Continuous software quality analysis for the ATLAS experiment

A Washbrook, on behalf of the ATLAS collaboration

SUPA, School of Physics and Astronomy, The University of Edinburgh, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh, EH9 3FD, United Kingdom

E-mail: awashbro@ph.ed.ac.uk

Abstract.

The software for the ATLAS experiment on the Large Hadron Collider at CERN has evolved over many years to meet the demands of Monte Carlo simulation, particle detector reconstruction and data analysis. At present over 3.8 million lines of C++ code (and close to 6 million total lines of code) are maintained by an active worldwide developer community. To run the experiment software efficiently it is essential to maintain a high level of software quality standards. The methods proposed to improve software quality practices by incorporating checks into the new ATLAS software build infrastructure will be discussed.

1. Introduction

The regular application of software quality tools in large collaborative projects is required to keep software defects to an acceptable level. Defects such as redundant code paths and errors of omission are often viewed as minor transgressions from the perspective of the individual developer and may not be flagged by compilers as part of the software build process. However if left unchecked the accumulation of defects invariably results in performance degradation at scale and leads to problems with the long-term sustainability of the software.

A wide range of software quality tools are used by the developer community in the ATLAS collaboration to identify, track and resolve any defects in their code. A limited selection of static code analysis tools (namely `cppcheck` [2] and `Coverity` [3]) are used to perform periodic scans across main development branches to provide scheduled notifications to code maintainers on any defects that require urgent resolution. More general code quality indicators, coverage testing tools and code formatting checkers are also used as part of the development and build process.

Although software quality tools have proven effective for defect identification, there remains a non-trivial sociological challenge to resolve defects in a timely manner. High priority defects (such as the existence of uninitialised variables or potential sources of memory leaks) are usually dealt with promptly, but lower priority defects can often remain unresolved. This is especially problematic in sections of code subjectively considered to be non-critical by maintainers and leads to a backlog of legacy defects that are likely to be unaddressed. The issue is compounded by areas of code where responsibility and provenance are unrecorded and prevalent where developer effort is reorganised or not retained.

To provide a high level of code quality a two-fold approach is therefore required. Firstly, defects over a certain age threshold should be periodically re-evaluated on their validity and

importance and be disregarded if their impact is viewed as marginal. Secondly, to reduce defect accumulation it is necessary to identify and address defects *before* they are introduced into a widely used software release. It is this latter approach that is of interest in the studies described here.

2. Continuous Software Quality Evaluation

Recent wholesale changes to the ATLAS software build infrastructure have provided an ideal opportunity to apply software quality evaluation as an integral part of the development workflow. Source code control is now provided by `git`, thereby enabling the use of the `Gitlab` [4] repository manager to provide a framework for code review (or *merge requests*) before any modifications are included in the main development branches. Any submitted merge request (or subsequent code modifications within the same merge request) triggers an ordered sequence of build correctness checking and unit testing jobs to be run by a continuous integration (CI) service (`Jenkins` [5]). A test result summary is then pushed to the `Gitlab` merge request discussion via the `Gitlab` Application Programming Interface (API). All test results are monitored by a dedicated rota of code review shifters to validate changes and to enforce experiment software guidelines before a merge request is accepted.

By extension, this framework can be leveraged to include the preemptive checking of defects by the addition of software quality tests within the CI test chain. Any new defects left unresolved after the code review can be attributed to the developer initiating the request and thus an audit trail of known defects is automatically provided. Given the high volume of merge requests it is necessary for CI-based software quality tests to be quick (typically less than 5 minutes) to reduce load on the build systems and to provide a reasonable response time for input into the review process. It is therefore preferable to perform checks *only on code directly affected by any changes in a given merge request*. The results from software quality tools should remain open to interpretation from code reviewers to avoid erroneous or trivial quality concerns blocking urgent merge requests from being accepted.

3. Feasibility Studies

The feasibility of continuous software quality evaluation was tested by applying an example static code analysis tool to run within the ATLAS CI framework.

3.1. Development Testbed

The inclusion of software quality CI tests without full validation was considered to be too disruptive to the ongoing review process. This motivated the construction of a test development environment hosted at multiple sites and isolated from the operational software build infrastructure.

The use of `Docker` [6] containers enabled the rapid emulation of key services in the ATLAS software build infrastructure, thereby creating a sandbox to develop new CI tests and to explore how to interpret and report software quality test results. An illustration of the distributed test-bed can be seen in Figure 1.

The merge request workflow was simulated by importing a snapshot of the ATLAS software repository. Mock merge requests were then generated to inject a range of code modification patterns that could test the response of the software quality tools and reporting mechanisms. The mock requests generated trivial changes (such as the addition of a comment in the code) and modifications that were more representative of developer contributions. Parallel development and reproducible results across test-bed instances was made possible by placing the deployed container instances under version control in a registry hosted in a common `Gitlab` project area. A stable version of the service and CI test configuration states could then be tagged for inclusion into the main ATLAS software build infrastructure.

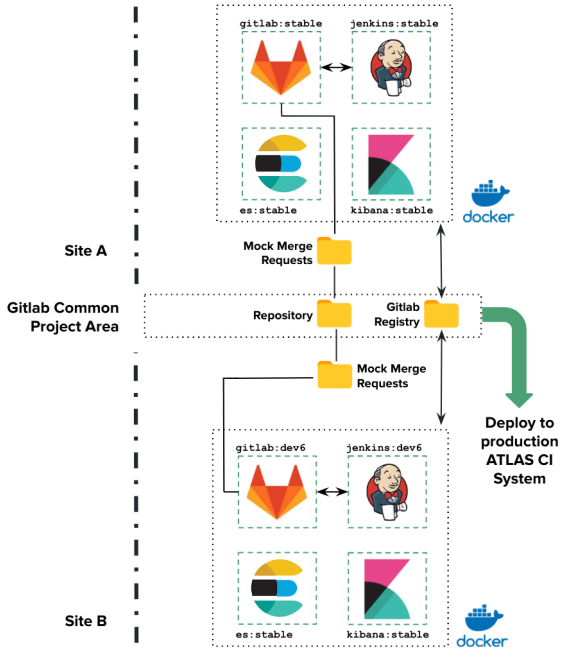

Figure 1. Illustration of the distributed container testbed used to develop CI software quality tests.

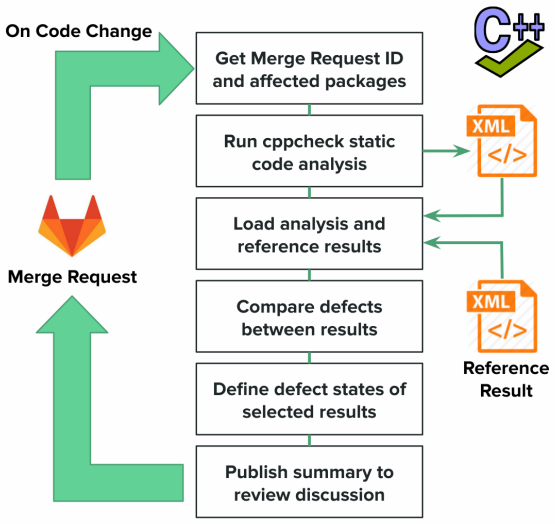

Figure 2. Illustrated workflow for the `cppcheck` CI test job.

3.2. Defect Identification and Reporting

The software quality application `cppcheck` was used to generate a representative set of defect results within a CI test job. This application was chosen because of the relatively quick time to generate results from a large code base (approximately 40 minutes for the entire ATLAS release) and for the minimal effort needed to apply checks to a selected subset of the code base.

The illustrated workflow is shown in Figure 2. It can be seen from the figure that payload checking is only small part of the overall workflow. The merge request data are first retrieved before the list of files affected by the proposed modification is extracted and passed to `cppcheck` for processing. This filtering step considerably reduced the execution time of `cppcheck` compared to running over the full development branch by default. It was reasonably expected that `cppcheck` would take less than a minute to run when only a small amount of code modifications were made.

In earlier iterations of this model it was anticipated that the results could be passed directly to the merge request for review. However, it was decided that this would pollute the review discussion. For example, a merge request may contain minor changes to numerous sections of code containing historical defects unrelated to the modification under review. It was therefore more beneficial to provide shorter feedback indicating a state change. Defects are either *introduced*, *removed* or remain *unresolved* against a reference result generated from the main development branch. The role of the software quality CI test after results gathering was therefore to identify and label differences between the merge request and reference result, and to publish a summary report of the state changes in the merge request discussion.

The summary report was constructed in a markdown syntax which could be parsed in a consistent format by `Gitlab`. An example report is shown in Figure 3. Some care was taken in applying defect prioritisation and in truncating the report to a maximum length where necessary. Any defects introduced by the proposed code modification were given more prominence to

motivate action to be taken and to halt further increases to the overall number of defects.

Cppcheck Results

No new defects were introduced by this merge request

14 defects unresolved in **files** changed by this merge request

- HLT/Trigger/TrigMonitoring/TrigOnlineMonitor/src/TrigALFAROBMonitor.cxx

Severity	Defect	Location
WARNING	Member variable 'TrigALFAROBMonitor::m_hist_goodData' is not initialized in the ..	TrigALFAROBMonitor.cxx:65
WARNING	Member variable 'TrigALFAROBMonitor::m_hist_goodDataLB15' is not initialized in ..	TrigALFAROBMonitor.cxx:65
WARNING	Member variable 'TrigALFAROBMonitor::m_hist_goodDataLB18' is not initialized in ..	TrigALFAROBMonitor.cxx:65
WARNING	Member variable 'TrigALFAROBMonitor::m_hist_PosDetector' is not initialized in t..	TrigALFAROBMonitor.cxx:65

(and **10** other defects of type WARNING)

Figure 3. Sample formatted report generated by the `cppcheck` CI test job.

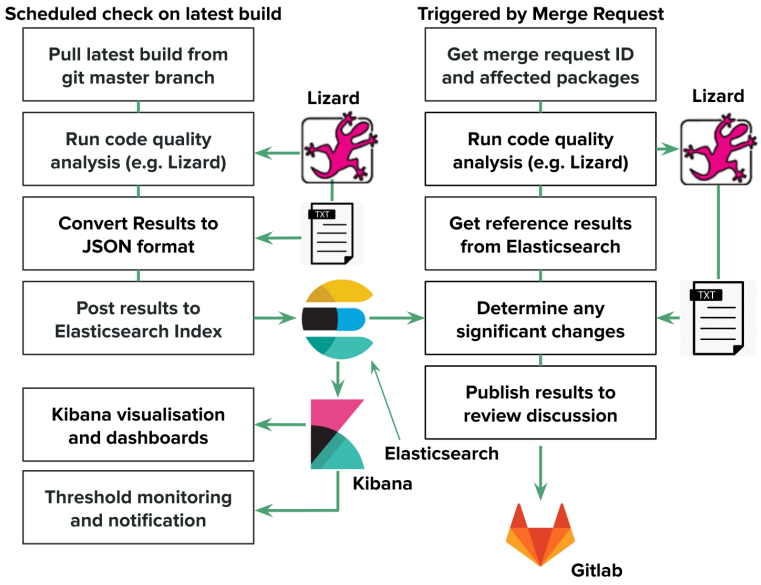

Figure 4. Illustrated workflow of the code quality trend analysis CI test job.

3.3. Trend Analysis

Defect identification is only one aspect of an overall software quality evaluation strategy. Code quality indicators can also be used to signify the quality and relative maintainability of a given piece of software. Indicators can be relatively simple metrics (e.g. the number of lines of code with comments) or more complex interpretations (e.g. function decision depth [8]).

Indicator values taken in isolation add limited information to the code review process. For example, a file with a reported *cyclomatic complexity* value (i.e. the number of possible logical

paths through a section of code) needs to be placed into context by supplementing the value with trend information (e.g. whether this has significantly changed as a consequence of a code change) and a threshold value given at which affirmative action should be taken.

The application of code quality indicators within a CI framework was tested using the `Lizard` [7] code quality tool, which provided a relatively fast method of generating several key quality indicator metric values for code contained in the main development branch. The CI test workflow was similar to the `cppcheck` feasibility test described in Section 3.2 and is shown in Figure 4.

Reference results were converted into a `JSON` format and stored in a `Elasticsearch` [9] index. `Kibana` [9] was then applied to the curated dataset to provide a dashboard visualisation of indicator trends across successive versions of the code.

The response of this proposed workflow was tested by storing the code quality indicator results for each defined `C++` function from several historical snapshots of the main development branch (where each branch contains over 219,000 functions). In the most recent snapshot a highly branched code section was inserted into a sample function to determine if changes in cyclomatic complexity could be easily identified through `Elasticsearch` queries and `Kibana` dashboards.

Despite the low latency found in isolating potential quality issues across a large code base, some care will be needed managing the size of `Elasticsearch` indices if the number of stored snapshots and code quality indicators is increased.

4. Future Improvements

Once the evaluation and deployment of a CI-based software quality check is completed, experience will be gathered from code review shifters and developers to help optimise testing coverage and results presentation. Some areas of potential development are described below.

4.1. Alternative Code Analysis Tools

The CI software quality test workflow was designed to have minimal dependency on the underlying static code quality application. Further development of the workflow could adopt a plugin approach to accommodate results gathering and comparison from other tools. `Coverity` is now being evaluated to provide more comprehensive code analysis coverage. As before, only the modifications covered by an individual merge request should be considered for analysis to reduce the overall time of the CI-based test. An incremental build method is one such more lightweight method being considered.

4.2. Triage Methods

New processes to assign defect resolution action through *reviewer-led* triage are being tested to help reduce the overall number of defects in the main development branches. Any new approach would need to naturally fit within the code review workflow. One such implementation provides a connection service between the `Coverity` triage store and `Gitlab`. For example, `Gitlab` *webhooks* could monitor triage actions initiated in the merge request discussion. Reviewer requests (e.g. assigning a particular defect to a responsible group) could be converted into a API call to the `Coverity` triage store via a third party service. An immediate benefit would be the suppression of unimportant or incorrectly identified defects at source to avoid repetition of future code review effort.

5. Outlook

The feasibility tests described above have demonstrated that it is possible to provide continuous software quality evaluation by the inclusion of lightweight tests in the new code review process.

Before significant effort is invested in providing additional features there needs to be an assessment of whether related, but more established, tools and services that provide similar functionality are preferable. The balance between the benefits of enhanced test coverage to the additional overhead of deployment (and user education) of a service external to the chosen ATLAS build infrastructure tools would have to be addressed.

The reduction of software defects can be helped further by developer engagement. For example, software quality considerations can be applied by promoting analysis tools integrated into code development environments. This will help catch defects before the code is committed for review.

Finally, the recent migration from legacy and bespoke elements in the build process has meant that the development of software quality CI tests has been independent of the underlying code. With growing experience the practices described above could be applied outside of the ATLAS experiment to help improve the code quality in other software projects in the high energy physics community and elsewhere.

References

[1] ATLAS Collaboration, The ATLAS Experiment at the CERN Large Hadron Collider, 621 JINST 3 (2008) S08003.

[2] Cppcheck - A tool for static C/C++ code analysis, "Cppcheck" [software], version 1.80, 2017. Available from http://cppcheck.sourceforge.net/ [accessed 2017-10-01]

[3] Synopsys Static Analysis, "Coverity" [software], version 8.7.1, 2017. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html [accessed 2017-10-01]

[4] Gitlab Community Edition [software], version 9.2.4, 2017. Available from https://packages.gitlab.com/gitlab/gitlab-ce [accessed 2017-10-01]

[5] Jenkins open source automation server [software], version 2.46.2, 2017. Available from https://jenkins.io/download/ [accessed 2017-10-01]

[6] Docker [software], version 1.12.6-11, 2017. Available from https://www.docker.com/get-docker [accessed 2017-10-01]

[7] Lizard [software], Available from https://github.com/terryyin/lizard [accessed 2017-10-01]

[8] Software Quality Metrics Resources [Online]. Available from https://www.imagix.com/links/software-metrics.html [accessed 2017-10-01]

[9] Elastic Open Source Search and Analytics [software], version 5.5.1, 2017. Available from https://www.elastic.co/downloads [accessed 2017-10-01]