Incremental Software Quality Analysis for Embedded Systems

Benjamin Hummel, Thomas Kinnen
CQSE GmbH
Garching b. München, Germany
hummel@cqse.eu

***Abstract*—Analysis tools play an important role in the development of high-quality software systems. For all aspects of software quality, including safety, security, maintainability or portability, there are tools that can automatically detect a large number of quality defects. While such tools are not sufficient to fully check a system's quality, tool support does help to reduce the number of defects found late in the development process and relieves developers and reviewers from performing repetitive and schematic checks manually. Existing analysis tools typically run in batch mode and analyze the entire system at once. Depending on the system's size, this can easily take hours, even when integrated in a continuous build process. While this is no problem during release preparation or at quality gates, the single developer usually needs a shorter feedback cycle to benefit from analysis results and react to the defects found. As a solution to this problem, we propose incremental quality analysis, which allows updating known quality data of a system based on the changes applied to the system's code files. As a developer typically changes only a couple of files at once, this allows processing the changes within seconds and providing the developer with immediate feedback on every change. This paper explains the technology behind incremental quality analysis, provides benchmarks that document the performance gain of several orders of magnitude, and presents promising results from the practical application of this approach.**

Keywords—software quality analysis; incremental analysis; continuous quality control

I. Introduction

Software quality is a term that is used with many different meanings. For some, it denotes only the customer visible quality, such as fulfillment of functional requirements or usability, while for developers, quality often means the quality of the source code, including aspects like readability and portability. In practice, all of these views are connected. While different quality aspects sometimes are described as being independent, they affect each other. For a complex system with low readability, it will be really hard to maintain high user-visible quality in the long run. Low security can also cause safety problems for some systems, if the security problems can cause critical software failures. Thus, software quality should be addressed as a whole, although the individual aspects as defined in ISO/IEC 9126 resp. ISO/IEC 25010 can help to structure an approach towards software quality improvement.

Over the previous decades, software has become more and more important for our economy and our every day's life. With the advent of ever more complex embedded systems and trends such as the *Internet of Things*, software quality no longer only affects the digital world, but has an immediate impact on the physical world as well. At the same time, the steady growth of software systems, both in number of functions and complexity, makes it increasingly difficult to understand a system's quality, as no single individual has full knowledge of all details and side-effects of the code base.

One part of the solution, that is often proposed, is the use of automated tools that check a system's various quality aspects. While these tools can never fully replace manual quality assurance (e.g. reviews), they can release engineers from boring and repetitive tasks, allowing them to spend their time on more relevant quality improvement tasks. Additionally, tools can easily provide an overview, even on large code bases, and perform certain analyses, such as finding duplicated source code, that are nearly impossible to perform manually.

A major problem with many quality analysis tools is the long feedback cycle between changes to the source code and the notification of the developers. With analysis times of many hours for larger systems, analysis is often only performed every night. This way, developers are informed about quality defects, when they are often already working on a different feature. Especially with the advent of agile practices in embedded development, this delayed feedback counters many of the benefits to be gained from small iterations and continuous integration.

Our solution to this problem is to speed up the analysis process by using *incremental analysis*. The goal is to build analyses that can incorporate changes to the code base, when they are committed by the developers. This way, not the entire system has to be reanalyzed, but only the small part that has been changed. This paper completes earlier publications, where we described single analyses [1] and the overall architecture of a system for incremental analysis [2], by describing a complete implementation and results of its practical application.

II. Incremental Analysis

A. General Approach

Studies [2] show, that the analysis of a large software system can easily take hours, while the typical commit (change) to a code base consist of only a couple of files. Incremental analysis leverages this fact and continuously keeps an up-to-date status of the code's quality in a database. Using the information from this database, the quality status can be updated by only analyzing the changed files and possibly a couple of related files. If this update can be performed efficiently, the database reflects the quality of the code base with only a few seconds delay.

This approach sounds simple in the first place and actually is for many quality analyses. Every analysis that can be applied independently to individual files can be used incrementally nearly without adaption. This category is quite large and includes many typical "guideline checks" (e.g. formatting, limiting usage of programming constructs, naming conventions) and also more advanced analyses, such as contextual comment analysis [3] or dataflow-based intra-method analysis. However, there are a number of analyses and challenges, which cannot be applied incrementally in such an obvious manner. Examples for these problems and possible solutions are described in the following.

B. Challenges in Incremental Analysis

Clone detection attempts to find duplicated source code fragments, typically created by copy&paste. The problem of these duplicates is the unnecessary increase in code size, which often increases the effort needed for inspections and testing, and the risk of applying changes and bug fixes only to one instance of the duplicate, leading to inconsistent or even erroneous behavior [4]. The problem with clone detection is that a changed file might potentially contain a copied code fragment from every other file in the code base, whether changed or not. Thus, the analysis cannot focus on only the changed files. The solution is to manage an additional data structure that allows to quickly find candidate files from which the changed files might contain clones and limit detection to this slightly extended file set. The details of this data structure are given in [1].

Dependencies between files and types of the system are required for many different analyses. One example is the long list of dependency metrics, another a conformity analysis between the intended architecture of a system and its actual implementation in the source code [5]. The key to keeping dependency information up to date is the observation that the outgoing dependencies of a file, i.e. the list of files a file depends on, can be determined by only looking at this single file. Thus, updating outgoing dependencies can be performed easily for all changed files. What makes this a bit more involved is that for many analyses using the dependencies, we also need information on the incoming dependencies (which files depend on this one) and information not on the file but on the type level. However, all of this information (including a mapping between files and types) can be updated incrementally based on changes of the outgoing dependencies.

Aggregation of quality metrics is an important aspect of any analysis, as we are typically not interested only in the quality of single files, but also on an aggregate for components of the system or the entire code base. Obviously, every change to a file can change its metric values and thus affect the overall aggregates. By keeping the aggregation relation (typically a tree following the directory or namespace hierarchy) explicit, a change of a metric value for a single file (a leaf) can be distributed recursively to the inner nodes leading to the root of the tree (representing the overall aggregate).

Tracking of file renames, moves and copies might sound like an unrelated problem at first. However, many analyses can benefit from information about these file-level operations and accurate information also eases interpretation of analysis results. In theory, this information should already be recorded in the version control system (VCS), but studies [6] show that for up to 39% of the files in a VCS this information is incomplete. To detect file moves and renames, it is sufficient to inspect all files within a commit. Interestingly, this analysis becomes easier when applied incrementally as compared to nightly, as this way the candidate set in each commit is much smaller than looking at the aggregated changes over a full day. However, similar to clone detection, for a copy operation all other files in the system are potential copy sources. The solution we use applies similar data structures as the incremental clone detection to circumvent this problem. The details of the approach are described in [6].

Tracking of quality issues links issues from the previous version to the current version. This allows to identify, which issues have been removed or added by a commit. Also, meta information attached to quality issues, such as the name of a developer responsible for removing it, or manual notes about the criticality of the issue, can be reliably kept attached to an issue even when moving the code to a different location. To work reliably, even in the context of complex refactorings, we utilize information on file moves and renames and apply a hash-based tracking algorithm (as described in [7]) on issues that are not in the same place after applying the file-level operations.

C. Implementation

Based on these algorithms for incremental quality analysis, we built Teamscale[1] [8] as an implementation of our approach. Besides the analysis algorithms, Teamscale also deals with the technical aspects required for reading changes and actual content from a version control system, as well as the configuration and scheduling of the individual analysis steps. Using an incremental storage schema, not only the quality data of the most recent revision of the code is available, but the data of every single analyzed commit can be accessed. This can be helpful to better understand the root-cause for changes in the quality of the code, but also supports various different

[1] http://www.teamscale.com

TABLE I. QUANTITATIVE RESULTS: ANALYSIS TIME STATISTICS

System Name	Programming Language	Lines of Code (in Head)	Number of Files (in Head)	Number of Analyzed Revisions	Average Files touched per Commit	Overall Analysis Time (seconds)	Average Time per Commit (seconds)
Agora	JavaScript	25,646	261	2,919	9.0	10867	3.7
Doxygen	C++	323,675	465	1,119	17.2	39214	35.0
Eclipse Code Recommenders	Java	71,507	605	2,387	26.1	21648	9.1
Jenkins	Java	162,659	1,024	11,691	19.4	132889	11.3
Unknown Horizons	Python	78,589	467	9,591	66.6	285187	29.7

strategies for dealing with quality defects, for example to focus mostly on issues introduced since a certain point in time (such as a release). To allow inspection and interpretation of the results, the data can be accessed in various ways via a web interface. For the developers there is also integration with the development environment (IDE) to display quality issues directly in the edited code.

III. RESULTS

Based on the implementation in Teamscale, we performed various experiments with the incremental approach.

A. *Quantitative Results*

First results have been published in [2], showing an average analysis time between 1 and 3 seconds per commit. This prototypical version, however, did not yet include the full set of analyses required for practical application. With Teamscale, the set of analyses now covers a wide range that allows its application in quality control [9]. We report results on 5 Open Source systems[2] in Table 1. These systems range between 250 and 1000 files and have between 25,000 and 323,000 code lines. The number of revisions analyzed was between 1,000 and 11,500. Interestingly, the average number of files touched by each commit is between 9 and 66, which is way larger than in our earlier study [2]. We assume this to be caused by all systems being maintained in Git (compared to SVN with the earlier study). In Git, most of the projects use many smaller commits in feature branches, but the main line of development only sees larger merge commits of these feature branches. Especially, the history of *Unknown Horizons* contains many large merges. Due to the larger number of files and also due to the increased set of analyses, the average analysis time required for each commit is between 4 and 35 seconds. This is slower than in our previous study, but still fast enough to provide a developer with timely feedback on a commit.

B. *Qualitative Results*

With the goal of improved analysis performance, the raw analysis times are surely of interest. For the practical applicability, however, they are only a small part of the solution. Using the implementation in Teamscale, we deployed incremental analysis at the insurance company *Lebensversicherung von 1871* [8], the automotive OEM *Audi* [9], and a couple of other partners. While batch analysis tools that run in a nightly build often only have a one-time effect that diminishes after some time [10], in all these projects the incremental analysis had a lasting effect. An example of the change in the number of quality issues in a code base after the introduction of Teamscale is shown in Figure 1. We observed developers to be more aware of quality issues in their code and to often quickly remove new quality issues right after committing changes. Interviews with the developers revealed that the increased awareness was caused both by the rapid feedback, so the developer was still working on the code when the notification of new issues arrived, and the tight integration with the IDE, which allows developers to get information about quality issues without leaving their primary tool. In combination, this additional and timely information was enough motivation to remove any quality issues in "their" code.

IV. DISCUSSION

The implementation of the approach in Teamscale and our results clearly show the feasibility and applicability of incremental analysis for software quality control. The quantitative results demonstrate that near real-time results are possible in a realistically sized development setup and the quantitative results imply that the increased responsiveness and integration of the analysis has an actual impact on the quality of the code base. However, there are two questions, which came up during our developer interviews, that point towards further directions of improvement.

The most frequent question was about pre-commit analysis. Given the additional transparency from the analysis of every single commit, the developers would even prefer to get notified of quality issues before committing their changes. This would ensure that only clean code is checked into the version control system. As with the per-commit analysis, we would expect a developer to usually touch only a couple of files. So the incremental analysis algorithms might be used against a known baseline and the changed files. The only challenge is in synchronizing the quality status on the server and on the developer's machine. One possible solution for this would be to transfer the changed files to a central server and perform the analysis there.

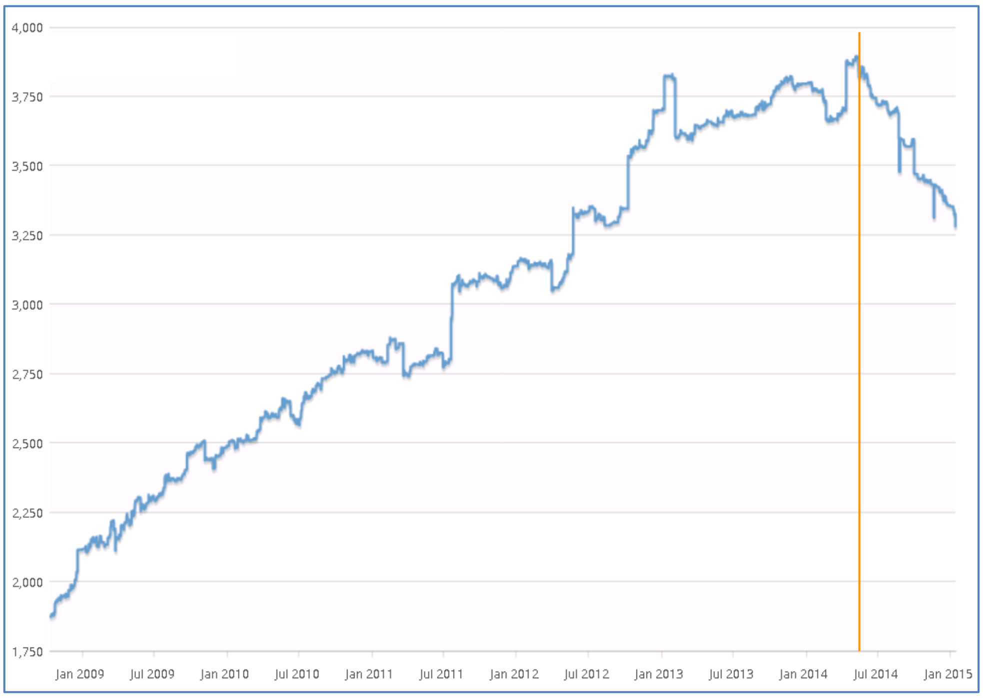
The second issue is related to the advent of distributed version control systems that allow inexpensive setup and

[2] The systems have been selected to cover multiple different programing languages. Their code is available at the following URLs:

- https://github.com/softwerkskammer/Agora
- http://www.doxygen.org/
- http://eclipse.org/recommenders/
- http://jenkins-ci.org/
- http://www.unknown-horizons.org/

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "Evo-Con, 01IS12034A". The responsibility for this article lies with the authors.

Fig. 1. Example of the development of the number of quality defects after the introduction of incremental quality analysis to a software project. The introduction of the tool is marked by the vertical line.



management of development branches. As a result, many teams develop new features on separate (feature) branches and integrate the branch only after the feature has been completed. The problem for an analysis tool is that the changes in the branch are not visible until the branches are merged, which can be a long time frame and counters the idea of real-time feedback. The solution to manually set up an analysis for every single branch is also not feasible, as often branches are frequently created and destroyed. This is a problem that is shared with traditional batch analysis tools, which also usually only run on a single or a few fixed branches. With incremental analysis, however, is would be feasible to analyze every single commit in every single branch and thus provide the quality status for every intermediate state of the system. The open challenge for this approach is the organization of the analysis data, which has to be stored for every single branch, and the processing of merge commits, where the quality status of multiple branches must be merged. An especially challenging question in this context is how to deal with tracking of quality issues. For example, issues created in a branch might not be counted as new in a merge commit, but an engineer only watching the main branch might still be interested in getting notified of the “new” issues.

V. Conclusion

This paper explained the concept of incremental quality analysis as a solution to the long feedback loops created by conventional quality analysis tools running in batch mode. We explained, how many of the well-known analysis algorithms can be adjusted to the incremental paradigm and demonstrated the feasibility by providing an implementation of these adjusted algorithms. The results show that the analysis times are fast enough to provide immediate feedback to the developers after a change. Additionally, experience with several professional development teams showed that the availability of timely quality data motivates developers to preserve and improve the quality of the software system.

The support for quality analysis of changes prior to their commit to a version control system and better support of branches during development are two directions that promise even further improvements in the tool support for continuous quality control. We plan to pursue both in the context of our future work.

References

[1] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, “Index-based code clone detection: incremental, distributed, scalable.” In Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM’10), 2010.

[2] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, M. Conradt, “A framework for incremental quality analysis of large software systems.”, in Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM’12), 2012.

[3] D. Steidl, B. Hummel, E. Juergens, “Quality analysis of source code comments.” in Proceedings of the 21st IEEE Internation Conference on Program Comprehension (ICPC’13), 2013.

[4] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, “Do Code Clones Matter?” in Proceedings of the 31st International Conference on Software Engineering (ICSE’09), 2009.

[5] M. Feilkas, D. Ratiu, E. Jürgens, “The loss of architectural knowledge during system evolution: an industrial case study.” in Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC’09), 2009.

[6] D. Steidl, B. Hummel, E. Juergens, “Incremental origin analysis of source code files.” In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR’14), 2014.

[7] S. P. Reiss, “Tracking Source Locations.” in Proceedings of the 30th International Conference on Software Engineering (ICSE’08), 2008.

[8] L. Heinemann, B. Hummel, D. Steidl, “Teamscale: software quality control in real-time.“ in Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE’14), 2014.

[9] K. Schüler, R. Trogus, M. Feilkas, T. Kinnen, “Managing product quality in complex software development projects.“ in Proceedings of the Embedded World Conference, 2015. In press.

[10] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, B. Uhink-Mergenthaler, “Continuous Software Quality Control in Practice.”, in Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME’14), 2014.