

TAV - Report

Project: Luminosus

Johannes Schneider and Tim Henning

February 6, 2018

Contents

0	Application Selection	4
0.1	Introduction	4
0.2	Learning from failures	5
0.3	Five Basic V&V Questions	7
1	Analysis	9
1.1	Human-based Static Analysis	9
1.1.1	Peer Review	9
1.1.2	Tools	10
1.2	Automated Static Analysis	11
1.2.1	Clang Static Analyzer	11
1.2.2	Clang-Tidy	12
1.2.3	CppCheck	13
1.2.4	Polyspace	13
1.2.5	CppLint	14
1.3	Interaction between Human-Based and Automated SA	14
2	Testing	16
2.1	Kick-off Tasks	16
2.1.1	Example Test Set	16
2.1.2	Type of Existing Test Set	18
2.1.3	Different Scopes of Test Case Execution	18
2.2	Test Case Development	20
2.2.1	Input Domain Modeling	20
2.2.2	Logic Expressions	22
2.2.3	Control Flow Graph	23
2.3	Automated Unit Tests	25
2.3.1	Google Test	25
2.3.2	Qt Test	26
2.3.3	Code Coverage Calculation	30
2.3.4	ASA and Testing	30
2.4	Evaluation	31
2.4.1	Bugs Found	31
2.4.2	Performance Regression Tests	31

3	Verification	33
3.1	Test Case Generation	33
3.1.1	KLEE	33
3.1.2	Visual Studio IntelliTest	35
3.2	ASA without Optimistic Inaccuray	36
3.2.1	Polyspace	36
3.3	Theorem Proving	38

0 Application Selection

0.1 Introduction

Modern lighting desks used to control the stage technique in large theaters and concert halls are complex embedded system, often running a proprietary software on a Linux or Windows system in combination with two or more multitouch screens and custom hardware like faders and encoders. They are connected to all the systems in an event location like house- and stage lights, projectors, hoists and sometimes even the sound equipment. For communication between the components the industry recently started to use standardized network protocols.

Extending the functionality of such a lighting desk can be a difficult and time consuming task. To make it easier to integrate and test new features, the idea was to create a separate modular software platform that is connected via network to the lighting console. As a solution for this "Luminosus" was developed.

It is designed for lighting consoles from the market leader Electronic Theatre Controls (ETC) and uses the Open Sound Control (OSC)¹ protocol to control them. The user interface consists of modular function blocks that can be freely moved and connected. An example can be seen in fig. 0.1.

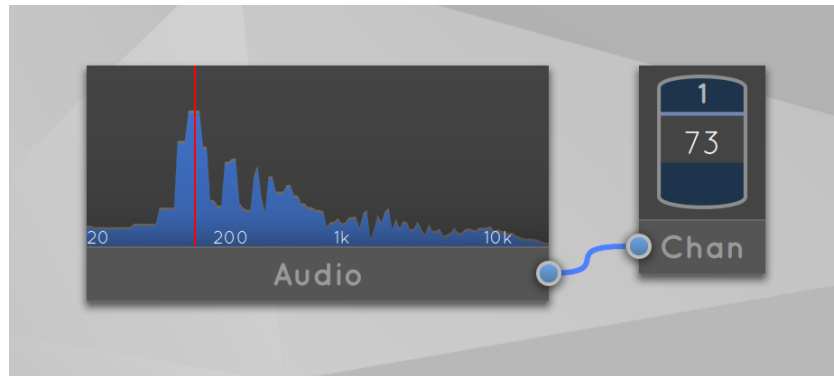


Figure 0.1: Luminosus Example: Controlling the brightness of channel 73 by the current audio level

The main project development was done by Tim Henning while writing his bachelor thesis. He is the only developer of the project until now. For this V&V project Johannes Schneider joined the project as a developer and tester.

¹see *The Open Sound Control 1.0 Specification* by Matt Wright

The programming language used is C++ 11 with syntax additions by Qt. The user interface part uses the declarative language QML² that includes function definitions in JavaScript.

The application is intended to be used by end users. It runs as a desktop application on Windows and MacOS and as an app on Android and iOS.

The existing code base consists of approximately 40k LOC (C++ source and header files + QML files). The code can be found on GitHub³.

Available artifacts are the source code, documentation, changelog⁴, issue tracker⁵, manual⁶ and in addition the Bachelor thesis *Entwicklung einer modularen Benutzeroberfläche als zusätzliche Bedieneinheit einer Lichtkonsole* including a requirement analysis and a discussion of the architectural decisions.

No specific development paradigm was used.

The current V&V status is that only manual tests are performed (adding all available function blocks as a kind of *smoke test* is available in the GUI) and static code analysis using Clang is provided by the IDE. Verification was not done yet.

0.2 Learning from failures

The following section is about learning from failures, which occurred over the course of developing the software.

Issue #2 Connection to Eos stopped working: Since the software is basically an extension for an existing hardware device, the so-called lighting console, the most important component is the network connection to the said hardware. Should this component fail to execute its duties the entire software would become useless. That said, this exact scenario appeared for a user during his show. The connection to his lighting console was interrupted for yet unknown reasons. Furthermore, the user was not able to pick up the lost connection even after restarting his entire setup, including multiple hardware components.

Although the fault is still undetected, it can be categorized as complexity related. This is, because the interaction of different devices in various environments is very hard to predict and even harder to test upfront. Reasons for this failure range from outdated driver software for the said hardware components over defective hardware up until a fault within the networking component of Luminosus.

Issue #3 Faders not updated: The Eos lighting console offers a set of faders to control the light intensity on different devices. But since this set is very limited in the amount of

² sometimes referred to as 'Qt Modeling Language', see <http://doc.qt.io/qt-5/qtqml-index.html>

³<https://github.com/ETCLabs/LuminosusEosEdition>

⁴<https://github.com/ETCLabs/LuminosusEosEdition/blob/master/doc/Changelog.txt>

⁵<https://github.com/ETCLabs/LuminosusEosEdition/issues>

⁶https://github.com/ETCLabs/LuminosusEosEdition/blob/master/doc/Manual_en.pdf

supported devices, Eos additionally includes virtual pages of faders. Thus, the effective amount of hardware one can manage using the Eos lighting console increases immensely.

Luminosus also offers the feature of switching pages of faders.

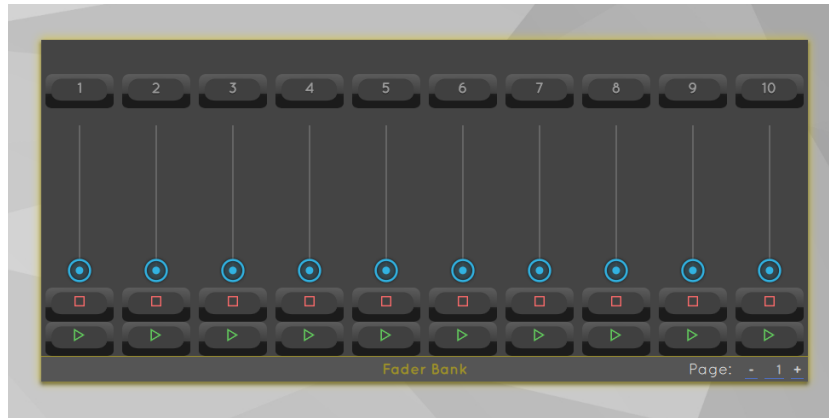


Figure 0.2: Luminosus Fader Bank: Visualization of the Eos faders including virtual pages

However, when switching between these pages, the corresponding faders were not updated properly. To be exact, they were not updated at all.

This failure was caused by a single missing line, which sets all faders to their initial position before requesting their current state. Thus, the fault was preventable, since a simple unit test could have covered this functionality.

Issue #4 Fullscreen on iOS not working: Luminosus is designed to be cross-platform compatible. Thus, when working on a desktop computer most users would like Luminosus to be presented in fullscreen mode. This wish was realized by implementing a button, which offers just this functionality. However, when running Luminosus on iOS, the fullscreen button caused the UI scaling to increase. This rendered the software practically unusable.

The exact cause for this failure is not discovered yet. Still you could say this failure was preventable by having a unit test which compares the scale property before and after the operation. However, the fault could also be treated as intelligent related, since the button should have not been enabled for mobile devices in the first place. So even if the function would have been working as expected, the resulting behavior would be irritating as the only difference of the fullscreen mode on a mobile OS is the removal of the status bar, which does not help the user at all.

Conclusion In general the introduction of unit tests would be a valuable addition to the existing code base. Furthermore, integration tests for typical user scenarios would help to prevent many kinds of faults. Last but not least we learned that to maintain so many different platforms the user feedback is an important part in the process of V&V.

0.3 Five Basic V&V Questions

When do V&V start? When are they complete? Validation started right in the beginning. The project was validated on a daily basis due to a lot of contact to the supervisor. The specification was adjusted very frequently to ensure that the features that were developed were suitable for the projects goal.

Analysis began early, too, in the form of static code analysis provided by the IDE (Qt Creator) and Clang. The development of tests did not start until the beginning of this V&V lecture.

The process of V&V does not end until the end of the lifetime of the product. In this case as long as the software is maintained.

What particular techniques should be applied during development? Static code analysis is very good applicable for C++ due to strict type declarations.

Continuous Integration should be applied to the project since it will be used on various platforms. Furthermore, automated unit testing will be applied during the semester.

Additionally, profiling the memory consumption can help finding memory leaks and performance benchmarks make sure that the specification is met in terms of latency and responsibility.

In the end a formal verification of standardized components such as the network protocol implementations should be applied.

How can we assess the readiness of the product? The readiness is assessed by using the defined features of the specification and their current implementation status. Before a release is considered ready it should be at least tested manually in a typical use case scenario.

Furthermore, open issues are a hint that the product is not yet ready.

In the end the validation of a product is easier when the developer is also a user, as in our case. The greater the distance of the developer to the users of the product, the harder it is to match the use cases with the implementation.

How can we control the quality of successive releases? To ensure that successive releases do not introduce new faults a pre-release policy can be established to let a small group of beta users test the system before it is released to the public.

The successful run of automated tests, especially regression tests, in a Continuous Integration environment can be a good indicator too.

How can the development process itself be improved? Increasing emphasis on applying various V&V techniques during further development will help to improve overall code quality.

Furthermore the development process can be improved by introducing a second developer to the project. Thus, pair programming and code reviews can be established

to help finding flaws. In addition maintaining the documentation will lead to a better process.

1 Analysis

1.1 Human-based Static Analysis

The following section is about the peer review we performed. We figured out advantages and weaknesses of this technique.

1.1.1 Peer Review

We organized our peer review sessions by picking suitable code snippets for each other. These snippets were supposed to be readable, even for someone who is not as deeply involved in the project or even the programming language at all. We then presented the selected code to the other group, while also introducing the general project. Afterwards, we further increased our understanding by asking questions about unclear syntax or general questions about details within the code. Finally, we were able to discuss issues and tried to find suitable solutions.

The code snippet we prepared is shown below.

```
src/eos_specific/EosCue.cpp
171 void EosCue::createCueBlock() {
172     EosCueBlock* block = qobject_cast<EosCueBlock*>(
        m_controller->blockManager()->addNewBlock("Eos_Cue"));
173     if (!block) {
174         qWarning() << "Could not create Cue Block.";
175     }
176     block->setCueNumber(m_cueNumber);
177     block->focus();
178 }
```

As requested, the code contained a fault. This fault was to be detected by our partners for the sake of this peer review. The fault was a missing **return**-statement after line 174. This caused the function to continue its execution even when the **block** is a **nullptr**. Consequentially, line 176 would try to dereference this invalid pointer thus causing most likely an uncaught exception leading to a termination of the software.

After only a short while, our partners were able to locate and also fix the mentioned defect.

In return, they had a code snippet prepared for us which included a defect as well.

When we finished understanding the context and general idea of the presented code, we were able to find the defect. Furthermore, we also suggested a change in the documentation, since it was misleading.

All together, our peer review session was mainly used for communication purposes and understanding the projects of each other. Additionally, the peer review helped us to exchange thoughts about various faults not only within our group but also with members of our partners group.

Main advantage of this kind of peer review is the fact, that every code snippet can be discussed directly with its author. This enables a very deep discussion about certain implementation decisions and also improves the overall code understanding by far. After just a short while, we were even able to detect a misleading documentation for a certain implementation, although we never saw the code before.

In general, peer reviews can help to improve the product by selecting difficult code and talking about it in a very objective way. Thus, we do not need to play the blaming game and can just focus on finding and avoiding faults for future implementations. Additionally, it is very important to do this kind of analysis on a regular basis, so that the reviewed code is still manageable in size and difficulty.

1.1.2 Tools

When performing peer reviews in a more professional way than we did, companies often use tools to support their process. The following section will evaluate some of the tools we already used.

GitHub

A tool we are very comfortable using is GitHub¹. GitHub does not only allow you to host git repositories, so all your code is in just one place, but also offers a great variety of reviewing tools, like commenting coding line by line (see fig. 1.1 on the next page). The concept of pull requests and its support is also a very strong feature in GitHub. It offers developers a very formal way of merging branches into each other. It supports a human based review process by naming one or more reviewers for the pull requests. These reviewers may then comment, ask questions or suggest improvements for the code at hand. Although this process is mostly covered by the features offered by GitHub, we would like to have a way of (autonomously) selecting code of interest as we did in our peer review. Right now, every reviewer has to read all the code submitted for merging, making it very time consuming.

While it is comfortable for the team members that these functions are integrated into the GitHub website and work in every browser, this also means that no code exploration features known from various IDEs are available. As a beginning it would be nice, for example, to be able to jump to a function declaration right from the review page of GitHub. This would help understanding the code a lot.

¹<https://www.github.com>

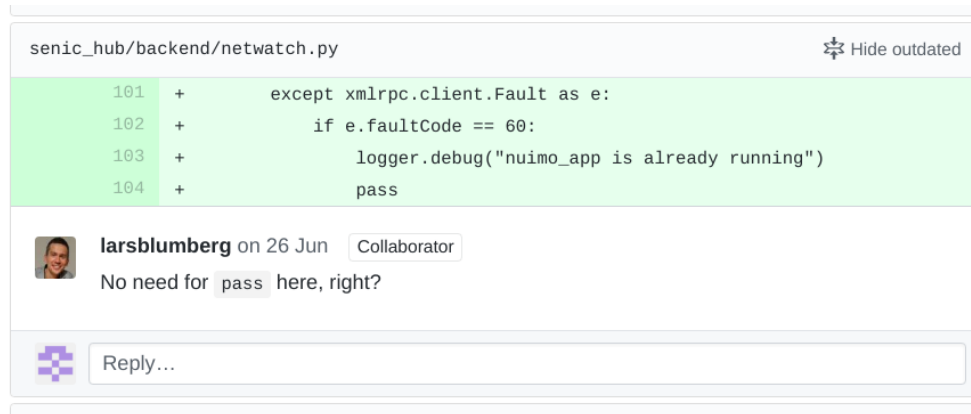


Figure 1.1: A comment in the GitHub review tool

SonarQube

Besides GitHub, SonarQube² is another excellent web based tool for reviewing code. In contrast to GitHub, SonarQube is not used as a repository host. What makes SonarQube still great is its build-in fault detection and static analysis. Thus, a reviewer can find any flaws very fast and also see, who authored the corresponding code. We will cover more of the automated static analysis feature of SonarQube in section 1.3.

What we missed when we used SonarQube was the possibility to see the entire committed code at once. SonarQube only offers to jump to fault it detected, but has no possibility to see the entire commit.

1.2 Automated Static Analysis

After we discovered and tested the human-based analysis of code, we also wanted to try automated analysis. Therefore, we chose tools, which are commonly used when analyzing C++ code. Our findings are documented in the following section.

1.2.1 Clang Static Analyzer

The first tool we tested was the Clang Static Analyzer³. We decided to test this tool not only because it has a great plugin for the IDE of our choice (Qt Creator⁴), but also because it is around for quite a while and therefore commonly known in the C++ community.

On the one hand, Clang helped to improve the code by finding a possible nullpointer dereference caused by a missing return statement (see section 1.1.1 on page 9). On the other hand it reported hundreds of false alarms in its default configuration, i.e. increase

²<https://www.sonarqube.org>

³<https://clang-analyzer.llvm.org>

⁴<https://www.qt.io/>

of precision of floating point numbers thereby making it harder to detect the real faults (see fig. 1.2).

Inaccuracy The developers of Clang Static Analyzer clearly stated that the tool was made to reported as less false positives as possible. However, they are more concerned about not finding a fault, so they would tolerate a moderate amount of false alarms. This leaves us with the conclusion that the Clang Static Analyzer is mainly incorporating pessimistic inaccuracy, which results in false alarms. However, the tool can be configured to also include optimistic inaccuracy, if needed. This can be realized by adding filters so that some checks will become less strict. This behavior will result in some faults not being detected any longer.

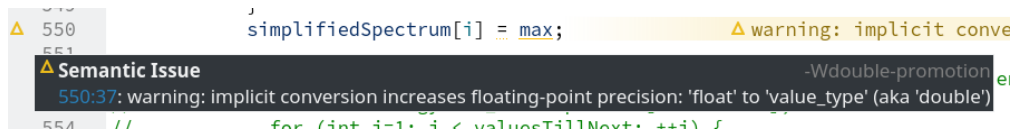


Figure 1.2: Clang warns of increase in precision of floating point number

Figure 1.2 is an example, of a very strict default configuration which leads to a false positive in our concrete project.

1.2.2 Clang-Tidy

Next to the Clang Static Analyzer there is also the command line tool clang-tidy⁵. The tools are quite similar, the static analyzer focuses more on checks that require some sort of control flow analysis while clang-tidy includes linter-style checks and checks that are related to a certain coding style⁶. Other than the static analyzer, clang-tidy is able to automatically fix most of the issues it found by modifying the source files directly, which is a very powerful feature to modernize legacy code bases.

In our case, clang-tidy was able to point out some hard to find readability issues such as a misleading use of `static_cast` where `dynamic_cast` would be more appropriate because it was a downcast from a base to a derived class. It also suggested good uses for the `auto` keyword, missing `override` keywords and showed function definitions where the variable names did not match those of the declaration. It even highlighted a case where a function call contained an inline comment with the name of the parameter (like `'foo(/*parameterName=*/ true)'`) that did not match the real parameter name. After a quick configuration we were able to suppress most of the warnings related to a different coding style resulting in mostly true positives were shown.

⁵<http://clang.llvm.org/extra/clang-tidy/>

⁶see discussion here: <http://lists.llvm.org/pipermail/cfe-dev/2015-September/044966.html>

1.2.3 CppCheck

Although CppCheck⁷ found faults which Clang missed, such as uninitialized member variables, we more disappointed for the most part. This is, because CppCheck claims to be designed to show as less false positives as possible, but still displayed a huge bunch of them for our project (see fig. 1.3).

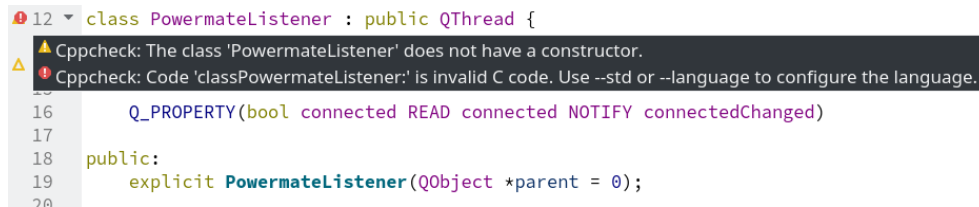


Figure 1.3: Bugs in CppCheck result in false positives

This unwanted behavior is most likely caused by a bug in CppCheck itself, making it even more unsatisfying to use this tool in the first place.

Inaccuracy According to their website, the goal of CppCheck is to have zero false positives, resulting in a low pessimistic inaccuracy. They even recommend to use an additional tool that has a greater pessimistic inaccuracy, if required for the project. Unfortunately, in practice there are still some false positives, mainly related to bugs in CppCheck, as already mentioned before. The developers state that there are many faults that are not detected by CppCheck which leads to a high optimistic inaccuracy.

1.2.4 Polyspace

Other than the previously shown tools, Polyspace⁸ is a commercial product, which is made for industry standards of security critical environments. It claims to be capable of handling huge amounts of code and is also able to apply very in-depth analysis techniques, such as proving the absence of some types of faults, like buffer overflows, division by zero and wrong array accesses⁹.

In the preparation of this report we tried to apply this tool to our project, but experienced some problems in using it (see fig. 1.4 on the next page). We were only able to validate one single source file, the only file with no dependencies other than the C++ standard library. To resolve the Qt includes, it would be necessary to monitor the build process of the project by Polyspace, which was not possible due to limited access to the system where Polyspace was installed on.

Inaccuracy Polyspace is the only tool that we evaluated which claims to have zero optimistic inaccuracy for a specific set of faults. It realizes this by abstract interpretation

⁷<http://cppcheck.sourceforge.net>

⁸<https://www.mathworks.com/products/polyspace.html>

⁹see <https://www.mathworks.com/products/polyspace-code-prover.html>

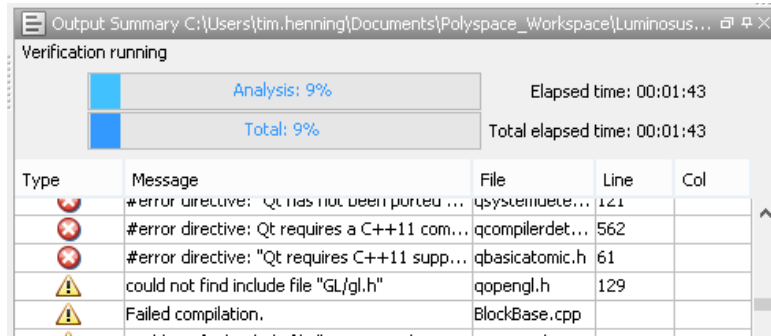


Figure 1.4: Polyspace showing errors while analyzing C++ code with Qt dependencies

of the code and automated formal validation methods.

1.2.5 CppLint

A tool that is specialized in style guide checks is Googles `cpplint.py`¹⁰. Even while this projects style guide differs from Googles, the tool can give some interesting hints. As an example, it was able to find missing includes that the other tools did not report. This is a valuable addition and shows again, that in the case of automated static analysis it is better to use a bunch of tools instead relying on a single one.

1.3 Interaction between Human-Based and Automated SA

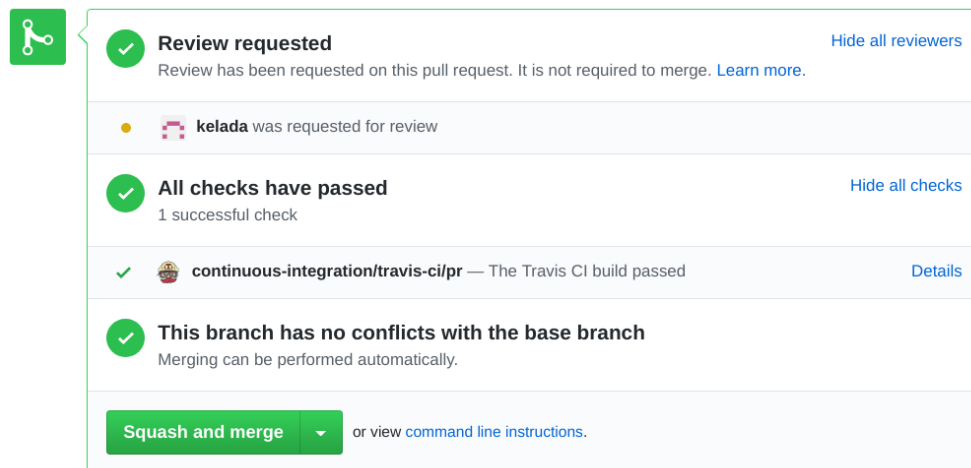


Figure 1.5: Combination of ASA and human-based reviews in GitHub

¹⁰<https://github.com/google/styleguide/tree/gh-pages/cpplint>

By using ASA a peer review can be prepared in order to point out potentially broken code parts. It can then be discussed if a detected fault is considered a false positive or not, and how to fix it. Additionally, some faults might be caused by a bad habit within the project team. These faults are very interesting to discover and can indicate certain lacks of the developing team, enabling more overall awareness once discussed thoroughly. They might even cause a change in the general consideration of good code in the team.

When using tools like SonarQube, the faults get matched with their author. This enables each developer to learn from their mistakes in a more direct way. Additionally, it also displays which kind of faults are often produced in a project. These faults can then be targeted by further education and more awareness.

When thinking the other way around, the results of a human-based peer review can be used to improve the configuration of the automated static analysis tools. This can reduce the number of false positives which are caused by the coding guidelines of the individual project and thus are intended. Depending on the use case, the relevance of faults is differently prioritized. In our specific case the implicit increase of precision of floating point numbers is not a problem at all. In contrast, in more space critical applications this could lead to unpredictable behavior.

In addition to the combination of ASA and human-based reviews, testing can also be integrated in this process. GitHub is a good example for this, as it shows the result of different software verification and validation methods at one glance, as seen in fig. 1.5 on page 14.

2 Testing

2.1 Kick-off Tasks

2.1.1 Example Test Set

As an example bug we chose issue #17¹. It describes a problem where the application sometimes crashes when receiving cue objects. The following lines of code caused this:

```
src/OSCNetworkManager.cpp
78 void onIncomingEosMessage(const EosOSCMessages& msg) {
    [...]
92     } else if (msg.path().size() <= 5) {
93         // this message contains detailed information about
           a cue
94         EosCueNumber cueNumber = EosCueNumber(msg.pathPart
           (2), msg.pathPart(3), msg.pathPart(4));
           [...]

```

Test-Case #1

The first test case should be designed to not only reach the fault but also produce the failure.

Test Value

An `EosOSCMessages` object, where `path().size()` equals 4.

Expected Output

The application should dispose the message, since its too short.

Fault

The fault is persisted in line 94, when `msg.pathPart(4)` is accessed. In our test case, this statement would access unallocated memory, leading to the `SegmentationFault`.

First Error State

The first error state is reached, as soon as the `if-statement` in line 92 is evaluated to `true`. Consequentially, this leads to the execution of the following block, thus producing the fault.

¹<https://github.com/ETCLabs/LuminosusEosEdition/issues/17>

Failure

The program crashes, leaving an unresolved `SegmentationFault`.

Test-Case #2

The second test case should not reach the fault at all.

Test Value

An `EosOSCMMessage` object, where `path().size()` equals 6.

Expected Output

The program continues processing the message, as specified by the protocol.

Fault

The fault is still the same as in section 2.1.1 on page 16. However, since the `if-statement` in line 92 evaluates to `false` the fault is not reached.

First Error State

-

Failure

-

Test-Case #3

The third test case is designed reach the fault while not producing an error.

Test Value

An `EosOSCMMessage` object, where `path().size()` equals 5.

Expected Output

The program reaches the fault, but does not produce an error. Thus, the message is processed according to the specification.

Fault

The fault remains as described in section 2.1.1 on page 16.

First Error State

This test case does not produce a first error state. This is, because line 94 accesses only valid elements of `msg.path()`.

Failure

-

2.1.2 Type of Existing Test Set

There is no existing test set. In the past, only manual testing and partly automated "monkey testing" was performed. A beta phase before releases was used to integrate the feedback from user testing.

2.1.3 Different Scopes of Test Case Execution

The following section will demonstrate, how small, medium and big tests can be applied to our project.

In order to do so, we choose to test following code snippet.

src/OSCNetworkManager.cpp

```
1  /*
2  Extracts one valid OSC packet from the given buffer.
3  The packet will then be removed from the buffer and is
   returned as QByteArray.

4  */
5  QByteArray OSCNetworkManager::
   popPacketLengthFramedPacketFromStreamData(QByteArray&
   tcpData) const {

6      [...]
7  }
```

As described in the documentation, this method will extract and remove a valid OSC packet from the given buffer (`tcpData`) and returns this packet as `QByteArray`.

This piece of code is especially interesting for our project, as it can be tested on multiple scope sizes as follows:

Small Testing

For small testing, the method is tested isolated (e.g. without any context). We can apply different testing strategies for this method:

Validation Testing

Test Value

A buffer (`tcpData`), which contains exactly one valid OSC packet.

Expected Output

A valid `QByteArray`, which contains all data of the given buffer. Additionally, the given buffer must be empty.

Defect Testing

Test Value

A buffer (`tcpData`), which does not contain a valid OSC packet.

Expected Output

An empty `QByteArray`. Additionally, the buffer is not modified.

These are only two examples, which do not complete any coverage criteria. Thus, the set of test cases should be expanded in the future. However, as for demonstrating how this method can be used to create small tests, these examples should be perfectly fine.

Medium Testing

For medium testing we added a little more context to the shown method. This will lead to a longer stack trace, as more methods are involved. One way of expanding the system under test is shown below:

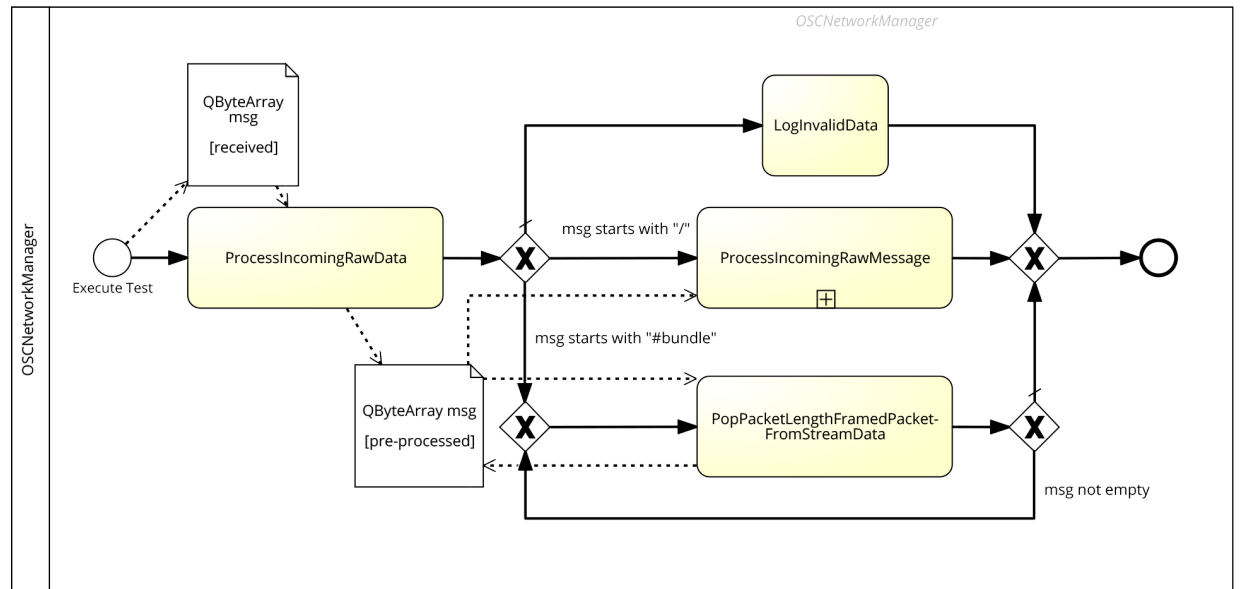


Figure 2.1: Testing context on a medium scale

When testing the displayed context, we need start testing the method `void OSCNetworkManager::processMsgData`. Thus, we span a call graph as shown in fig. 2.1. The edges can be reached by manipulating the `msg` input parameter.

One method to develop test cases for our scenario is to define subsets for our input domain. To be concrete, we divided the input domain as follows:

- `msg` is empty
- `msg` is not empty
 - `msg` starts with "/"

– `msg` start with `"#bundle"`

When using this input domain subsets, we can define a test set, which covers every edge of the call graph. For evaluation of the test results we need to use a human oracle, as the concrete expected behavior depends heavily on the input data and can not be verified by specification.

Big Testing

As for big testing, we decided to replace the synthetic `msg` object from fig. 2.1 on page 19 with the real network adapter. This allows us to also include other components, which are depending on the networking module.

In order to execute tests on this scope, a real TCP connection (e.g. a TCP client, which uses the running application as its server) is needed. Using this setup, we would like to perform defect testing mainly. This is, because including hardware (such as the networking card) on a real network configuration might lead to unexpected behavior, which is otherwise hard to test. Concrete, we would like to test various things on our entire networking subsystem, like packet fragmentation, packet loss and the behavior when connections are closed unexpectedly. These test cases are very hard to simulate in a synthetic environment, since the exact appearance of the previously named factors are rather hard to predict and therefore very hard to create in automated tests.

2.2 Test Case Development

There are multiple different approaches to generated test values for a specific test case to reach a certain kind of coverage. We will discuss some of them in the following sections, using the `FileSystemManager::saveFile()` and `ColorMatrix::rescaleTo()` methods as examples.

2.2.1 Input Domain Modeling

One approach to generate test cases for a function is Input Domain Modeling (IDM). IDM can either be done functionality or interface based. In the first case only the type of the arguments is considered, regardless of the functionality behind the code, while in the latter case the combined meaning of the arguments is used to partition the input domain.

In this section we want to find test cases using IDM for the method `saveFile()` shown in ?? 2.1 and compare interface to functionality-based IDM.

Listing 2.1: `src/FileSystemManager.h`

```
47  /**
48  * @brief saveFile saves QByteArray object to a file in the
      data dir
```

```

49 * It overwrites the file if it already exists.
50 * @param dir sub dir inside the app data dir
51 * @param filename for the file that will be written
52 * @param content to be written
53 * @return true if the file was successfully written
54 */
55 bool saveFile(QString dir, QString filename, QByteArray
    content) const;

```

Interface-Based Interface based partitioning of the arguments of `saveFile()` could for example result in the characteristics q_1 and q_2 that describe the length of the strings `dir` and `filename` and q_3 as the size of the `QByteArray` in `content`. q_1 and q_2 have the partitions $length = 0$, $length = 1$ and $length > 1$ and q_3 is similarly divided into $size = 0$, $size = 1$ and $size > 1$.

This would mean that $3 * 3 * 3 = 18$ test cases are required to cover *All Combinations (ACoC)* of the blocks of the interface based input domain. The number of test cases needed can be decreased by using the *Each Choice (EC)* criterion. The number of test cases would then be the size of the largest characteristic, being 3 in this case. With *Pair Wise (PW)* choice $3 * 3 = 9$ test cases would be required. In the end the most common criterion to choose combinations of blocks is *Base Choice (BC)*. This would require $1 + (2 + 2 + 2) = 7$ test cases and leads to many test cases that are close to the real use of the function.

Functionality-Based For functionality based partitioning only two characteristics are required in this case: q_1 as the validity of the provided file path (including the filename) and q_2 as the size of the content in relation to the available disc space. The partitions for q_1 are *valid filename* and *invalid filename* (respective to the used filesystem, i.e. NTFS). q_2 can be divided into $size = 0$, $size = 1$, $1 < size \leq availableSpace$ and $size > availableSpace$.

To cover this partitioning the following numbers of test cases are required:

- *All Combinations (ACoC)*: $2 * 4 = 8$ test cases
- *Each Choice (EC)*: 4 test cases
- *Pair Wise (PW)*: $2 * 4 = 8$ test cases
- *Base Choice (BC)*: $1 + (1 + 3) = 5$ test cases

Comparison In our case functionality based testing will most probably lead to much more interesting results. Because the interface based method only considers the length of the strings provided for `dir` and `filename` the resulting paths will be invalid in most of the cases and the function will then always fail. The functionality based approach takes this into account and gives therefore the chance to see the effect of the other argument, the content of the file, on the result of the function.

2.2.2 Logic Expressions

Logical expression coverage is another criterion to create test cases for a certain part of code. It aims at finding bugs and unexpected behavior by not only executing each branch of code but also evaluating all clauses in conditions separately. This is not guaranteed by other coverage criteria because short-circuiting conditions may not evaluate all their clauses, even if there are test cases for the true and false cases.

We will design test cases for logical expression coverage based on the condition in the code shown in ?? 2.2.

The individual clauses are:

$a = indexChangedDuringTouch$

$b = isTap$

$c = dropDownItem$

The predicate is $a \vee (\neg b \wedge c)$.

Listing 2.2: qml/CustomControls/ComboBox2.qml

```
123 // if the index changed
124 // or if it was not a tap and there is a DropDown item:
125 if (indexChangedDuringTouch || (!isTap && dropDownItem)) {
126     // destroy the DropDown:
127     dropDownItem.destroy();
128 }
```

Predicate Coverage (PC) Predicate coverage is reached by finding test cases where the logical expression, the *predicate*, is at least once true and false.

To find suitable test cases we can use the truth table shown in table 2.1 on the next page. One test case where the predicate is true (1, 2, 3, 4 or 7) and one where it is false (5, 6 or 8) is enough to satisfy PC, i.e. the test cases 1 and 8.

Clause Coverage (CC) For clause coverage it is required that each clause of the expression evaluated in at least one test case to true and false. Many test sets would fulfill this requirement, like (2, 7) or (3, 6). The test set (1, 8) has the additional benefit that it also satisfies predicate coverage.

Correlated Active Clause Coverage (CACC) Correlated Active Clause Coverage, where each clause is tested independently, can be reached by choosing one pair of (1, 5), (2, 6) or (3, 8) for the major clause a , the test cases (5, 7) for the major clause b and (7, 8) for the major clause c .

A minimal test set that satisfies PC, CC and CACC would then be (1, 5, 7, 8).

Combinatorial Coverage (CoC) The most comprehensive coverage criterion for logical expression based testing is to test all combinations of truth values of each clause. This needs 2^N tests, with N being the number of clauses. All 8 combinations are shown in table 2.1.

Table 2.1: Correlated Active Clause Coverage (CACC)

	a	b	c	$a \vee (\neg b \wedge c)$	p_a	p_b	p_c
1	T	T	T	T	*		
2	T	T	F	T	*		
3	T	F	T	T	*		
4	T	F	F	T			
5	F	T	T	F	*	*	
6	F	T	F	F	*		
7	F	F	T	T		*	*
8	F	F	F	F	*		*

2.2.3 Control Flow Graph

Node Coverage The simplest way to cover a control flow graph as seen in fig. 2.2 on the next page is to create test cases that visit all nodes at least once. For the `rescaleTo()` method three paths have to be covered for this: (0, 1), (0, 2, 3), (0, 2, 4, 5, 6, 7, 8, 7, 9, 10).

The four input parameters that `rescaleTo()` depends on, are the current width and height of the matrix and the parameter of the function `sx` and `sy`. To execute the three mentioned paths the following input values can be used: [1, 1, 1, 1], [1, 1, 0, 0], [2, 2, 1, 1].

Edge Coverage The same test values can be used to reach edge coverage, i.e. visit all edges of the graph at least once.

Edge Pair Coverage To cover all pairs of edges, two additional paths would have to be covered. The path (0, 2, 4, 5, 10) is infeasible because the second if-statement makes sure that the rest of the method is only executed when the size changes, which means the for-loops are executed every time node 4 is reached. The path (0, 2, 4, 5, 6, 7, 8, 7, 9, 10) is expanded to (0, 2, 4, 5, 6, 7, 8, 7, 8, 7, 9, 5, 6, 7, 8, 7, 8, 7, 9, 5, 10) to also cover the edge pair (9, 5, 6). This path is executed with the input values [2, 2, 1, 1].

Prime Path Coverage The following prime paths were calculated with the online tool by Ammann and Offutt²:

a) 0, 1

²<https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage>

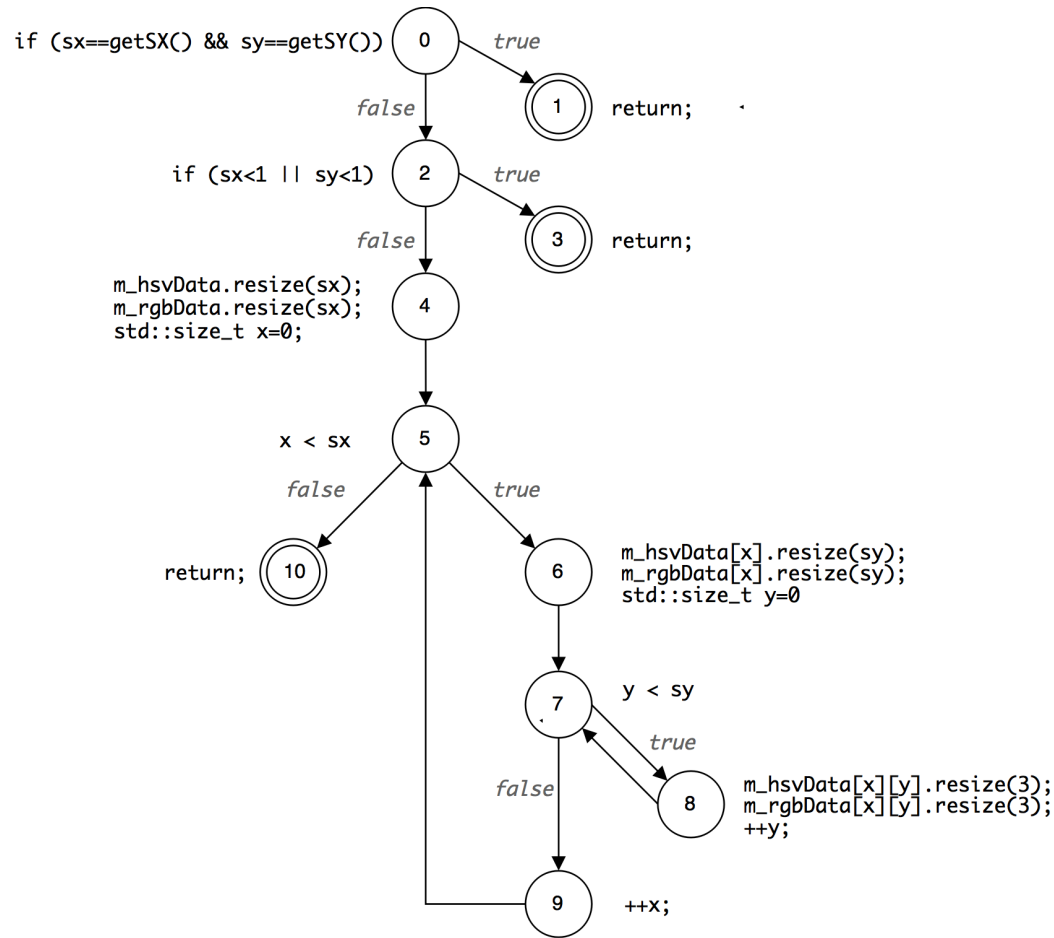


Figure 2.2: Control Flow Graph of the rescaleTo() Method

- b) 0, 2, 3
- c) 0, 2, 4, 5, 10
- d) 0, 2, 4, 5, 6, 7, 8, 7, 9, 5, 10
- e) 0, 2, 4, 5, 6, 7, 8, 7, 8, 7, 9, 5, 10
- f) 0, 2, 4, 5, 6, 7, 9, 5, 6, 7, 8, 7, 9, 5, 6, 7, 9, 5, 10

Path c is again infeasible. Path f is infeasible, too, but to cover at least the contained prime path (7,9,5,6,7) the path (0, 2, 4, 5, 6, 7, 8, 7, 8, 7, 9, 5, 6, 7, 8, 7, 8, 7, 9, 5, 10) was chosen as an alternative.

The following table shows the input values to execute those paths. The values were determined manually because of the lack of tool support for this.

Table 2.2: Test Cases for Prime Path Coverage

Path	width	height	sx	sy
0, 1	1	1	1	1
0, 2, 3	1	1	0	0
0, 2, 4, 5, 6, 7, 8, 7, 9, 5, 10	2	2	1	1
0, 2, 4, 5, 6, 7, 8, 7, 8, 7, 9, 5, 10	2	1	1	2
0, 2, 4, 5, 6, 7, 8, 7, 8, 7, 9, 5, 6, 7, 8, 7, 8, 7, 9, 5, 10	1	1	2	2

Complete Coverage The complete coverage of all existing paths is not possible as there is an almost infinite number of paths due to the for-loops.

2.3 Automated Unit Tests

The following chapter is about how test cases, which were developed as examples earlier, can be executed and validated automatically. Therefore, we decided to try (a) Googles C++ Testing Framework³ as well as the (b) Qt Testing Framework⁴. These two frameworks offer similar set of features and will be evaluated in the following sections.

2.3.1 Google Test

This section is about the basics of the Google Testing Framework for C++. We will explain what is needed to setup the framework and how to write tests. Additionally, we will display how Googles Testing Framework is integrated in the QtCreator.

Setup

Setting it up is rather simple: after installing the *gtest* and *gmock* libraries, it is possible to create a new executable that executes all available tests with only a few lines of code:

```

                                tests-google/main.cpp
1  #include <gtest/gtest.h>

3  int main(int argc, char[] argv) {
4      ::testing::InitGoogleTest(&argc, argv);
5      return RUN_ALL_TESTS();
6  }
```

³<https://github.com/google/googletest>

⁴<http://doc.qt.io/qt-5/qtest-overview.html>

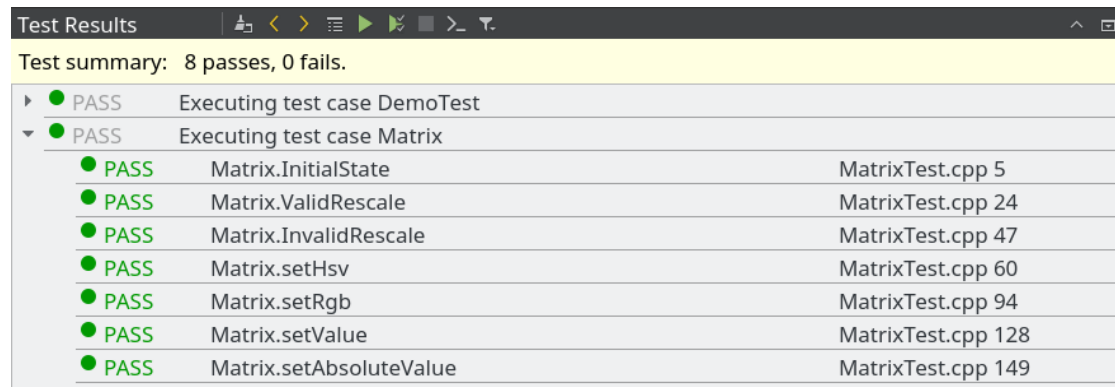
Example Test

A very basic test could look like this:

```
tests-google/Demo.cpp
1  #include <gtest/gtest.h>
3  TEST(DemoTest, Add) {
4      ASSERT_EQ(1+1, 2);
5  }
```

IDE Integration

QtCreator has a basic Google Test integration. It can start all or only individual tests and displays the test results in the user interface.



The screenshot shows the 'Test Results' panel in QtCreator. At the top, a yellow bar indicates 'Test summary: 8 passes, 0 fails.' Below this, a tree view shows the test execution results. The 'DemoTest' is expanded, showing a 'PASS' status. The 'Matrix' test case is also expanded, showing seven individual tests, all of which passed. Each test entry includes a green circle icon, the word 'PASS', the test name, and the file and line number where the test is defined.

Test Case	Test Name	File
Executing test case DemoTest		
Executing test case Matrix		
Matrix.InitialState	PASS	MatrixTest.cpp 5
Matrix.ValidRescale	PASS	MatrixTest.cpp 24
Matrix.InvalidRescale	PASS	MatrixTest.cpp 47
Matrix.setHsv	PASS	MatrixTest.cpp 60
Matrix.setRgb	PASS	MatrixTest.cpp 94
Matrix.setValue	PASS	MatrixTest.cpp 128
Matrix.setAbsoluteValue	PASS	MatrixTest.cpp 149

Figure 2.3: Google Test Results in QtCreator

2.3.2 Qt Test

While the Google Test Framework is well suitable for general C++ code and has its major advantage in being universal for all platforms and types of code, it has some disadvantages when testing more Qt specific functionalities. With Google Test, much boilerplate code is required to test the Qt Signal&Slot mechanism for instance.

To overcome these issues we also looked at the Qt Test Framework and evaluated whether it solves them.

Setup and Test Driver

Setting up the Qt Testing Framework does not require any additional libraries to be installed (assuming one uses the QtCreator IDE, as we did). However, setting up the

test driver requires more code as for the Google driver because the tests can not be found automatically. This is not necessarily a bad thing as it opens up a more flexible test configuration, but may also be more error prone.

The test driver we use for the tests looks like this:

tests-qt/main.cpp

```
1  #include <QtTest>

3  #include "MatrixTest.h"
4  #include "OscNetworkManagerTest.h"
5  #include "BlockCreationTest.h"

7  int main(int argc, char** argv) {
8      QApplication app(argc, argv);
9      QList<QObject*> testCases;
10     testCases << new MatrixTest();
11     testCases << new OscNetworkManagerTest();
12     testCases << new BlockCreationTest();

14     int returnValue = 0;
15     for (QObject* testCase: testCases) {
16         returnValue = returnValue | QTest::qExec(
            testCase, argc, argv);

17     }
18     return returnValue;
19 }
```

Example Test

Implementing test cases follows the *convention over configuration* approach. For each group of test cases a custom class that inherits from `QObject` has to be created. To prepare a set of test cases, the method `initTestCase()` can be declared and is executed automatically by Qt Test before any test case is executed. The method `cleanupTestCase()` handles the opposite case of cleaning up after the execution of all tests. `init()` and `cleanup()` work in the same fashion, but are executed again before each single test case. An example class that declares all these methods can be seen here:

tests-qt/ClassUnderTest.h

```
1  #pragma once
```

```

3  #include <QtTests>

5  class ClassUnderTest : public QObject {
6      Q_OBJECT
7  private slots:
8      // Will be called before the first test function is
        executed.

9      void initTestCase();

11     // Will be called after the last test function was
        executed.

12     void cleanupTestCase();

14     // Will be called before each test function is
        executed.

15     void init();

17     // Will be called after every test function.
18     void cleanup();
19 }

```

Each additional method of the test class which's name starts with 'test' is considered as a test case and is found by Qt with the introspection feature of QObjects.

A very simple test case like in the Google Test example would look like the following:

tests-qt/Demo.cpp

```

1  void testDemo() {
2      QCOMPARE(1+1, 2);
3  }

```

Differences to Google Test

Simulating Mouse and Keyboard The controllability of GUI tests is much increased by Qt Tests as it offers the functions `keyClick()` and `mouseClick()`. They simulate keyboard and mouse events by sending internal Qt events instead of hijacking the input device on operating system level, which means that there are much less side effects when using them.

Sequences of inputs can be specified and the delay between them adjusted by command line parameters. This makes it quite easy to test custom GUI objects like in our application in an isolated and reproducible way.

Testing Signals and Slots To increase the observability, Qt Test provides the class `QSignalSpy`. It makes it possible to include the amount of signal emission in the expected output of a test and even compare the parameter values of the emitted signals.

In addition, `QSignalSpy` can start an event loop inside of a test to be able to detect signals that are send between threads using queues or that are emitted with a short delay. The event loop is quited after a specified timeout and the test is considered failed if the expected behavior was not observed in the meantime.

Beside that, it is possible to log all emitted signals with the `'-vs'` command line parameter, which makes debugging much easier. It also allows to detect unnecessarily repeated signals, which also caused problems in our project before.

Microbenchmarks It is possible to include microbenchmarks directly inside the tests. A reasonable number of iterations is detected automatically while running the tests, the type of time measurement can be chosen and the results are reported right after the corresponding test results. Why this is important for GUI application development is further discussed in section 2.4.2 on page 31.

IDE Integration

The Qt Test Framework is integrated in the graphical user interface of Qt Creator, too. As with Google Test, you can start all or only individual tests and see the results aggregated in a summary at the bottom of the window. One of the advantages of Qt Tests can also be seen here: the results of the microbenchmarks, including the number of iterations (see fig. 2.4).

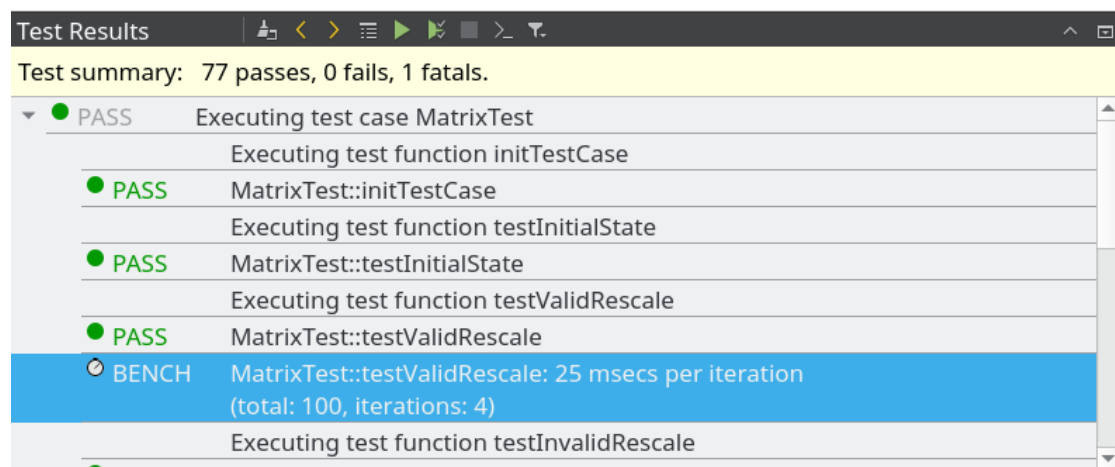


Figure 2.4: Qt Test Results in QtCreator

2.3.3 Code Coverage Calculation

It is not possible to calculate and display the code coverage of tests done with the Google Testing framework or Qt Test within QtCreator.

Fortunately at least the GCC compiler has a compilation option that modifies the program to log the number of times each line of code is executed. The results are stored in separate files next to the executable after the first run of the software and can be analyzed with a tool called *gcov*⁵ or visualized as HTML pages with *lcov*⁶. The coverage results for some of the source files in our project can be seen in fig. 2.5.

Even while *gcov* can only calculate line coverage and not branch coverage or even more complex coverage criterions, it is already a very useful metric to get insights which parts of the application are tested in an automated way. If the line coverage for some files is under 100%, for example after a code change, this is a clear indicator that tests are missing.



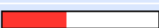


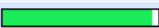
Filename	Line Coverage ↕		Functions ↕	
LogManager.cpp		84.1 %	37 / 44	71.4 %
LogManager.h		100.0 %	1 / 1	100.0 %
MainController.cpp		41.0 %	91 / 222	31.2 %
MainController.h		72.7 %	16 / 22	68.2 %
Matrix.cpp		60.1 %	110 / 183	82.4 %
Matrix.h		96.3 %	26 / 27	90.9 %

Figure 2.5: Test Coverage Results

2.3.4 ASA and Testing

Code Metrics Metrics generated by ASA like cyclomatic code complexity could indicate error prone code which should be investigated whether exhaustive test cases should be developed.

In our case we used automated static analysis to calculate the cyclomatic complexity for the most crucial parts of the software. However the average complexity was quite low, to be concrete between 3 and 5. High complexity values were mostly a result of switch case statements with many branches and we decided to not design test cases for them.

Test Generation Another use case of ASA for dynamic testing could be the automated generation of test cases. The only tool we found that claims to do that for C++ code is VectorCAST⁷. It automatically generates test cases by using decision paths, which can be very useful especially for rather large projects as it potentially saves a large amount of

⁵<https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>

⁶<http://ltp.sourceforge.net/coverage/lcov.php>

⁷<https://www.vectorcast.com/software-testing-products/c-unit-testing>

time. While it would be interesting to try this for our project, this tool was not available for us.

2.4 Evaluation

2.4.1 Bugs Found

Even while writing the first few unit tests, we were able to identify some issues where the code was not behaving as expected.

In the first case an operation was never executed because of a wrong condition in an if-statement (`rescaleTo()` in `Matrix.h`).

Furthermore in the same class it was possible to set a variable to a higher or lower value than specified through its setter method. The expected behavior in this case would be that the variable is then limited to its upper or lower bound.

2.4.2 Performance Regression Tests

The execution time of functions is an especially important aspect in the development of GUI applications. If a function consumes too much CPU time it can not only affect loading times, that can get annoying for the user if they are too long. It can also affect the general perceived responsiveness of the application: the responsiveness is higher if the delays between user interactions and visible changes are kept to a minimum. But in the end on mobile platforms, the most important effect of used CPU time is the battery consumption. When developing an app or in the case of C++ even a part of the operating system, the goal is most of the time to reach the lowest possible power consumption.

A typical way to measure and be able to improve the execution time of single functions are *microbenchmarks*. They measure the used CPU time per function call and summarize the results in a report. The developer can then start identifying performance bottlenecks and fix them.

But in the context of automated testing it would be quite interesting to combine microbenchmarks with tests, especially regression tests.

As most functions in a GUI application have to return in under 20ms for a lag-free user experience, a default timeout of the targeted frame time should be available. Tests that run longer than this duration should be considered failed or at least produce a warning message. The only exception to this are functions that are executed in a dedicated worker thread. It should be possible to mark those functions and disable or increase the timeout for them.

Even better would be the possibility to run performance regression tests. An overview as proposed in fig. 2.6 on the next page could show which of the execution times of tested units increased or decreased after the last change.

This would be very helpful to find regressions in code quality and even misuse of APIs. An example could be that in a bug fix the used sorting algorithm was changed or a linked

Test Results			
Test summary: 8 passes, 0 fails.		Compared with commit 43b03bac:	
▶ ● PASS	Executing test case DemoTest		
▼ ● PASS	Executing test case Matrix		
● PASS	Matrix.InitialState	15% faster	MatrixTest.cpp 5
● PASS	Matrix.ValidRescale	no change	MatrixTest.cpp 24
● PASS	Matrix.InvalidRescale	2% faster	MatrixTest.cpp 47
● PASS	Matrix.setHsv	1% faster	MatrixTest.cpp 60
● PASS	Matrix.setRgb	182% slower	MatrixTest.cpp 94
● PASS	Matrix.setValue	no change	MatrixTest.cpp 128
● PASS	Matrix.setAbsoluteValue	no change	MatrixTest.cpp 149

Figure 2.6: *Mockup*: Possible Visualization of Performance Regression Test Results

list was used instead of a vector. A normal unit test would still pass in this case because the output of the function did not change, but only its CPU and memory footprint.

Another example would be the addition of a parameter check to a function as a fix for a failing test. This would seem reasonable to the developer while trying to make all tests pass, but the developer might miss that this function is called million times per second and the parameter checks were removed intentionally because the functions was revealed as a performance bottleneck beforehand. An automated performance regression tests could easily show the effects of the change to the rest of the code and a more speed friendly fix could be found for the failing test.

In reality, an unusually long execution time is often also caused by a large amount of generated debug or warning messages, which is a good indicator for a bug, too.

An overview of aggregated performance changes over many commits and releases could be a good candidate as an indicator for general project health, too. The developers could be motivated to reach a performance improvement in each release by trying to make the report appearing in green, while of course the readability of the code should still have the highest priority in most cases.

Unfortunately we could not find a feature like this in any of the regarded testing frameworks. Qt Test has the ability to include simple benchmarks in the tests but can not compare the results to those of prior commits. Google Test does not even have the ability built in to make benchmarks while testing. While Google offers a separate benchmarking framework for C++ to solve some of the mentioned issues, it is not easily possible to create regressions overviews, either.

To realize performance regression tests, they would need to be executed each time on the same machine and under the same conditions, or the results would need to be normalized in respect to a prior general performance measurement. But in regard of the possible benefits this seems to be a solvable problem.

3 Verification

Within the next chapter, we will present and discuss different approaches of verifying properties within a given piece of code. The main focus will be directed towards various tools, which support different strategies of verification.

3.1 Test Case Generation

The first of these strategies is to generate test cases automatically. This is especially useful if exhaustive testing is not an option. Therefore, we rely on methods such as symbolic execution to find crucial edge cases and other important inputs, which might lead to unexpected behavior of our software. If done correctly, the generated test cases can prove some properties of the verified code.

We started searching tools specialized in verifying C++ code. However, we realized that tooling support for C++ is rather rare within the public domain. Additionally, many of the tools we found were outdated and no longer maintained, making it impossible to use them with our modern C++ code. The only tool we were able to use for our C++ code was Polyspace. In order to find some more tools we could test, we started looking for other supported languages. We found KLEE¹, a symbolic execution tool for C. Additionally, we tried IntelliTest², another symbolic execution tool for C#. We had to translate our code to valid C and C# code when trying these tools, which limited the amount of code we could verify. Therefore, we decided to translate and test our HSV to RGB conversion method. At this point, we would like to mention that translating code comes with the risk of changing the behavior. Thus, the correctness of the translation must be proven in order to completely verify the given code. However, we verified our code only for the sake of finding and using the above mentioned tools, and therefore, we will not prove that our translation is correct.

3.1.1 KLEE

*"KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure [...]"*¹

We installed KLEE on a virtual Ubuntu machine, using the pre-build docker image provided in their manual. We had to modify our code translation, telling KLEE which variables should be tested (`klee_make_symbolic`). We ended up with the following code:

¹<http://klee.github.io/>, accessed 25th January 2018

²<https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/introduction>

```

21 void hsvToRgb(float h, float s, float v, float* r, float* g,
    float* b) {
    [...]
46 }

48 void main(int argc, char** argv) {
49     float r = 0.f;
50     float g = 0.f;
51     float b = 0.f;

53     float h = 0.f;
54     float s = 0.f;
55     float v = 0.f;

57     klee_make_symbolic(&h, sizeof(h), "h");
58     klee_make_symbolic(&s, sizeof(s), "s");
59     klee_make_symbolic(&v, sizeof(v), "v");

61     hsvToRgb(h, s, v, &r, &g, &b);
62 }

```

We compiled the code, using the included Clang compiler and enabled the `-emit-llvm` flag, so that KLEE could operate on the result. Afterwards, we executed KLEE as follows:

```

~/verification$ klee Matrix.bc
KLEE: output directory is "/home/klee/verification/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: silently concretizing (reason: floating
point) expression (ReadLSB w32 0 s) to value 0 (/home/
klee/verification/Matrix.c:24)

KLEE: done: total instructions = 58
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1

```

Unfortunately, KLEE does not support floating point numbers, resulting in just one test case being generated.

```

~/verification$ ktest-tool klee-last/test000001.ktest

```

```

ktest file : 'klee-last/test000001.ktest'
args      : ['Matrix.bc']
num objects: 3
object    0: name: b'h'
object    0: size: 4
object    0: data: b'\x00\x00\x00\x00'
object    1: name: b's'
object    1: size: 4
object    1: data: b'\x00\x00\x00\x00'
object    2: name: b'v'
object    2: size: 4
object    2: data: b'\x00\x00\x00\x00'

```

The test case consists of zeros for all input values, which is not very helpful for verifying properties within our translated code. Thus, we did not further investigate KLEE.

3.1.2 Visual Studio IntelliTest

IntelliTest is Microsofts symbolic execution testing framework, which is based on PEX³. It is embedded in Visual Studio and can handle managed .NET code.

In order to test our code, we had to translate it to managed .NET code, where C# was our language of choice. With only very little effort, Visual Studio generated a testing project and also generated code for our method. All that was left to do, was to add assumptions and assertions to increase the value of the generated test cases.

The final code for letting IntelliTest generate test cases for our HsvToRgb is shown below:

```

1 [PexMethod]
2 public void HsvToRgbTest([PexAssumeUnderTest]Matrix target,
3
4     float h,
5     float s,
6     float v,
7     out float r,
8     out float g,
9     out float b
10 )
11 {
12     PexAssume.IsTrue(h >= 0.0f && h <= 1.0f);
13     PexAssume.IsTrue(s >= 0.0f && s <= 1.0f);
14     PexAssume.IsTrue(v > 0.0f && v <= 1.0f);

```

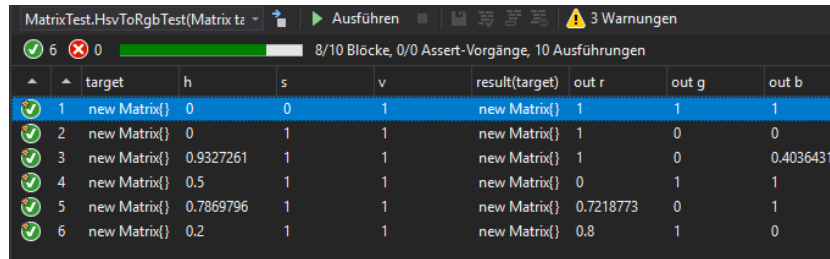
³<https://pex4fun.com>

```

15         target.HsvToRgb(h, s, v, out r, out g, out b);
17
18     PexAssert.IsTrue(r >= 0.0f && r <= 1.0f);
19     PexAssert.IsTrue(g >= 0.0f && g <= 1.0f);
20     PexAssert.IsTrue(b >= 0.0f && b <= 1.0f);
21 }

```

Although IntelliTest generated 6 different tests, which all met our specification, the testing framework reported some issues about floating point numbers. This also seems to be the reason for the low variety of the *s* and *v* parameters. Additionally, IntelliTest reported that the stack observation, used for test case generation, is limited in our case. This might also lead to less test cases being generated.



	target	h	s	v	result(target)	out r	out g	out b
1	new Matrix()	0	0	1	new Matrix()	1	1	1
2	new Matrix()	0	1	1	new Matrix()	1	0	0
3	new Matrix()	0.9327261	1	1	new Matrix()	1	0	0.4036431
4	new Matrix()	0.5	1	1	new Matrix()	0	1	1
5	new Matrix()	0.7869796	1	1	new Matrix()	0.7218773	0	1
6	new Matrix()	0.2	1	1	new Matrix()	0.8	1	0

Figure 3.1: Intelli Test Results

Overall, we can say, that we liked the ease which was provided by the seamless integration of IntelliTest into Visual Studio. The test case generation was fast and well configurable. We were glad to get at least some results, in contrast to KLEE, but wished for better support for more mathematical focused code (such as floating point based calculation).

3.2 ASA without Optimistic Inaccuracy

Because manual theorem proving can be quite difficult and time consuming, it is worth to have a look at tools that claim to automatically prove the absence of certain failures without optimistic inaccuracy. In the following section we want to analyze how well this can be done for our code and what additional benefits automated symbolic execution can have by using the only tool we found for modern C++ code, Polyspace.

3.2.1 Polyspace

This commercial tool can analyze code and prove the existence or absence of run time errors by using abstract interpretation. Beside the "Bug Finder" module used in the ASA section, it also offers "Code Prover" which is specialized in proving certain properties beyond classic static analysis.

The detected issues are separated in four categories⁴:

- **Green:** proven free of run-time errors
- **Red:** proven faulty each time the operation is executed
- **Gray:** proven unreachable (may indicate a functional issue)
- **Orange:** unproven operation may be faulty under certain conditions

The Code Prover results for our code example (hsv2rgb) can be seen in fig. 3.2.

Family	Information	File
Run-time Check		1 47 359
Gray Check		1
Unreachable code		1
Orange Check		47
Illegally dereferenced pointer		13
Non-initialized pointer		10
Non-initialized variable		9
Null this-pointer calling method		2
User assertion		13
Green Check		359
Division by zero		1
Function not returning value		12
Illegally dereferenced pointer		49
Non-initialized local variable		94
Non-initialized pointer		23
Non-initialized variable		2
Null this-pointer calling method		40
Overflow		27
Uncaught exception		106
User assertion		5

Figure 3.2: Polyspace Code Prover Categories

Furthermore Polyspace can show the possible value range of each variable at every given time as a result of the automatic symbolic execution. These information can be very useful for debugging, too. An example for these information can be seen in fig. 3.3 on the next page.

Polyspace can also display useful information about the state of variables for each line of code, for example if the content of a variable could be null (see fig. 3.4 on the following page).

Unfortunately it is very hard to compile code with complex dependencies like Qt in Polyspace and we had to rewrite the `hsv2rgb()` function with pure C++ code and STL containers. This may introduce new errors and behave differently compared to the Qt container classes.

In the end Polyspace has proven to be a very useful tool and especially for developing and optimizing the critical parts of an application it can be worth the money it costs.

⁴<https://www.mathworks.com/products/polyspace-code-prover/features.html>

```

int i = int(h * 6);
f = (h * 6) - i;
p = v * (1 - s);
g = v * (1 - s * f);
t = v * (1 - s * (1 - f));
i = i % 6;
switch (i) {
case 0: r = v, g = t, b = p; break;
case 1: r = g, g = v, b = p; break;

```

operator * on type float 64

left: [-0.0 .. 1.0]

right: 6.0

result: [-0.0 .. 6.0]

Press 'F2' for focus

Figure 3.3: Symbolic Execution in Polyspace

```

h = m_hsvData[x][y][0];
s = m_hsvData[x][y][1];
v = m_hsvData[x][y][2];
assert(h <= 1);
assert(h >= 0);
assert(s <= 1);
assert(s >= 0);
assert(v <= 1);
assert(v >= 0);

```

Dereferenced value (float 64): full-range [-1.7977E+308 .. 1.7977E+308]

Dereference of expression (pointer to float 64, size: 64 bits):

- Pointer is not null (but may not be allocated memory).
- Points to 8 bytes at unknown offset in buffer of unknown size, so may be outside bounds.
- Pointer may point to dynamically allocated memory.

Press 'F2' for focus

Figure 3.4: Polyspace Variable Information

3.3 Theorem Proving

Proving certain properties of source code is especially useful for "mission critical" parts, like single points of failures that could render the whole application useless if they fail, and security relevant features that handle or protect sensible user data. In our application there are no essential parts of either of these categories, as the critical parts of lighting control are implemented in the lighting console and not in this companion app and no sensible information is handled.

Anyway it would still be good to prove for example the correctness of the implementation of the network protocols used to communicate with the console to be able to guarantee a stable connection. Being able to exclude the implementation of the protocols as a source of failures is a huge facilitation while looking for the origin of a connection problem, too.