TAV - Report

Project: Luminosus

Johannes Schneider and Tim Henning

November 28, 2017

Contents

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.

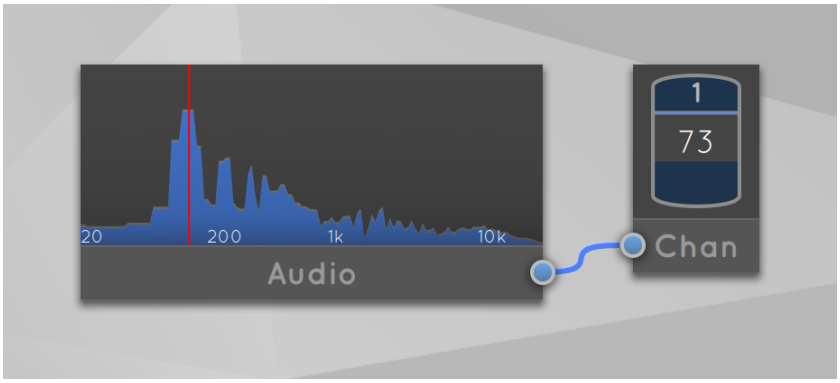

Figure 0.1: Luminosus Example: Controlling the brightness of channel 73 by the current audio level

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.

[1] see *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 QML[2] 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 GitHub[3].

Available artifacts are the source code, documentation, changelog[4], issue tracker[5], manual[6] 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.

0.2.1 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.

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

0.2.2 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 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.

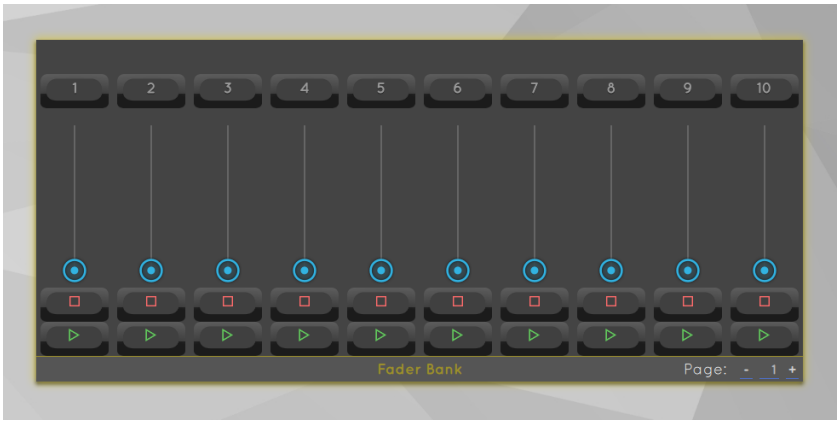

Figure 0.2: Luminosus Fader Bank: Visualization of the Eos faders including virtual pages

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.

0.2.3 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.

0.2.4 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 asses 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

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.

```
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

A tool we are very comfortable using is GitHub[1]. 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). 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.

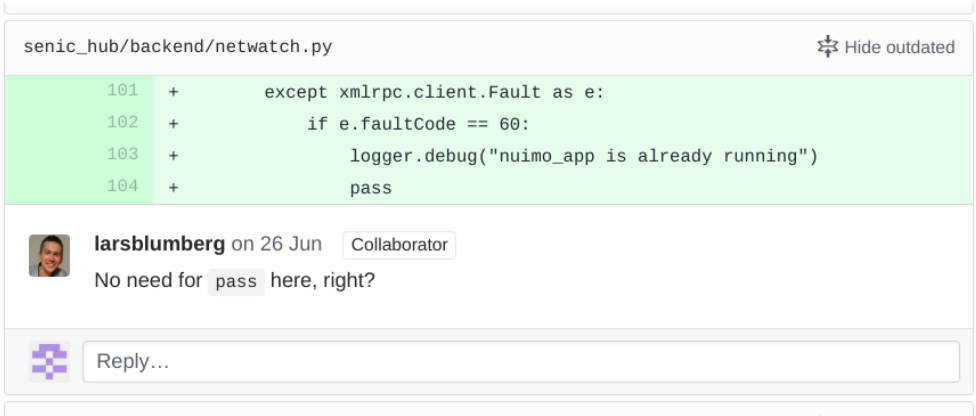

Figure 1.1: A comment in the GitHub review tool

While it is comfortable for the team members that these functions are integrated into the GitHub website and work in every browser, this means that no code exploration

[1] https://www.github.com

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. It would help understanding the code a lot.

Besides GitHub, SonarQube[2] 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.

1.2 Automated Static Analysis

In order to test automated static analysis, we used the Clang Static Analyzer[3] and CppCheck for our project. We chose these tools not only because they offer great plugins for the IDE of our choice (Qt Creator), but also because they are made for analyzing C++ code. Any detected fault is displayed directly in the code editor, next to the line where it is located at.

1.2.1 Clang Static Analyzer

On the one hand, Clang helped to improve the code by finding a possible nullpointer dereference caused by a missing return statement. On the other hand it reported hundreds of false alarms in its default configuration, i.e. increase of precision of floating point numbers thereby making it harder to detect the real faults (see fig. 1.4 on page 12).

1.2.2 clang-tidy

Next to the Clang Static Analyzer there is also the command line tool clang-tidy[4]. 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 style[5]. 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 were 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

[2] `https://www.sonarqube.org`
[3] `https://clang-analyzer.llvm.org`
[4] `http://clang.llvm.org/extra/clang-tidy/`
[5] see discussion here: `http://lists.llvm.org/pipermail/cfe-dev/2015-September/044966.html`

a quick configuration we were able to suppress most of the warnings related to a different coding style and mostly true positives were shown.

1.2.3 CppCheck

Other than Clang, CppCheck[6] found not initialized member variables. But even while it was designed not to show false positives, many of them were shown due to bugs in CppCheck (see fig. 1.2).

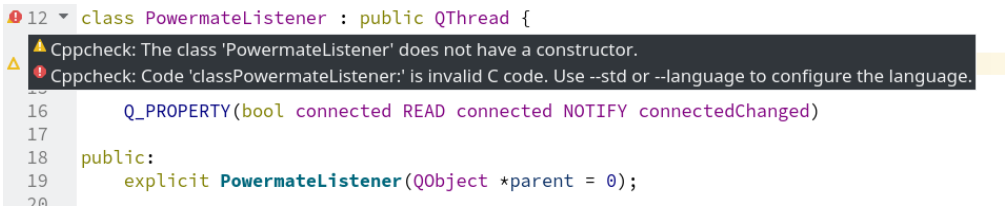

Figure 1.2: Bugs in CppCheck result in false positives

1.2.4 Polyspace

Due to the heavy usage of C++ in security critical environments, there are many non-free tools for bug finding and formal validation. An example for this is Polyspace[7], which even claims to be able to proof the absence of some types of faults, such as buffer overflows, division by zero and wrong array accesses[8].

In the preparation of this report we tried to apply this tool to our project, but experienced some problems in using it (see fig. 1.3 on the following 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 a specific missing compiler on the system.

1.2.5 cpplint

A tool that is specialized in style guide checks is Googles cpplint.py[9]. 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.

[6] `http://cppcheck.sourceforge.net`
[7] `https://www.mathworks.com/products/polyspace.html`
[8] see `https://www.mathworks.com/products/polyspace-code-prover.html`
[9] `https://github.com/google/styleguide/tree/gh-pages/cpplint`

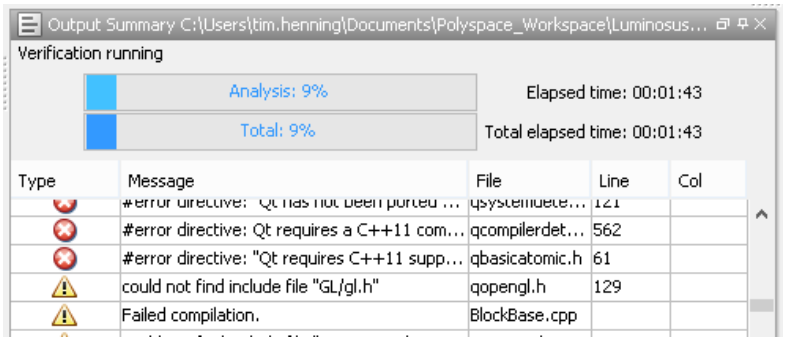

Figure 1.3: Polyspace showing errors while analyzing C++ code with Qt dependencies

1.2.6 Type of Inaccuracy

While investigating the design and philosophy of the Clang static analyzer, the developers 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.

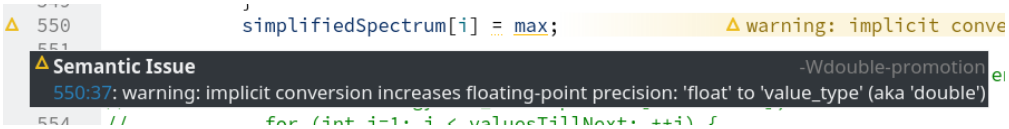

Figure 1.4: Clang warns of increase in precision of floating point number

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.

This shows that trying to show as few false positives as possible often also implies that the user has to live with optimistic inaccuracy and that it is intended that not all faults are detected. Those tools can therefore not guarantee the absence of faults.

Polyspace is the only tool that we evaluated that claims to have zero optimistic inaccuracy for some kind of faults. It realizes this by abstract interpretation of the code and automated formal validation methods.

1.3 Interaction between Human-Based and Automated SA

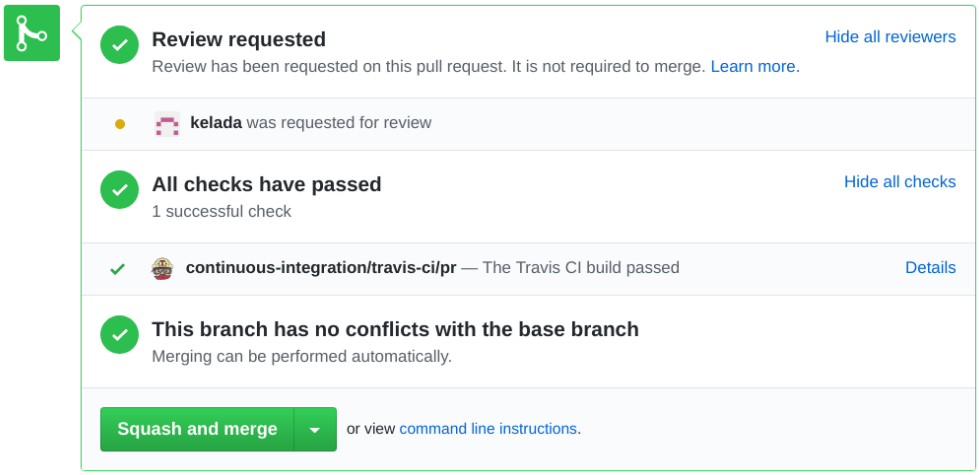

Figure 1.5: Combination of ASA and human-based reviews in GitHub

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.

2 Testing

2.1 Kick-off Tasks

2.1.1 Example Test Set

As an example bug we chose issue #17[1]. It describes a problem where sometimes the application crashes when receiving cue objects. The following lines of code caused this:

```
78  void onIncomingEosMessage(const EosOSCMessage& 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));
            [...]
```

The input data for a test case that leads to a failure would be a `msg` object with less than 5 path parts. The first error state would be that the program flow reaches line 94 with an `msg` object with less than 5 path parts. The fault is therefore in the if-condition in line 92 that evaluates to true in this case. By trying to access the not existing 5th part of the path with `msg.pathPart(4)` the program crashes with an index out of bounds exception.

A test case that does not reach the fault is produced by using a `msg` object thats second path part is not *cue*. This is caught by an if-statement that leads to an early exit before the fault.

If the `msg` object is correct and has 5 or more path parts, the fault is reached but no error is produced.

It is not possible to create a test case that reaches the fault and produces an error without a failure. Each time line 94 is reached with an `msg` object with less than 5 path parts (the error state), the failure is visible by the index out of bounds exception.

[1] `https://github.com/ETCLabs/LuminosusEosEdition/issues/17`