TAV - Report

Project: Luminosus

Johannes Schneider and Tim Henning

November 13, 2017

0 Application Selection

0.1 Introduction

Extending the functionality of a lighting desk can be a difficult and time consuming task. To make it more easy 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.

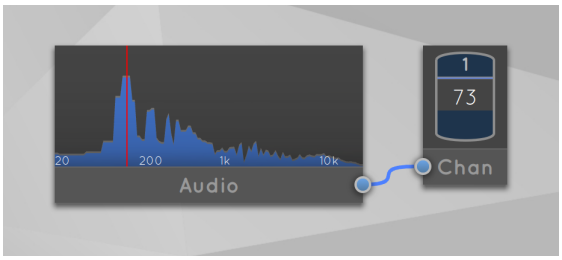

Figure 1: Luminosus Example: Controlling the brightness of channel 73 by the current audio level

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

The programming language used is C++ 11 with syntax additions by Qt. The user interface part uses the descriptive language QML 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).

Available artifacts are the source code, documentation, changelog, issue tracker, manual 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

In the following section some random failures of the application shall be introduced and analyzed what can be learned from them.

Issue #2 Connection to Eos stopped working: After a while the network connection of the software to the lighting console stopped working. The user was not able to continue using the application for the show even after restarting everything.

While the fault is not yet known the issue is probably complexity related. Many different devices were involved and it is not clear whether the problem was caused by this software or a different failing part in the system.

Issue #3 Faders not updated: The user described an issue where the fader positions were not updated correctly after switching the page of faders.

The issue was caused by a single missing line of code that would reset the positions of all faders to 0 before requesting and setting the new positions. Thus the issue can be categorized as preventable as the fault could have been found by unit tests that cover all branches of the code.

Issue #4 Fullscreen on iOS not working: The issues states that the fullscreen button is not working correctly on the iOS platform. The failure is visible as too large UI elements that render the application unusable after clicking on the fullscreen button.

On the one hand the fault can be categorized as preventable since it could be detected by a simple unit test that compares the scale property before and after the operation. On the other hand it is also a type of intelligent fault since the real problem was the pure existence of a fullscreen button on a mobile operating system. 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 that is in this case not very useful.

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 kind 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 till the begin of this V&V project.

The process of V&V does not end till 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 like 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

Peer Review The peer review helped understanding the code because it was possible to discuss the reviewed code directly with the author. We were able to improve the general code quality and comments by pointing out parts that were difficult to understand.

The previously known bugs were found and we could even find a case where the description of a method did not match its content.

A good peer review is organized a few days before so that everyone can get into the code. Therefore the discussion can start right away. It also helps to question even minor things so that the author can either explain his thoughts or comes to really think of it for the first time.

Tools A tool that is well suited for human-based peer reviews is the review function of GitHub (see fig. 2). It allows for example to mark persons which have to review a pull request before it can be merged. Comments can be made for each line of code individually and the author can respond to them directly. It can also integrate various different tools and incorporate dynamical tests of continuous integration system like Travis CI, too.

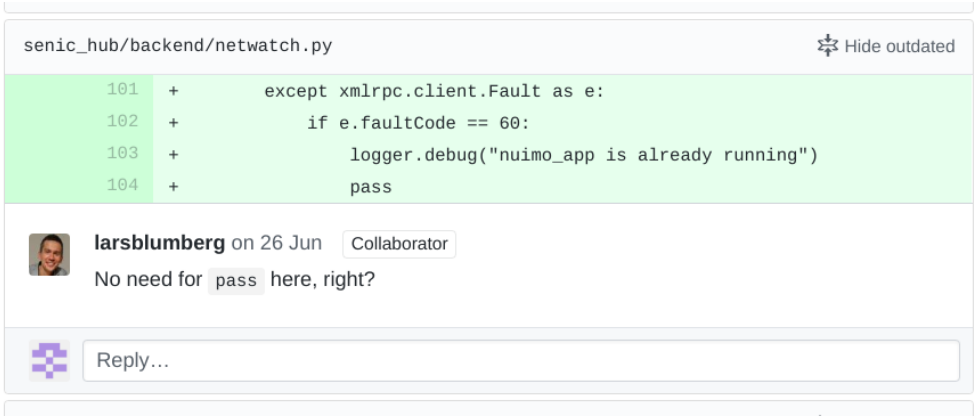

Figure 2: A comment in the GitHub review tool

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 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 the code understanding a lot.

1.2 Automated Static Analysis

Evaluation of Tools We used Clang Static Analyzer and CppCheck for this project. Both tools are well suited for C++ code and are supported by the IDE (Qt Creator).

The detected faults are intuitively shown next to the corresponding line in the code.

On the in 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 such making it harder to detect the real faults (see fig. 4).

Other than Clang, CppCheck 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. 3).

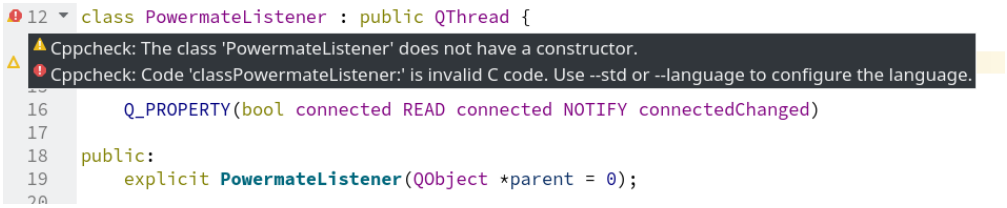

Figure 3: Bugs in CppCheck result in false positives

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.

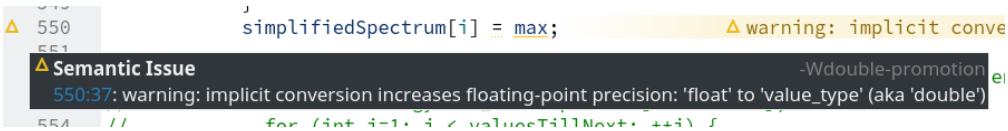

Figure 4: Clang warns of increase in precision of floating point number

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 positive 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.

1.3 Interaction between Human-Based and Automated SA

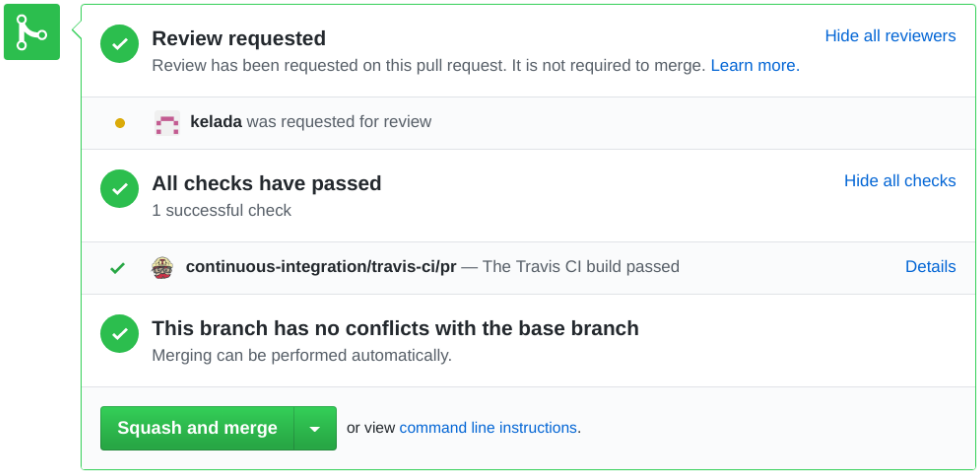

Figure 5: Combination of ASA and human-based reviews in GitHub

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.

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.

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 the 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. 5.