A Framework for Incremental Quality Analysis of Large Software Systems

Veronika Bauer, Lars Heinemann, Benjamin Hummel
Technische Universität München, Germany
{*bauerv,heineman,hummelb*}*@in.tum.de*

Elmar Juergens
CQSE GmbH, Germany
juergens@cqse.eu

Michael Conradt
Google Germany GmbH
conradt@google.com

***Abstract*—To provide rapid feedback to engineers, software quality analysis must be incremental. However, most existing analyses are either not incremental, or limited to isolated quality characteristics. In practice, this prevents their integration into a uniform quality control approach. In this paper, we present a framework for the incremental and distributed computation of quality characteristics. It is fast enough for real-time analysis of large systems and provides a complete history of analysis results. An evaluation on several open source software systems demonstrates its scalability to large code bases under active development.**

I. Introduction

If no counter-measures are taken, the code quality of long-lived software systems is likely to decay over time [1], [2], [3]. In consequence, maintenance efforts increase—causing higher costs and reduced agility in fixing bugs or implementing new features. One approach to counter software decay is the measurement and inspection of key quality characteristics. If performed on a continuous basis, this allows to discover quality defects early, while their removal is still inexpensive. In this paper, we refer to such approaches as *continuous quality control* [4], [5].

A plethora of automated software quality analyses have been proposed for this purpose, including, *e. g.*, clone detection [6], architecture conformance analysis [7], [8], bug pattern detection [9], or code complexity metrics [10].

Their results are used both by developers and quality engineers. Developers react to analysis findings in their code and rectify problems. To be effective, they require rapid feedback to newly introduced quality defects. Quality engineers inspect left-over findings to identify and address root causes. For effective root-cause analysis, they need to be able to quickly identify changes that introduced problems. To this end, they require analysis results for each version in the version history of the software system. To fulfill both requirements, analysis results must be updated quickly when code changes. In practice, we can achieve this only with incremental approaches. This applies especially for companies with a large code basis and high code churn, like Google, where in 2011 more than 5000 developers committed more than 20 changes per minute[1].

[1]http://google-engtools.blogspot.de/2011/05/welcome-to-google-engineering-tools.html, accessed 2012-04-02

Most quality analysis approaches, however, do not work incrementally. Instead, they (re-)process the entire software. For real-world systems, this often takes too long to run for each individual change. Instead, they are run on a nightly or weekly basis. This significantly delays feedback for developers, thus increasing accidental complexity involved in the rectification of the detected quality defects. It also makes root cause analysis harder for quality engineers, since the number of changes between two analysis runs, each of which could have caused a certain change to the quality indicators, can be large.

Several research groups have recognized this problem independently and have proposed incremental algorithms for individual quality analyses [11], [12], [13]. However, they focus on a single quality aspect in isolation. They lack an underlying framework, forcing developers of new incremental analyses to re-implement common functionality, such as version control synchronization, filtering, visualization and historization of analysis results. As a consequence, we lack tool support that can compute a collection of quality analyses incrementally, as required for continuous quality control.

Problem: To best support continuous quality control, we need incremental analyses of a comprehensive set of quality characteristics. Existing incremental analyses, however, are developed in isolation of each other. To integrate them into a coherent tool platform required for quality control, we need a general framework on which to build incremental quality analyses.

Contribution: In this paper, we present a framework for the incremental, distributable computation of quality characteristics. Through it, even complex quality indicators (such as system-wide cloning) can be computed in real-time for every commit to a repository to provide rapid feedback for developers. Furthermore, it stores results for the entire version history, to support quality engineers during root cause analysis. We present the results of an evaluation that demonstrates its scalability to large real-world software under active development. Our framework is available as part of the Open Source quality analysis toolkit ConQAT[2].

II. Tool Requirements

This section details the requirements that drove the development of our tool support for continuous quality control.

[2]http://www.conqat.org/

978-1-4673-2312-3/12/$31.00

They are based on experience the CQSE GmbH gathered performing software quality control in practice.

A. Background

The CQSE GmbH is a consulting company specialized in software quality analysis and assessment. It supports its customers by introducing and performing software quality control for their own development or for their contractors. The tool requirements for developers and quality engineers arise from this context.

B. Rapid Developer Feedback

Quality control involves software developers. They react to new analysis findings in their code and rectify them. To be effective, these findings should be available to developers as early as possible, ideally immediately after committing their code to the version control system[3].

If several minutes or even hours pass between introduction of a quality defect and notification, the developer will probably already be working on a different task and needs to perform a context switch. If several days or weeks pass, he will probably need to get re-accustomed with the affected code. Changes to it are likely to require additional testing effort, since quality assurance has already inspected the original modifications. New changes thus need additional quality assurance efforts. In a nutshell, increasing the time between introduction and notification increases the accidental complexity involved in rectifying quality problems.

For rapid feedback, we require analysis tools that update their results to code repository changes immediately.

C. Historization

Quality control also involves quality engineers, who periodically inspect changes to quality indicators. One important quality-engineering task is to understand the root causes of quality problems. As long as they are not rectified, new code is likely to contain similar problems.

Root cause analysis involves manual inspection of version control logs to understand how each individual change in an artifact's version history affected the quality characteristic under analysis. When quality control is introduced for large existing systems, typically a large number of quality issues are found. From our experience, it is essential to be able to tell old issues (possibly stemming from a different development team from the previous contractor) from new ones, for which developers more readily feel responsible. In consequence, to support root cause analysis, we require analysis tools to store results for each revision of the software in the version history.

[3]Continuous analysis during typing could provide even quicker feedback. We analyze commits, however, since we are only interested in logically complete changes, not incomplete intermediate results.

D. Re-computation of Analysis Results

Quality analyses have parameters. Clone detection, for example, takes the minimal clone length as a parameter. As the experience with continuous quality control grows in an organization, parameter values change. For example, we often start with rather conservative settings, *e. g.*, a minimal clone length of 15 statements, to initially limit the number of analysis findings and focus on the most problematic ones. When quality control is established and many of the findings have been cleared up, we can make the tool more sensitive, *e. g.*, reduce the minimal clone length to 10 and then to 7 statements.

Furthermore, quality analysis software is software, too. It is thus also subject to software evolution. Often, improvements to the analysis software affect its results. Additional checks might produce new findings. Better filters could suppress certain types of false positives previously reported.

To update results if the analysis software or its parameters change, tool support must be able to re-compute the quality analyses for each revision in the software history. To be feasible in practice, this re-computation should be possible in a manageable time span.

III. Approach

In this section, we explain our approach for incremental quality analysis. We start by explaining the design principles for our approach, followed by the architecture of the analysis system. Based on this, we describe the algorithms and storage layout used to enable the incremental analysis. For the algorithms we differentiate between *local metrics*, for which only information local to a single class, file, or method is required for its calculation, and *global metrics*, for which changes to one file can potentially affect the metric value of potentially any other file in the system.

A. Design Goals

Our approach evolves around two goals. The first is to split an analysis into smaller (and as far as possible independent) steps. The granularity is the source file level, *i. e.*, it should be possible to perform each analysis step independently for single files or small sets of files, and integrate these individual results into a larger metric database. This allows us to incrementally update analysis results by performing the analysis only on files affected by a change (such as a commit to a version control system) and integrating the results with the existing ones. This allows us to *scale in time*, *e. g.*, when analyzing a history of changes, as at each step we only have to re-compute values for the (typically small) set of changed files instead of for the entire system. Additionally, this allows us to utilize multiple cores of a CPU or many nodes of a compute cluster by distributing the files to be processed. This allows us to *scale in the size* of the analyzed system.

The second goal is to reuse existing quality analysis tools as far as possible. More concretely, we want to use the capabilities of ConQAT [4], our Open Source source code analysis framework, that provides multiple person-years worth of quality analysis functionality. ConQAT offers both a graphical data-flow configuration language and hundreds of individual building blocks for analyses, which we want to reuse. For many of the analyses, our approach provides a generic way of identifying the affected files and integrating the analysis results, while the analysis itself can be performed by the existing code from ConQAT on a reduced set of files.

B. Technical Architecture

Storage System: To support incremental analysis, we need a way to store the history of analysis results and intermediate values. As most of the data we process and store is of a simple key/value structure, we use a key/value store instead of a relational database.

Our approach uses a central storage system that provides access to multiple key/value stores identified by names. It allows storage (put) and deletion of data based on keys. It supports queries based on single keys, key ranges (*i. e.*, return all keys and values for which the keys are lexicographically between provided keys) and key prefixes (*i. e.*, return all keys and values for which a given string is a prefix of the key).

Configuration: The configuration of the analysis system is defined by a set of *triggers*. A trigger describes the individual steps of an analysis, as well as the names of stores accessed. Triggers may be configured as *periodic*, indicating that a job for this trigger is scheduled repeatedly with a fixed time delay. Periodic triggers are used, for example, to poll the version control system for new changes. A simple example of a configuration is shown in Figure 1, where triggers correspond to the gray boxes.

Job Scheduling: The execution of the analysis steps on a set of files is performed in what we call a *job*. A job is the unit of distribution and treated as atomic (*i. e.*, is not parallelized). The input for a job consists of the trigger to be executed and a set of keys to be treated as new, modified, or deleted in these stores (called a *delta*). The output of a job is directly written into one or more other stores. Jobs for non-periodic triggers are scheduled when a delta required as input for the trigger becomes available. After the execution of a job, a delta based on the changes to one or more stores is automatically recorded. Thus, executing a job causes a delta to be created, which in turn may cause another job to execute if any data was changed.

In the configuration shown in Figure 1, a store containing the content of the source files is kept up to date by the periodic trigger A that queries a version control system. Any changes made to this store cause jobs for triggers B and C to be scheduled that update two metrics and persist them in separate stores. A third metric derived from "metric 2" is calculated by trigger D based on the delta produced by jobs for trigger C. This also means that trigger D is only executed if there are changes in "metric 2" at all, which might not be the case for all changes of the file content (*e. g.*, changes in comments do not affect certain metrics). Especially for larger configurations, this saves processing time.

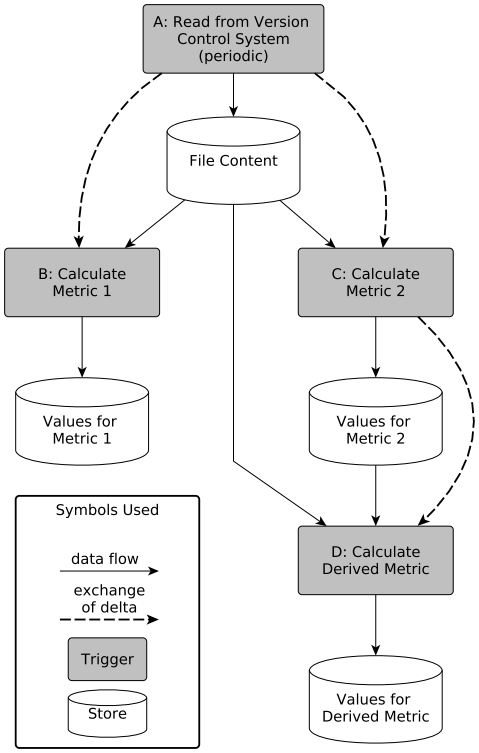

Figure 1. An example analysis configuration.

C. Implementation Details

The evaluation in this paper is executed on a single machine. Thus, we use a simple store implementation based on LevelDB[4], an in-process key/value store.

We built our own simple distributed processing system instead of relying on an existing one (*e. g.*, MapReduce [14]) because the dependencies between the jobs have to be respected. Our system consists of a *scheduler* that coordinates the execution of jobs and one or more *workers* that process these jobs. The workers are implemented as Java programs and thus run as separate threads on the same machine to utilize multi-core processors, or as processes on different machines. Communication between the scheduler and the workers is implemented by reading and writing entries to a dedicated store.

[4]http://code.google.com/p/leveldb/

Besides creating new jobs for periodic triggers and new deltas, and assigning them to free workers, the scheduler also maintains scheduling constraints. One is the maximal size of deltas, configured in the triggers. If the size of the deltas (*i. e.*, the number of keys added, changed, or deleted) exceeds this limit, the delta is split to allow distribution to multiple workers for parallelization. Additionally, the scheduler is aware of revisions, *i. e.*, jobs and deltas can carry information about the revision[5] they belong to. This information is used to avoid data from later revisions being processed before all updates for earlier revisions have been performed. Finally, there are analysis-specific constraints, which may cause certain jobs to be only scheduled when no other job modifies the stores it retrieves data from. This constraint is required, for example, for the calculation of the clone coverage metric described in Section III-E.

D. Incremental Update of Local Metrics

A *local metric* is a metric for which only information local to a single class, file, or method is required for its calculation. More specifically, we call a metric *local*, if for a given set of changed input keys, the metric values for non-changed keys do not have to be updated. To calculate updated metric values for the changed keys, only the values stored for these keys are required. This definition depends on the granularity of our keys, *i. e.*, the unit of a system represented by a key. In our design, a key is in most cases just the path of a file. Thus, all metrics that can be calculated for single files are *local*[6].

For the update of local metrics we just have to calculate the new metric value for each changed file and store the new value. As this file-based calculation is also common in analyses built with ConQAT, these are easy to adapt. We have to adjust the input loading phase to read from a store instead of from the file system, and the output phase to write to another store instead of writing a report. The modular design of ConQAT supports this replacement well.

Many commonly used metrics are local [15], such as lines of code (in any of its variants), depth of nesting, length of program elements (classes, methods, functions), cyclomatic complexity [10], or the per file Halstead volume [16]. Even metrics that do not seem to be local at first sight can be calculated in a local fashion. For example the *fan-out* metric that calculates the number of references to other classes can be calculated locally for most programming languages as all external references can be extracted from the file itself (unless we include references from super classes as well).

[5]We assume that the version control system provides a notion of a revision. Otherwise, timestamps can be used, as we only require revisions to be distinct and increasing.

[6]If we had chosen a smaller granularity, *e. g.*, representing single lines by a key, these metrics would require a more complicated update strategy.

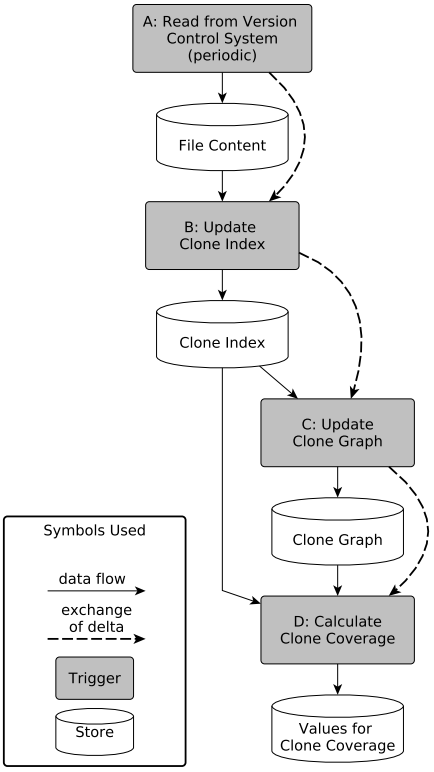

Figure 2. A configuration for calculating clone coverage.

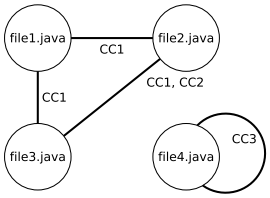

Figure 3. Example of a clone graph.

E. Incremental Update of Global Metrics

Metrics for which changing a single file might affect the metric values of other keys are called *global*. An example of a global metric is *clone coverage*. A *clone* is a piece of code that appears more than once in the source code [17]. The clone coverage for a file is defined as the ratio of statements that are covered by at least one clone [18]. Copying a piece of code from file A to file B will only change file B, while the clone coverage might change for both A and B. Potentially, any changed file can affect the clone coverage of any other file.

The individual steps performed by our approach to calculate the clone coverage are shown as a configuration in Figure 2. When a set of files is changed, the first step is to update a data structure called the *clone index*. The clone index supports incremental updates for single files.

Additionally, it can be mapped to our key/value based storage system and also allows to retrieve the list of all clone classes (sets of related clones) covering a single file. The details of the clone index are described in [13]. Based on the clone index, the clone coverage for a single file can be easily calculated after retrieving all clones for the file. However, from the clone index, we do not know for which files the coverage might have changed. To model this information, we introduce a second data structure, called the *clone graph*. The nodes in this graph represent the files and an (undirected) edge exists between two nodes (files), if there is a clone class with instances in both files. This includes self-loops for clones within a single file. An example of a clone graph for the three clone classes $CC1$ (files $[1, 2, 3]$), $CC2$ (files $[2, 3]$), and $CC3$ (in single file 4) is shown in Figure 3.

To update the clone graph for a changed file, we retrieve its clone classes from the clone index. From these clone classes, we can update the incident edges for this file in the clone graph. This way, the information in the clone graph is incrementally kept up-to-date and allows us to limit the number of files for which the clone coverage has to be calculated. The final step consists of recalculating the clone coverage for all nodes incident to one of the edges in the clone graph whose clone class has changed. The coverage value is determined from the information in the clone index.

The idea of managing a dependency graph that encodes which metric values need to be updated if a file changes, can be carried over to other global metrics, such as fan-in, depth of inheritance tree, or test coverage. For the fan-in metric, this would be the dependency graph. For inheritance depth, this is the inheritance tree (or graph for languages with multiple inheritance). For test coverage, the correspondence to the clone index would store coverage information for each unit test, while the dependency graph would link single test cases to files which can change the outcome of the test, if changed. Calculating such a graph can be performed with so-called safe selective regression testing techniques (see, *e. g.*, [19]). Also the calculation of aggregated metric values (*e. g.*, along the directory hierarchy) is a global value. Here the dependency graph is a directory tree augmented with additional nodes representing directories and transitive update of parent nodes. Contrary to local metrics, for global metrics additional code for managing the dependency graph has to be written to allow incremental analysis. However, our framework provides suitable abstractions to keep the amount of additional code for each metric minimal.

F. Historization

We only store a new metric value for a file and revision, when its value actually changed. This results in a sparse matrix representation as shown in Figure 4[7]. If a file is deleted and thus no valid metric value exists anymore, a special deletion token is stored, illustrated by $< del >$ in the figure. This storage schema allows us to retrieve information from earlier revisions, *e. g.*, for trend and root cause analysis, while keeping the space required linear in the number of file changes, as opposed to a quadratic space requirement for storing every value for every revision (number of files times number of revisions). In addition, we store for each file the value for the current head revision. This allows for fast access to the complete set of metric values of the current head revision, which might be needed for calculating derived metrics. Our implementation uses this historizing storage strategy not only for metric values, but for all stores, including the content, which allows us to access all stored information for any revision.

	Rev. 1	Rev. 2	Rev. 3	Rev. 4	...	Rev. N
file1.java	60					
file2.java	70		50	<del>		
...						
fileN.java	80	90	80			100

Figure 4. Storage organization for the historized metric values.

G. Visualization of Analysis Results

We adapted the visualization back-end of ConQAT to read metric values from a store (instead of processing the values computed during a batch analysis). As ConQAT's presentation is also configurable, we can use its entire range of visualizations, including simple lists and trees, trend charts, and treemaps. A detailed description of the visualization capabilities of ConQAT is beyond the scope of this paper. Please refer to the ConQAT documentation on the ConQAT website for further details.

IV. Evaluation

We evaluated the performance of our incremental approach against the non-incremental version of ConQAT. The reason to limit the comparison to ConQAT and not use other quality analysis tools is that we are interested in the performance gain of the incremental approach and not in a comparison of different analysis engines. Furthermore, as discussed in Section V, none of the existing tools provides incremental analysis over a large range of analysis types.

A. Evaluation Objects

Our evaluation objects are three open source software systems with different sizes and programming languages to avoid a system-specific bias: JabRef[8], ArgoUML[9], and

[7]The actual store is one-dimensional. The entries of the matrix are stored by building storage keys from the file name and the revision.

[8]https://jabref.svn.sourceforge.net/svnroot/jabref/

[9]http://argouml.tigris.org/svn/argouml/

Table I
EVALUATION OBJECTS

System	JabRef	ArgoUML	Chromium
Domain	Reference mgmt.	UML modeling	Web browsing
LOC	130,267	362,605	2,569,784
#Files	626	1,866	13,005
Start rev.	6	5487	8
Head Rev.	3683	19744	105817
Timeframe	2003-10-16 to 2011-10-14	2005-02-11 to 2011-10-09	2008-07-26 to 2011-10-17

Chromium[10]. All evaluation objects use Subversion (SVN)[11] as their version control system or at least provide an SVN mirror. They have different characteristics regarding the commit behavior, *i. e.*, they differ in the "size" and length of the SVN history. Chromium, although an open source system, is actively developed by Google engineers. We thus expect it to show a similar commit behavior as commercially developed systems. Table I shows overview information about the evaluation objects. Columns *Start rev.*, *Head rev.* and *Timeframe* illustrate the section of the revision history that we considered for the analyses. We used the complete history from the time where the directory containing the source files was created up to the time when we mirrored the SVN.

B. Evaluation Questions

EQ 1: *What performance challenges arise from the commit activity in the evaluation objects?* We investigate the commit activity of the evaluation objects and assess the commit frequency to deduce the performance requirements for our approach.

EQ 2: *How long does a complete non-incremental quality analysis take?* To determine if incremental analysis is necessary as well as to have a baseline for comparison, we estimate the required time for running a non-incremental analysis on all revisions separately.

EQ 3: *How long does it take to replay the history with the incremental approach?* We evaluate how well our approach can handle the reconfiguration requirement, where all analysis results have to be re-computed.

EQ 4: *How long does it take to process a single revision with the incremental approach?* We investigate how well our approach is capable of providing rapid feedback to developers in response to individual changes to the code base.

EQ 5: *How long does it take to retrieve historized metric values?* To assess our approach with regards to root cause analysis of metric changes, we determine the times required to retrieve historized metric values.

[10]http://src.chromium.org/svn/
[11]http://subversion.apache.org/

Table II
COMPUTED METRICS

Metric	**Description**	**L/G**
LOC	Total number of lines in source file	L
SLOC	Total number of non-empty non-commented lines in source file	L
Max. Nesting Depth	Maximum number of nested blocks	L
Comment Ratio	Fraction between characters in comments and non-comments	L
Longest Method	Length in lines of the longest method within a source file	L
Clone Coverage	Fraction of statements in a source code file that are cloned	G
Aggregation	Aggregated values of all mentioned metrics along the directory tree	G

Table III
ANALYSIS CONFIGURATIONS

System	JabRef	ArgoUML	Chromium
SVN path	trunk/jabref/src/java	trunk/src	trunk/src
includes	.java	.java	.c,.cc,.h,.hh
excludes	-	-	*third_party*

C. Design and Procedure

We mirrored the complete SVN repositories of the evaluation objects on a server in our local network. This ensures that the impact of network delays in our evaluation is kept low. Also, this is closer to a realistic production setting where the quality analysis are performed on a machine with a fast network connection to the SVN server.

We configured both the non-incremental analysis (EQ 2) as well as the incremental analysis server (EQ 3-5) to compute the metrics listed in Table II. The metrics were chosen to provide some variety in calculation technique, *not* to provide a complete quality assessment. Hence, we do not provide details on the metrics used. The column *L/G* indicates whether the metric is local (*L*) or global (*G*).

In all analyses, we included only source code files and thus ignored files that are not relevant for our software metrics. Moreover, we excluded third-party code, since this code is usually not maintained to the same extent as own code. The analysis configurations used for the evaluation objects are summarized in Table III.

The evaluation was performed on a 64-bit machine with two Dual Core AMD Opteron 280 2.4 GHz CPUs, 10 GB of RAM and a Gentoo Linux operating system.

1) EQ 1: Commit activity: We investigated the number of commits over the project history and assessed the maximum number of commits per month, day, hour, and minute. In addition, we determined the average number of files changed per commit, the largest commit, and the total number of files changed.

2) EQ 2: Non-incremental quality analysis: We calculated the metrics mentioned above with a non-incremental

run of ConQAT for both a base revision[12] and the head revision of our evaluation objects. This was performed non-distributedly on a single CPU core. We measured the times t_{base} and t_{head} for the complete analysis process of both revisions, but did not include the time required for the check-out from the version management system. Based on these times, we estimated the time needed to calculate the metric values for all revisions as $t_{full} = 0.5 \times (t_{base} + t_{head}) \times R$, where R is the number of revisions where at least one code file was changed (as determined in EQ 1). Using the average between the analysis time required for the base and head revision assumes linear growth in size, which is only an estimate. However, as we expect the system to grow faster in the beginning, the estimate is likely to be a lower bound.

3) EQ 3: Incremental re-computation of history: We "replayed" the development history of the evaluation objects by processing all changes committed to the SVN repository and incrementally updating the quality indexes. We used only a single worker thread, thus our tool utilizes a single core of the CPU most of the time. Additional threads are used for scheduling and periodic synchronization between scheduler and worker, but these require nearly no CPU resources. We measured the processing time for the complete replay process including fetching revisions and file contents from SVN and the incremental metric computation of all metrics.

4) EQ 4: Processing of incremental changes: From the analysis run of EQ 3 we computed the average time required to process a single revision. For this average we counted only revisions where at least one of the files included in the analysis configuration was changed.

5) EQ 5: Retrieval of historized metric values: We queried the historized clone index for 100 randomly generated pairs of file and revision and retrieved the metric value that the file had in the specific revision. We measured the minimum, maximum, median, and average query time.

D. Results

1) EQ 1: Commit activity: Table IV summarizes the commit activity for the evaluation objects. The maximum number of commits recorded per hour is 50 for ArgoUML. Chromium accounts for the maximum number of commits per day, which is 217. This data enables us to make an estimate about the performance required of our approach. We are interested in peaks of commit activity which are displayed in Figure 5 as an aggregated daily trend view of the development activity over the last six months. All projects are visibly active, however, Chromium surpasses JabRef and ArgoUML by one order of magnitude[13] in terms of commit activity, regularly exceeding 100 daily commits.

[12]We incremented the revision where the source directory was created by 100 to determine the base revision. This was chosen to skip the phase where the initial import of existing source code was performed.

[13]To keep the data for JabRef and ArgoUML visible, the y-axis of the diagram is plotted in log scale.

Table IV
COMMIT ACTIVITY ON THE EVALUATION OBJECTS

Evaluation object	JabRef	ArgoUML	Chromium
Relevant Commits	1496	3650	53470
Max Commits/Day	23	89	217
Max Commits/Hour	17	50	27
Largest Commit (#Files)	285	3636	2700
Total File Changes	7207	18531	346214
Avg. Files/Commit	4.82	5.07	6.47

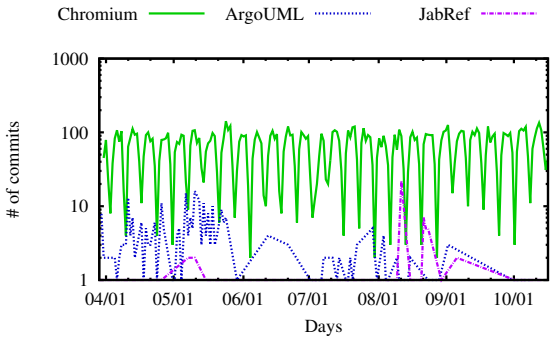

Figure 5. The number of commits per day.

2) EQ 2: Non-incremental quality analysis: Table V shows the analysis times for the non-incremental analysis of the base and head revision of all evaluation objects as well as our estimate for the time required for analyzing all revisions. The analysis of the base revision took from 16 to 83 seconds while the head revision required from 33 to 351 seconds. The resulting estimate for the analysis of all revision falls in the range of 10.8 hours to more than 4 months for the evaluation objects.

3) EQ 3: Incremental re-computation of history: Table VI shows the processing statistics for the incremental history analyses. Column *Overall* contains the overall time required for the replay. Column *Speedup* shows the theoretical speedup factor (*i. e.*, performance gain) of the incremental approach compared to the non-incremental estimate. The overall running times of the incremental analysis ranged between 20 minutes for JabRef and about two days for chromium. The speedup factor ranges from about 30 for the smaller evaluation objects to 65 for Chromium.

4) EQ 4: Processing of incremental changes: Column *Per revision* in Table VI shows the average times for processing a single revision. They are less than one second for the smaller systems, and less than 4 for Chromium.

Table V
CONQAT NON-INCREMENTAL ANALYSIS TIMES

Evaluation object	Analysis of base rev.	Analysis of head rev.	Estimate f. analysis of all revisions
JabRef	19 seconds	33 seconds	10.8 hours
ArgoUML	16 seconds	60 seconds	38.5 hours
Chromium	83 seconds	351 seconds	134.3 days

Table VI
INCREMENTAL ANALYSIS

Evaluation object	Analysis of all revisions	Speedup	Per revision
JabRef	20 minutes	32.3	0.8 seconds
ArgoUML	66 minutes	34.9	0.9 seconds
Chromium	49.2 hours	65.5	3.3 seconds

Table VII
TIMES FOR RETRIEVING HISTORIZED VALUES

Evaluation object	Min	Max	Median	Average
JabRef	0.003 s	0.013 s	0.005 s	0.005 s
ArgoUML	0.003 s	0.012 s	0.004 s	0.004 s
Chromium	0.002 s	0.018 s	0.004 s	0.004 s

5) EQ 5: Retrieval of historized metric values: Table VII shows the times for retrieving a single historized metric value. The times for retrieval were below 0.01 seconds for median and average over all three systems.

E. Discussion

The numbers from Table IV suggest that a "real-time analysis system" should be able to handle at least 50 commits per hour to avoid congestion of the analysis queue during times of many commits. This corresponds to about one minute per analysis. The times from Table V indicate that a non-incremental approach might be sufficient for the smaller systems (at least as long as we do not want to calculate more metrics), while for Chromium there will be a considerable delay between the commit and the analysis results if many commits are performed in a short time interval.

Our incremental analysis approach only requires a couple of seconds per revision, including communication with the SVN server[14]. This is fast enough for real-time developer feedback, as typical delays in an IDE for compiling and committing of changed code are in a similar time range. This is also fast enough to avoid congestion of the analysis queue even under heavy load. This makes our approach also useful for analyzing a larger number of small systems in a software ecosystem on a single machine, while for non-incremental analyses a separate machine has to be allocated for every one or two projects to ensure timely feedback. This is also backed by the disk space occupied by the storage system in our approach, which was about 200 MB and 380 MB for the two smaller systems and 10 GB for Chromium. This allows many projects to be stored on a single machine.

The speedup when performing full history analysis is a factor of 30 to 65. As we were using only a single worker thread, this speedup is not caused by parallel computation but only by the incremental nature of the algorithms. The higher speedup for the larger system is caused by the quadratic run-time of the non-incremental analysis (system size times number of commits), while our approach is expected to run in sub-quadratic time as we only have to recalculate a fraction of the metric values. The time per revision, however, also depends on the size of the system, as for a larger system potentially more files are affected by changes in global metrics.

As shown by Table VII, our organization of metric data as a sparse table in a key/value store is fast enough to support the use-case of interactive querying of metric values when needed (*e. g.*, when opening a file). The way we access the historized metric data (by a prefix query with the file name) makes accessing the metric value for a single revision and retrieving the entire metric history for a file essentially the same operation. Thus, the numbers reported are the same for accessing the full history, *e. g.*, for root cause analysis.

A possible limitation of our approach is the file granularity. Even if only a single bit in a large file changes, we have to process the entire file. This is not a problem as long as the size of the files is within reasonable bounds, as is recommended by most programming guidelines[15]. However, there may be some fairly large files in generated code. The (excluded) third-party code of Chromium, for example, contains a generated file with more than 125,000 lines. Such a file could significantly impact the required analysis times. One solution would be to use a more fine grained unit (such as chunks of lines) for incremental updates. In practice, however, we expect such large files to be either excluded (cloning in generated code is typically not interesting) or to not change very often. In both cases, the impact on the analysis times is manageable.

F. Threats to validity

The selection of evaluation objects always introduces a bias, as it remains unclear how representative they are for the set of all software projects. We tried to mitigate this risk by choosing systems of different sizes in terms of their code base, activity, number of developers, and age. Furthermore, we carefully report the relevant characteristics of the commit history of the evaluation objects, to help the reader compare these numbers to other systems.

The usage of ConQAT when comparing analysis time with a non-incremental approach could bias the results as other analysis frameworks might be faster. However, we use the same analysis code for most parts of the incremental approach as well, thus the difference in time should be only caused by the incremental approach and not differences in the analysis code.

Finally, the performance results may be different for another set of metrics. We tried to use metrics that are well-

[14]To estimate the delay caused by communication with the SVN server, we also analyzed the entire JabRef history from a SVN mirror on the local hard disk. This took only about 6 instead of 20 minutes. However, we decided to stick to the setup with a separate SVN server, as this is the more realistic scenario in a real environment.

[15]The largest file in our evaluation objects had about 17,000 lines, which is way larger than most recommendations, but not large enough to seriously impact our approach.

known, can be applied consistently to both C/C++ and Java code, and contain both local and global metrics. By using not a single metric, but a set of metrics, we also expect to *even out* major differences in the potential performance characteristics of different metrics.

V. Related Work

We differentiate between *manual* and *automated* quality analyses. Both have shortcomings if used in isolation. First, important quality attributes, such as the concise and consistent naming of identifiers [20], elude automatic analysis. Furthermore, questionable quality measures have been proposed, whose value to engineering is unclear [21], [22]. However, manual inspections alone are infeasible for continuous quality control, too. Real-world software is often too large for regular, full-blown manual reviews. These weaknesses are alleviated to a certain degree, if automated and manual analyses are combined. Consequently, automated analyses form the basis of quality control, since they can be executed inexpensively on a continuous basis. In this paper, we thus focus on automated analyses.

A lot of work has been done on automated quality analyses for very large software systems. To tackle the scalability challenges, the following strategies have been proposed: distributing the analyses, incrementally updating the results, as well as a combination of both. Some approaches specialize on a single type of analysis, like architecture conformance checking or clone detection, while others integrate a variety of analyses into a framework to support continuous quality monitoring. We present approaches for each of these strategies, followed by an overview of frameworks designed for continuous quality monitoring.

Incremental: Already in 1995, Wagner and Graham proposed a combination of incremental analysis algorithms and version management for implementing an artefact versioning system [23]. They employ incremental analysis algorithms to restore consistency in documents after modifications, while relating the changes to a specific document version. However, quality, besides consistency, was not yet addressed.

Koschke [12] presents an incremental approach for architecture conformance checking. By investigating the changes possible in either the architectural model, the implementation, or the mapping between those two, strategies are devised to determine entities affected by a modification and thus limit the scope of recalculations of the analysis. As a result, the incremental reflection analysis gives real-time feedback on changes which occurred during an interactive assessment of the architecture. Göde and Koschke [11] present an incremental clone analysis, which calculates the evolution of clones over a system's history. It bases the analysis for each revision on the findings of the earlier ones, creating a mapping between them. The goal is to integrate the incremental clone analysis into an IDE for rapid feedback to the developer. Unlike our approach, both incremental approaches are not designed for a distributed environment, nor do they attempt to integrate multiple analyses.

Distributed: Shang et al. [15] propose to address scalability by adapting tools which originally were designed for a non-distributed environment and deploying them on a distributed platform such as MapReduce. They, however, do not employ incremental analyses.

Incremental and Distributed: In [13], we presented an approach for incremental clone analysis, which is index-based, and uses MapReduce to distribute the computation. As a consequence, the analysis is suitable for being employed in real-time clone management tools. In this paper, we extend this approach with support for additional analyses. Additionally, we provide a change history for the results.

Frameworks: A framework integrating a plethora of analyses in order to support continuous quality monitoring is SQUANER [24]. Like our approach, it proceeds incrementally, and updates are triggered by commits to a version control system. The goal, like ours, is also to provide rapid developer feedback. In addition, SQUANER presents advice for improving the analyzed code base, based on the findings of their analyses. However, the types of analyses supported differs: SQUANER, unlike our approach, focusses exclusively on object-oriented systems. Furthermore, the metrics calculated by SQUANER are file-based and thus limited to local analyses. Our approach, in contrast, supports local as well as global analyses. Additionally, we provide quality history of each file in the system at a per-commit granularity. The type of continuous quality control data provided by SQUANER could not be determined, as the corresponding web site was unreachable. As far as performance is concerned, we can not compare our approach to SQUANER, as no empirical data was available.

Another framework for continuous monitoring of software evolution is proposed by Robbes et al. [25]. In contrast to our approach, they propose a change-based granularity, which draws its information directly from the developers' IDE. This way, every single change can be traced, as modifications are not aggregated in one commit. Instead, we used a file-commit granularity for the following reasons: Firstly, the metrics currently computed are commonly reported on a per-file basis, and aggregated for the entire system. As far as the size of commits, and thus the accuracy of our increments are concerned, we assume that providing rapid feedback on a file-commit granularity to developers might induce them to commit coherent changes more frequently in order to assess their effects. Furthermore, the purpose of continuous quality control is to assess the big picture of a system, instead of experimental edits by developers.

Also Microsoft provides a reporting framework with its Team Foundation Server (TFS) [26], which amongst others

also reports on a set of quality metrics[16]. Changes by the developers are stored in relational databases, which are connected to a data warehouse to which an OLAP[17] cube is attached. Queries requesting quality data run against either the data warehouse or the OLAP cube. On Microsoft documentation [27] we could find information on the refresh rate of the data warehouse, which is by default set to 3600 seconds. The cube cache might even take longer to learn the new data. Recommendations advise not to set this interval to considerably smaller values to avoid excessive blocking of resources. Therefore we assume that metric calculations might be done periodically in batch mode, instead of commit-based incrementally. Even though TFS might be suitable for our continuous quality monitoring use case, it is not accurate enough when it comes to rapid feedback, which our approach is providing within seconds.

VI. Conclusion and Future Work

We have presented a platform for incremental and distributed computation of quality metrics. To support a broad range of analysis approaches, it can compute both local (*e. g.*, depth of nesting) and global metrics (*e. g.*, clone coverage). Moreover, it provides functionality common to different incremental analyses such as version control synchronization, filtering, caching and historization of analysis results, to simplify the development of novel ones.

The evaluation we performed on three open-source software systems demonstrates that it scales both in size and time: for each evaluation object, incremental computation of a set of both local and global metrics took less than 4 seconds on average. This is fast enough for real-time developer feedback. Furthermore, retrieval of metric values for historic system versions took less than 0.1 seconds, which is fast enough for interactive use.

We plan several extensions: first, we want to improve result presentation and visualization. For example, user-specific views, which allow a developer to focus on the code she is currently working on could improve usability. Second, we want to implement interactive filtering capabilities to exclude individual false positives. Finally, we want to recomputate the metric values for the program history in reverse order to quickly produce results for the near past.

References

[1] D. L. Parnas, "Software aging," in *ICSE'94*, 1994.

[2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *TSE*, vol. 27, no. 1, 2001.

[3] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE '09*, 2009.

[4] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool support for continuous quality control," *IEEE Softw.*, 2008.

[5] F. Deissenboeck, "Continuous quality control of long-lived software systems," Ph.D. dissertation, TU München, 2009.

[6] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, 2009.

[7] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," *Softw. Eng. Notes*, vol. 20, no. 4, 1995.

[8] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," in *ICPC'09*, 2009.

[9] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, 2008.

[10] T. J. McCabe, "A complexity measure," in *ICSE'76*, 1976.

[11] N. Göde and R. Koschke, "Incremental clone detection," in *CSMR'09*, 2009.

[12] R. Koschke, "Incremental reflexion analysis," in *CSMR'10*, 2010.

[13] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *ICSM'10*, 2010.

[14] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, 2010.

[15] W. Shang, B. Adams, and A. E. Hassan, "An experience report on scaling tools for mining software repositories using mapreduce," in *ASE'10*, 2010.

[16] M. H. Halstead, *Elements of software science*. Elsevier, 1977.

[17] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, 2007.

[18] E. Juergens, "Why and how to control cloning in software artifacts," Ph.D. dissertation, TU München, 2011.

[19] G. Rothermel and M. Harrold, "A safe, efficient regression test selection technique," *ACM TOSEM*, 1997.

[20] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Softw. Quality J.*, vol. 14, no. 3, 2006.

[21] Kitchenham, Jeffery, and Connaughton, "Misleading metrics and unsound analyses," *IEEE Softw.*, vol. 24, no. 2, 2007.

[22] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in *International Software Metrics Symposium*, 2004.

[23] T. Wagner and S. Graham, "Integrating incremental analysis with version management," in *ESEC'95*, 1995.

[24] N. Haderer, F. Khomh, and G. Antoniol, "SQUANER: A framework for monitoring the quality of software systems," in *ICSM'10*, 2010.

[25] R. Robbes and M. Lanza, "A change-based approach to software evolution," *Elec. Notes Comp. S.*, 2007.

[26] Microsoft team foundation server. [Online]. Available: http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview

[27] V. Blasberg. TFS reporting architecture notes. [Online]. Available: http://weblogs.asp.net/vblasberg/archive/2008/06/04/tfs-reporting-architecture-notes.aspx

[16] TFS is included for the sake of completeness and because of its industrial presence — despite the fact that not enough information about its internal mechanisms could be obtained to compare it to our approach.

[17] On-Line Analytical Processing