
:set_cp
    set CP=%CP%;%1
goto :eof
```

Listing B.8: Windows batch script for registering JAR files in the Robot Framework.

As the test library is developed in Java, the test suite must be executed through Jython, the Java port of Python. This is done by utilizing the Command Line Interface (CLI) for Jython with the absolute path of the test suite as the only parameter. In the console the execution of the test suite provides the test report shown in Figure B.18. Links are provided to the detailed HTML versions of the test report.

As shown in Figure B.1, the test suite is set to use the COM port named “COM3” which corresponds to the LPC4357DK device in the test environment used during framework evaluation. In order to execute the test suite on the TouchGFX simulator, this may be changed to “COM8” instead.
Appendix B. Implementation of the Proof-of-Concept Prototype

<table>
<thead>
<tr>
<th>PrototypeTestSuite</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment and decrement counter</td>
<td>PASS</td>
</tr>
<tr>
<td>Transition between screens</td>
<td>PASS</td>
</tr>
<tr>
<td>Increment counter then transition and decrement counter</td>
<td>FAIL</td>
</tr>
<tr>
<td>Failed: Expected 0, but was 1</td>
<td></td>
</tr>
<tr>
<td>Transition to drag screen and drag object</td>
<td>PASS</td>
</tr>
<tr>
<td>Drag object then transition and validate new position</td>
<td>FAIL</td>
</tr>
<tr>
<td>Failed: Expected (317,109), but was (52,109)</td>
<td></td>
</tr>
<tr>
<td>PrototypeTestSuite</td>
<td>FAIL</td>
</tr>
<tr>
<td>5 critical tests, 3 passed, 2 failed</td>
<td></td>
</tr>
<tr>
<td>5 tests total, 3 passed, 2 failed</td>
<td></td>
</tr>
</tbody>
</table>

Output: D:\ProgramFiles\eclipse-standard-kepler-SR2-win32\eclipse\output.xml
Log: D:\ProgramFiles\eclipse-standard-kepler-SR2-win32\eclipse\log.html
Report: D:\ProgramFiles\eclipse-standard-kepler-SR2-win32\eclipse\report.html

Figure B.18: Console test report generated by execution of the test suite through the prototype framework.
APPENDIX C

Results of the Prototype Evaluation

This appendix provides a detailed description of the results of the evaluation presented in Chapter 5. This includes a description of the results as well as the details of how measurements have been made.

C.1. Functional suitability

This quality property represents the degree to which the GUI testing framework provides a set of functions appropriate for the expected tasks and user objectives and provides the specified results with the needed degree of precision and accuracy.

1.1 The framework shall support features expected of an automated GUI testing framework.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1</td>
<td>The framework supports specification of user scenarios on GUI applications.</td>
<td>✓</td>
</tr>
<tr>
<td>1.1.2</td>
<td>The framework supports assertion of the state of a GUI application.</td>
<td>✓</td>
</tr>
<tr>
<td>1.1.3</td>
<td>The framework provides appropriate reporting of the execution of an automated test.</td>
<td>✓</td>
</tr>
<tr>
<td>1.1.4</td>
<td>The framework is able to automate user actions on GUI applications.</td>
<td>✓</td>
</tr>
<tr>
<td>1.1.5</td>
<td>The framework is able to retrieve the state of a GUI application for validation.</td>
<td>✓</td>
</tr>
<tr>
<td>1.1.6</td>
<td>The framework is able to reset a GUI application to the initial state of a testing session.</td>
<td>✓</td>
</tr>
</tbody>
</table>

All measurements for Requirement 1.1 were taken by successful specification and execution of the test suite described in Section 5.3.2.3 on the Application Under Test described in Section 5.3.2.2. Test execution was performed by means of the prototype test framework described in Section 5.3.1.

1.2 The framework shall be able to detect and report defects in a GUI application.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.1</td>
<td>The framework is able to detect and report the existence of at least two defects.</td>
<td>✓</td>
</tr>
</tbody>
</table>

The measurement for Requirement 1.2 was taken by execution of the test suite described in Section 5.3.2.3 on the Application Under Test described in Section 5.3.2.2. Test execution was performed by means of the prototype test framework described in Section 5.3.1. Figure C.1 shows
Appendix C. Results of the Prototype Evaluation

the test report on which the result of the measurement was based.

![Test Details Table]

Figure C.1: Test report for the execution of the test suite as part of the framework evaluation.

1.3 The framework shall support easy maintenance of test specifications.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.1</td>
<td>The framework supports reuse of test scenarios in terms of sequences of user actions and other commands.</td>
<td>✓</td>
</tr>
<tr>
<td>1.3.2</td>
<td>The reference to GUI objects in the test specification is abstracted from the actual state of the AUT.</td>
<td>✓</td>
</tr>
<tr>
<td>1.3.3</td>
<td>The framework is able to recognize GUI objects after insignificant changes to the GUI, i.e., changes to position or size of the GUI object.</td>
<td>✓</td>
</tr>
</tbody>
</table>

The measurements for Requirement 1.3 were taken by successful specification of the test suite described in Section 5.3.2.3 and validation of the AUT described in Section 5.3.2.2. The result of Measurement 1.3.3 is based on the test case titled “Transition to drag screen and drag object”. As the draggable object is snapped into place, the position used in the assertion is not the exact position of the object, but the object is successfully identified nonetheless.

C.2. Performance efficiency

This property represents the degree to which the test framework meets requirements for performance relative to the amount of resources used under specified conditions.

2.1 The framework shall have low resource utilization.

In order to assess the code size of the test framework on the target the IAR Embedded Workbench for ARM\(^1\) was used.

\(^1\)http://www.iar.com/Products/IAR-Embedded-Workbench/ARM/
Performance efficiency

<table>
<thead>
<tr>
<th>2.1.1</th>
<th>The total code size of the framework is less than 20 KB.</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.2</td>
<td>The runtime memory utilization of the framework is less than 10 KB.</td>
<td>✓</td>
</tr>
<tr>
<td>2.1.3</td>
<td>The maximum load on the processor during automated testing by the framework does not exceed the maximum load occurring during manual testing by more than 5% of the total load possible.</td>
<td>✓</td>
</tr>
</tbody>
</table>

The IAR Embedded Workbench for ARM is a complete IDE capable of producing and compiling highly optimized code for ARM-based embedded devices. With the IAR Workbench, the AUT was compiled to the LPC4357DK device, first with and later without the test framework enabled. Upon compilation a linker-generated memory map is produced. The memory map provides details of the compilation process in terms of the memory use for each file input as well as the nature of the memory use. Three types of memory use are shown: read-only code memory (typically FLASH or EEPROM), read-only data (typically external FLASH) and read-write data (RAM).

Listings C.1 and C.2 show the compilation details for the AUT with and without the test framework respectively.

```plaintext
Module  ro code  ro data  rw data
--------  -------  -------  -------
C:\projects\touchgfx\...\LPC4357DevKit\IAR\Debug\Obj: [1]
  main.o  826  172  6  897
  ...
  ----------------------------------------------------------
  Total:  42 312  808 167  18 042

command line: [2] ...

dl7M_tln.a: [3]
  dlmalloc.o  5  312  496
  ...
  ----------------------------------------------------------
  Total:  9 726  532

dlpp7M_t1_nc.a: [4] ...
m7M_tlv.a: [5] ...
rt7M_t1.a: [6] ...
shb_l.a: [7] ...
testgfx.a: [8]
  ...
  ----------------------------------------------------------
  Total:  6 086  376  12

touchgfx_core_testgfx.a: [9]
  Application.o  2 028  412  12
  HAL.o  1 032  14
  ...
```
### Listing C.1: Module summary for the compilation of the AUT with the test framework.

<table>
<thead>
<tr>
<th>Module</th>
<th>ro code</th>
<th>ro data</th>
<th>rw data</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\projects\touchgfx...\LPC4357DevKit\IAR\Release\Obj: [1] main.o</td>
<td>868</td>
<td>160</td>
<td>5 853</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>42 438</td>
<td>808</td>
<td>138 169</td>
</tr>
<tr>
<td>command line: [2] ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dl7M_tln.a: [3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>3 876</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>dlpp7M_t1_nc.a: [4] ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m7M_tlv.a: [5] ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt7M_t1.a: [6] ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shb_l.a: [7] ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>touchgfx_core.a: [8]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application.o</td>
<td>2 128</td>
<td>412</td>
<td>12</td>
</tr>
<tr>
<td>HAL.o</td>
<td>1 172</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>16 618</td>
<td>1 812</td>
<td>109</td>
</tr>
<tr>
<td>touchgfx_hal.a: [9]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALNXP.o</td>
<td>560</td>
<td>142</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>1 194</td>
<td>206</td>
<td>4 412</td>
</tr>
</tbody>
</table>
Performance efficiency

<table>
<thead>
<tr>
<th></th>
<th>Gaps</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linker created</td>
<td></td>
<td>49</td>
<td>10 496</td>
</tr>
<tr>
<td>Grand Total:</td>
<td>67 895</td>
<td>810</td>
<td>206</td>
</tr>
</tbody>
</table>

Listing C.2: Module summary for the compilation of the AUT without the test framework.

Measurement 2.1.1
In each of the presented listings a “Grand Total” is presented for the read-only code, read-only data and read-write data memory. Comparing the grand total of the read-only code memory used in both compilations reveals an overhead of \(80,460 - 67,895 = 12,565\) \(\text{bytes}\) or \(~12.6\) KB. In accordance with the threshold specified in Measurement 2.1.1 the framework is validated to be below 20 KB in code size.

Noted as future work in Section 6.3, the major constituents of the test framework overhead are the TestGFX core assembly, “testgfx.a”, and a utility assembly, “dl7M_tln.a”, as seen in Listing C.1. The TestGFX core produces only \(~6\) KB overhead. The utility assembly however, includes an object file, “dl_malloc.o”, which represents the Doug Lea memory allocator driver\(^2\). The overhead produced by this single object file is \(~5.3\) KB. The dependency to this assembly is made by the test framework and future work should remove this dependency.

Measurement 2.1.2
In order to assess memory consumption of the test framework and validate Measurement 2.1.2, the read-write data memory consumption is inspected. Comparing the grand total of both compilations reveals an overhead of \(41,805 - 32,049 = 9,9756\) \(\text{bytes}\) or \(~9.8\) KB. In accordance with the threshold specified in Measurement 2.1.2 the memory consumption of the framework is validated to be below 10 KB.

Measurement 2.1.3
In order to assess the impact on the processor and validate Measurement 2.1.3, the MCU load is monitored at runtime while executing tests on the LPC4357DK device. MCU load describes the percentage of the time the MCU processor is actively carrying out computations. By adding a custom visual component, the MCU load indicator, the maximum MCU load can be read from the AUT at runtime. The MCU load indicator is implemented through an algorithm measuring the time the FreeRTOS thread manager spends executing the “IDLE” task. The “IDLE” task represents the lowest priority thread executed in FreeRTOS. It is active whenever no other task claims the processor. The implementation of this benchmarking feature has been provided by Mjølner Informatics A/S. Figure C.2 shows the button screen of the AUT with the MCU load indicator in the top left corner.

The maximum MCU load was noted while automating the test suite presented in Section 5.3.2.3 on the LPC4375DK device. Similarly the maximum MCU load was noted while running the test suite manually. The maximum MCU load on the target when performing automated testing was noted as 15%. In comparison the load while performing manual testing was noted as 13%.

2.2 The framework shall be able to perform GUI automation within reasonable time.
The measurements for Requirement 2.2 were taken by sending specific commands through the prototype described in Section 5.3.1 to the AUT described in Section 5.3.2.2. The time it took

\(^2\)http://g.oswego.edu/dl/html/malloc.html

123
Appendix C. Results of the Prototype Evaluation

Figure C.2: The button screen of the AUT with an MCU load indicator on the LPC4375DK device.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1</td>
<td>The framework is able to automate a user action on the GUI of the AUT within 2 seconds.</td>
<td>X</td>
</tr>
<tr>
<td>2.2.2</td>
<td>The framework is able to retrieve the state of the GUI of the AUT within 2 seconds.</td>
<td>✓</td>
</tr>
</tbody>
</table>

from a command was sent and until a visual confirmation could be made was measured using a stopwatch.

Measurement 2.2.1 was taken by sending a click command from the host to the AUT. A total of ten measurements were taken from the time the command was sent until the time the GUI had completed the click animation. The mean value of the ten measurements was 2.43 seconds.

Measurement 2.2.2 was taken by sending a state command from the host to the AUT. A total of ten measurements were taken from the time the command was sent until the time the test library on the host received the state of the GUI. The mean value of the ten measurements was 1.68 seconds.

C.3. Compatibility

This quality property represents the degree to which the GUI testing framework can interoperate with other products, i.e., TouchGFX and the AUT, and perform its required functions while sharing the target platform with the AUT.

3.1 The framework shall execute GUI events reliably and in deterministic order.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1</td>
<td>Multiple executions of commands in quick succession are performed by the AUT in the order they are sent.</td>
<td>✓</td>
</tr>
</tbody>
</table>
Maintainability

3.2 While deployed in the target environment, the framework shall not affect the execution of the AUT.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>The automated execution of the test suite by the framework provides similar results as the manual execution of the same test suite.</td>
<td>✓</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The automated execution of the test suite by the framework is visually similar as the manual execution.</td>
<td>✓</td>
</tr>
</tbody>
</table>

The measurements for requirements 3.1 and 3.2 were taken by the successful execution of the test suite described in Section 5.3.2.3 on the AUT described in Section 5.3.2.2, running on the LPC4357DK device. The result of the measurement for Requirement 3.1 is based on the test case titled “Increment and decrement counter” in which multiple click commands are sent in immediate succession.

C.4. Maintainability

This quality property represents the degree to which the GUI testing framework can be effectively and efficiently modified by the intended personnel.

4.1 The classes of the framework shall adhere to the principles of SOLID design [32].

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1</td>
<td>Classes of the framework have only a single reason to change, or source of change, in accordance with the Single Responsibility Principle.</td>
<td>✓</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Classes of the framework can be extended without alteration to the existing source code, in accordance with the Open/Closed Principle.</td>
<td>✓</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Classes of the framework may substitute their super-types without incurring changes to the overall behavior of the framework, in accordance with the Liskov Substitution Principle.</td>
<td>✓</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Internal as well as external parties relying on classes of the framework do not depend on methods they do not use, in accordance with the Interface Segregation Principle.</td>
<td>X</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Internal as well as external parties relying on functionality of the framework depend on abstractions rather than implementations of that functionality, in accordance with the Dependency Inversion Principle.</td>
<td>✓</td>
</tr>
</tbody>
</table>

The measurements for Requirement 4.1 were verified by code review of the host and target prototype implementations. Measurement 4.1.4 fails due to the command abstraction in the host implementation. While the use of commands is very different in the test library and TestGFX, they share the same interface.
Appendix C. Results of the Prototype Evaluation

C.5. **Transferability**

This quality property represents the degree to which the GUI testing framework can be transferred from one environment to another.

5.1 **The framework shall be easy to port from one target environment to the next.**

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>No changes are required to the host framework in order to support a new target environment.</td>
<td>✓</td>
</tr>
<tr>
<td>5.1.2</td>
<td>The changes to the target framework necessary in order to support a new target environment are limited to low-level device drivers.</td>
<td>✓</td>
</tr>
<tr>
<td>5.1.3</td>
<td>The target framework is applicable to a new AUT without any changes to the framework required.</td>
<td>X</td>
</tr>
</tbody>
</table>

The measurements for Requirement 5.1 were taken based on code review of the host and target prototype implementations as well as execution of the target prototype on different target environments. The target environments used were the LPC4357DK device and the TouchGFX simulator. Measurement 5.1.3 fails as the target prototype requires application-specific implementation of the FrontendAAL, described in Section 4.4.2.1.
Overview of the Thesis Work

This appendix presents an overview of the thesis work in terms of the work distribution among the authors of the thesis as well as an outline of the progression of the thesis work.

D.1. Work distribution

Table D.1 shows the distribution of the work carried out in this thesis. The areas of work are divided into Chapters 1 - 6 and the partitions of the constructed prototype, host and target. The granularity of workload has been divided into two categories: First author (F) and Second author (S).

<table>
<thead>
<tr>
<th>Area</th>
<th>Jesper Gaarsdal</th>
<th>Jacob Emborg Sønderskov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>F</td>
<td>S</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>S</td>
<td>F</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Prototype (host)</td>
<td>F</td>
<td>S</td>
</tr>
<tr>
<td>Prototype (target)</td>
<td>S</td>
<td>F</td>
</tr>
</tbody>
</table>

D.2. Progression of the thesis work

Figure D.1 shows a rough outline of the progression of the thesis work. As it appears from the timeline, much of the work was completed late in the process. Part of the explanation for this behavior is in the fact that this was the authors’ first experience with a large body of scientific work. Research and especially writing a thesis was much more time consuming than expected. Furthermore, obstacles in the enabling GUI framework, TouchGFX, delayed the completion of the target prototype which in turn delayed the experimentation process.
Appendix D. Overview of the Thesis Work

Figure D.1: Outline of the progression of the thesis work.