Incremental Origin Analysis of Source Code Files[*]

Daniela Steidl
steidl@cqse.eu

Benjamin Hummel
hummel@cqse.eu

Elmar Juergens
juergens@cqse.eu

CQSE GmbH Garching b. München Germany

ABSTRACT

The history of software systems tracked by version control systems is often incomplete because many file movements are not recorded. However, static code analyses that mine the file history, such as change frequency or code churn, produce precise results only if the complete history of a source code file is available. In this paper, we show that up to 38.9% of the files in open source systems have an incomplete history, and we propose an incremental, commit-based approach to reconstruct the history based on clone information and name similarity. With this approach, the history of a file can be reconstructed across repository boundaries and thus provides accurate information for any source code analysis. We evaluate the approach in terms of correctness, completeness, performance, and relevance with a case study among seven open source systems and a developer survey.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics

General Terms

Algorithms

Keywords

Software Evolution, Origin Analysis, Clone Detection

1. INTRODUCTION

Software development relies on version control systems (VCS) to track modifications of the system over the time. During development, the system history provides the opportunity to inspect changes in a file, find previous authors, or recover deleted code. For source code analyses, such as quality or maintenance assessments, the history of a system makes it possible to evaluate the software evolution and to discover metric trends. Many analyses depend on the complete history of source code files. For example, the clone community evaluates the impact of clones on maintenance by measuring the stability of cloned code in terms of age since the last change [21] or as change frequency [8, 20]. The defect prediction community also relies on history information: Prominent approaches use code churn, code stability, change frequency, or the number of authors per file to predict software bugs [15, 23, 24]. Other analyses such as type change analysis, signature change analysis, instability analysis [16], or clone genealogy extractors [17] also rely on accurate program element mapping between two versions.

Unfortunately, the source code history of a file is often not complete because movements or copies of the file are not recorded in the repository. We refer to those moves as *implicit* moves in contrast to recorded, *explicit* moves. Lavoie [22] has shown that a significant portion of file movements are implicit. This is due to several different reasons: When moving a file, programmers might not use the commands as provided by VCSs such as subversion (SVN), *e. g.*, *svn move*. Sophisticated development environments such as Eclipse offer the refactoring *rename* which automatically records the rename as a move in the repository. When developers rename the file manually, though, or copy and paste it, the move/copy will not be recorded in the VCS. Furthermore, when projects are migrated from one type of VCS to another, move/copy information will be lost. For example, many open source projects have been previously managed with CVS (where no origin information was stored at all) and were later migrated to SVN, where the moves and copies in the history of CVS time periods remain missing. On top, if parts of a repository are moved across repository boundaries, this event is not recorded in the history and cannot be captured by any VCS. With our approach, we will show that up to 38.9% of the files in open source systems have an incomplete history.

Problem Statement. *The history of source code files as recorded in VCSs is often incomplete and provides inaccurate information for history-based static code analyses.*

Reconstructing a complete source code history requires an *origin analysis* for each file. Reverse engineering the history of a file is not trivial: If a file is moved or renamed, it may also be modified at the same time. Hence, a simple comparison of whether the file's content remains the same is not sufficient. Current approaches that perform any kind of origin analysis, either on file level [7, 22] or function level [9, 10] are release-

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

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597111

based and are not run incrementally. All published case studies include only few releases of a software system as snap-shots and compare them to each other. Release-based origin analysis can not reverse-engineer the complete history in feasible time, and thus provides only a coarse picture. Our approach, by contrast, works incrementally and can analyze every commit, even in histories with ten thousands of commits. Hence, our approach provides a fast and fine-grained origin analysis.

For every commit in the system's history, our approach determines for an added file whether it was moved or copied from a file in the previous revision. We extract repository information to detect explicit moves and use heuristics based on cloning, naming, and location information to detect implicit moves. We evaluate the approach in terms of correctness, completeness, performance, and relevance with a case study with seven open source systems and a survey among developers. Although the approach is generally independent of the underlying version control system, we evaluated it only on SVN-based systems.

Contribution. *In this paper, we present an incremental, commit-based approach using repository information as well as clone, naming, and location information to reconstruct the complete history of a file.*

2. RELATED WORK

We group related work to origin analyses by file-based, function-based, and line-based origin analyses and code provenance. [16] gives an overview of other program element matching techniques in the current state of the art.

2.1 File-based Origin Analysis

The work by Lavoie et al., [22], is most similar to ours as the authors also focus on inferring the history of repository file modifications. They reverse engineer moves of files between released versions of a system using nearest-neighbor clone detection. They evaluate their approach on three open source systems Tomcat, JHotDraw, and Adempiere. The evaluation compares the moves detected by their approach with commit information extracted from the repository. Our work differs in three major aspects from the work by Lavoie: First, we also detect file copies, besides moves. Lavoie et al. do not consider copied files, hence, our approach provides more information to recover a complete history. Second, Lavoie et al. reconstruct file movements only between released versions of a system in feasible time. With our approach, we analyze every single commit in the history and find moves at commit level still with feasible runtime. Hence, we provide more details for developers: When a developer accesses the history of a file, *e. g.*, to view changes, review status, or commit authors, he needs the complete history on commit level and not just between released versions. Third, Lavoie et al. compare their results only against the repository oracle, although their approach reveals a significant amount of *implicit moves.* As the authors state, they were not able to thoroughly evaluate these implicit moves except for a plausibility check of a small sample performed by the authors themselves. From this point of view, our work is an extension of Lavoie's work: In addition to an evaluation based on the repository information, we conduct a survey amongst developers to evaluate implicit moves. Hence, we provide reliable information about the overall precision of our approach.

The work of [2] automatically identifies class-level refactorings based on vector space co- sine similarity on class identifiers. The approach aimed to identify cases of class replacement, split, merge, as well as factoring in and out of features. Based on a case study with only one system, the approach does not always identify a unique evolution path per class. Instead, the authors manually inspected the results. Our work, in contrast, uses clone detection as suggested by the authors in their future work section, to obtain unique results which we evaluated in a large case study based on repository and developer information.

Git [1] provides an internal heuristic mechanism to track renames and copies based on file similarity. To our best knowledge, there exists no scientific evaluation on Git's heuristic to which we could compare our approach. In contrast, many users report on web forums such as stackoverflow.com that Git fails to detect renames[1] or becomes very slow when the option to also detect copies is set.

Kpodjedo [19] addresses a similar problem with a different approach. Using snapshots of the Mozilla project, the authors reverse engineer a class diagram for each revision and apply the ECGM algorithm (Error Correcting Graph Matching) to track classes through the evolution of the software. They do not aim to find the origin for each file though, but to discover a stable core of classes that have existed from the very first snapshot with no or little distortion.

The work of [7] detects refactorings such as splitting a class and moving parts of it to the super- or subclass, merging functionality from a class and its super- or subclass, moving functionality from a class to another class, or splitting methods. The refactorings are detected by changes in metrics of the affected code – this relates to our origin analysis as they also track movements of code. However, the authors admit that their approach is vulnerable to renaming: "Measuring changes on a piece of code requires that one is capable of identifying the same piece of code in a different version." As the authors use only "names to anchor pieces of code", they cannot track metric changes in the same source code file if the file name changed, leading to a large number of false positives in their case study. Hence, our approach constitutes a solution to one of the drawbacks of the work by [7] by providing accurate renaming information.

On a very loose end, the work of [14] also relates to our work: it proposes a new merging algorithm for source code documents and includes a rename detector, which can detect renames of a file - which is also part of our origin analysis. However, the authors use their rename detector in quite a different context: They present a new merging algorithm that merges two different versions of the same file and can detect when one version is basically the same as the other except for a syntactical rename. Due to the context of merging two files, the input to their algorithm is quite different to the input of our algorithm.

2.2 Function-based Origin Analysis

Several papers have been published to track the origin of functions instead of files (classes). One could argue that aggregating the result of function tracking can solve the problem of file tracking. However, we disagree for the several reasons. First, there has been no scientific evaluation of how

[1] `http://stackoverflow.com/questions/14527257/git-doesnt-recognize-renamed-and-modified-package-file`, last access 2013-12-03

many functions a file may differ over and still be considered the same file. Our approach is also based on thresholding, but we present a thorough evaluation that confirms the choice of thresholds. Second, even if the thresholds for aggregating function tracking were evaluated, our origin analysis would produce more flexible results: Our origin analysis is based on clone detection in flexible blocks of source code such that we can detect that a file is the same even if code was extracted into new methods, added to a method, or switched between methods. Third, relying on function tracking would make the analysis dependent on languages that contain functions. Our approach, by contrast, is language-independent and can be also applied to function-less code artefacts such as HTML, CSS, or XML.

Godfrey et al. coined the term *origin analysis* in [9,10,25]. They detect merging and splitting of functions based on a call relation analysis as well as several attributes of the function entities themselves. Automatically detecting merging and splitting on file level remained an area of active research for the authors in [10] as this is currently done only manually in their tool. As merging files is part of our move detection, our work is an extension of their work.

In [18], the authors propose a function mapping across revisions which is robust to renames of functions. Their algorithm is based on function similarity based on naming information, incoming and outgoing calls, signature, lines of code, complexity, and cloning information from CCFinder. The authors conclude that dominant similarity factors include the function name, the text diff and the outgoing call set. In contrast, complexity metrics and cloning information (CCFinder) were insignificant. Our work also relies on naming information, however, for our work also the cloning information provided a significant information gain.

Rysselberghe [26] presents an approach to detect method moves during evolution using clone detection. However, in contrast to our work, as the authors use an exact line matching technique for clone detection, their approach cannot handle method moves with identifier renaming.

2.3 Line-based Origin Analysis

Quite a few papers track the origin of a single code line [3,5] with a similar goal–to provide accurate history information for metric calculations on the software's evolution. However, identifying changed source code lines cannot provide information for the more general problem of file origin analysis for two reasons. First, it has not been evaluated yet how many source code lines in a file can change such that the file remains the "same" file. Second, tracking single lines of code excludes a lot of context information. Many code lines are similar by chance, hence, aggregating these results to file level creates unreliable results.

2.4 Code Provenance

Code provenance and bertillonage [6,11] constitute another form of origin analysis: It also determines where a code artifact stems from, but either for copyright reasons (such as illegally contained open-source code in commercial software) or for detection of out-dated libraries. Code provenance differs from our origin analysis as we detect the origin of a file only within the current software containing the file whereas code provenance considers a multiple project scope for origin detection.

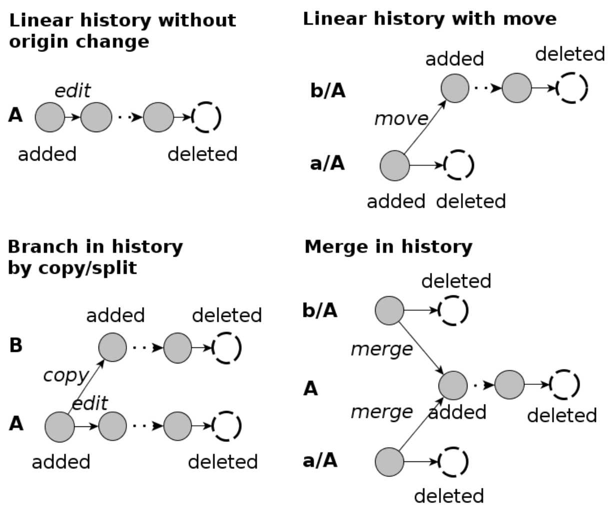

Figure 1: Metamodel of the history of a file.

3. TERMINOLOGY

We use the following terms to model the history of a source code file: In the simplest case, the history of a file is *linear* (Figure 1); a file is truly added – in the sense that the file is really new and had no predecessor in the system – with a specific name (*e. g.*, A.java or A.c) at a specific location (*e. g.*, a path) in revision n. Then the file may be edited in some revisions, and finally deleted (or kept until the head revision of the repository). The other cases which are more complex are the target of our detection (Figure 1):

- *Move*: A file is moved if it changes its location (its path) in the system. We also consider a rename of the file name as a move of the file. Repositories represent a move typically as a file that existed in revision n at some path, got deleted at this location in revision $n+1$, and was added at a different location and/or with a different name in revision $n+1$.
- *Copy*: A file is copied if the file existed in revision n and $n+1$ at some location (*e. g.*, a path), and was added at a second location in revision $n+1$ without being deleted at the original location. A copy may include a name change for the new file. In that case, a copy might also be a split of a file into two files.
- *Merge*: A file got merged if multiple files were unified into one single file. The merged file got added in revision $n+1$ and multiple files from revision n are deleted in revision $n+1$. As VCSs such as SVN cannot capture merges at all, repository information about them is missing.

A moved file still has a linear, unique history (no branch or split, and a unique predecessor and successor for each revision except those of the initial add and the final delete). A copied file can have two sucessors but both the original and the copied file have a unique predecessor. In case of a merge, the predecessors of the merged file are not unique.

The detection of moves and copies is conceptually the same in our approach, as they differ only insofar as the original file is deleted during a move, whereas it is retained during a copy. Hence, we will not differentiate between the two

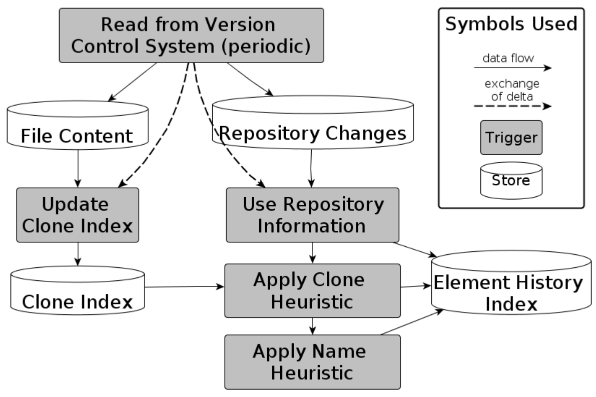

Figure 2: Design of a the approach.

in the remainder of this paper and use the term *move* as generalization for both copy and move.

Although we use the terms *branch* and *merge* in this paper, they do not have the same meaning as *branching* and *merging* in the SVN language: For SVN, a *branch* is used to create multiple development lines for experimental purposes or different customer versions of the same product. A *merge* is used afterwards to unify changes on a branch and the main trunk. In our case, however, we limit our analysis to the trunk of the repository only. Still, merges as described above can occur, where both the two original files are part of the trunk and were consolidated into one file.

4. APPROACH

In this section, we present the design and implementation details of our approach to recover the history of a source code file. For each commit, the approach determines whether an added file had a predecessor in the previous revision.

4.1 Underlying Framework

The approach is implemented within the code-analysis framework Teamscale [4, 12] which performs code analyses incrementally and distributable for large-scale software systems. It allows to configure analyses that are triggered by each commit in the history of a system – the analyses then process the added, changed, and deleted files per commit and update the results incrementally. The framework provides repository connectors to many common VCSs such as SVN, Git, or TFS and an incremental clone detector [13].

4.2 Overview

The approach detects explicit and implicit moves, copies, and merges of a source code file using different sources of information - the repository information (when move information is explicitly recorded) and information from heuristics (when explicit information is missing). We suggest two different heuristics, a *clone* heuristic and a *name-based* heuristic. The clone heuristic uses clone detection as a measure of content similarity and aims to detect renames. We do not use simpler metrics such as longest common sequence algorithm to find content similarity because we also want to detect files as moves even if their code was restructured. The name-based heuristic uses primarily information about the name and path, but also includes the content similarity based on clone detection to avoid false positives.The name-based heuristic can only detect moves without rename.

The heuristics are designed to complement each other. The heuristics and the repository information can be applied sequentially in any order. All three sources of information are described in more detail in Sections 4.3–4.5. For now, we choose an exemplary sequence of using the repository information first – as this is the most reliable one – and then, the clone heuristic, followed by the name heuristic.

In this setup, the approach processes the set of added files for each revision n in the history, as follows:

1. Extract information about explicit moves, *i. e.*, moves that were recorded by the VCS. Mark explicitly moved files as detected and remove them from the set.

2. For the remaining set, apply the clone detection to find files at revision $n-1$ that have a very similar content. Mark the detected files and remove them from the set.

3. For the remaining set, apply the name-based heuristic. Mark the detected files and remove them from the set.

We do not know in advance whether it is best to use only one heuristic, or to combine them both (either *name* before *clone* or *clone* before *name*). Figure 2 visualizes the exemplary sequential approach as described above. We will evaluate the best ordering in Section 5.

4.3 Explicit Move Information

Partly, information about a file's origin are already explicitly recorded in the VCS. VCSs such as SVN, for example, provide commands for developers such as *svn move*, *svn copy*, or *svn rename*. Using these commands for a move of a file f from path p to path p', SVN stores the information that p is the origin for f when added at location p'. Sophisticated development environments such as Eclipse offer rename refactorings or move commands that automatically record the origin information in the repository. The recorded information can then be extracted by our framework and stored as a detected move or copy.

4.4 Clone Heuristic

The clone heuristic processes all added files $f \in F$ of revision n in its input set as follows: Using incremental clone detection, it determines the file from revision $n-1$ which is the most similar to f. We use a *clone index* to retrieve possible origin candidates $f' \in F'$ and a *clone similarity* metric to determine if f and any f' are similar enough to be detected as a move. This metric is a version of the standard clone coverage.

Incremental Clone Index. We use parts of the token-based, incremental clone detection as described in [13]. However, any other incremental clone detector could be used as well. The clone detector splits each file into *chunks* which are sequences of l normalized tokens. The *chunk length* l is a parameter of the clone detection as well as the *normalization*. The chunks of all files are stored together with their MD5 hash value as a key in a *clone index* which is updated incrementally for each commit. With the MD5 hash value, we identify chunks in file f that were cloned from other files f' which are, hence, considered to be the set F' of possible origin candidates. With the information about cloned chunks,

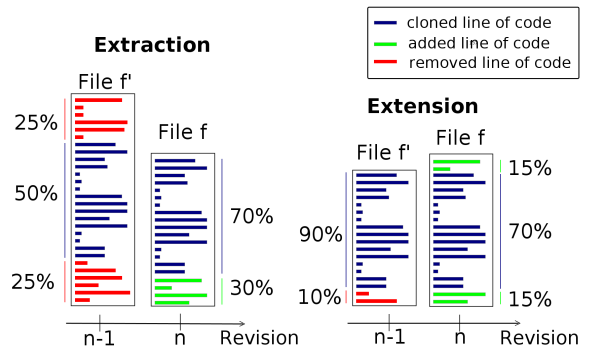

Figure 3: Examples of a detected move/copy of file f' to f with thresholds $\theta_1 = 0.7$ and $\theta_2 = 0.5$.

we define the clone similarity metric. (More details about the clone index and how to use it to retrieve clones are provided in [13].)

Calculating Clone Similarity. The *clone similarity* metric measures the similarity between two files: f is similar to f', if the number of chunks in file f that also occur in f' (based on the MD5 hash) relative to the total number of chunks in f is higher than a certain threshold θ_1. Intuitively, this corresponds to: f is similar to f' if at least a certain amount of code in f was cloned from f'. We pose a second constraint that also a certain amount θ_2 of code in f' still needs to be in f. Figure 3 shows two examples f' and f that are similar based on $\theta_1 = 0.7$ and $\theta_2 = 0.5$.

If we had used only threshold θ_1, then we would have, *e. g.*, detected a very large file f' as origin for a very small file f, if most of the content in f stems from f'. However, there are potentially many large files f' that could be the predecessor for the small piece of code in f, hence, the origin analysis is error-prone to false positives. Consequently, we use two thresholds to ensure that also a certain amount of code from f' is still in f.

Intuitively, one might set $\theta_1 = \theta_2$. This would allow to detect only moves/copies including extensions of the origin file – when the new file as additional code compared to its origin (Figure 3 right example). However, we also would like to detect extractions, *i. e.*, a part of f' is extracted into a new file f (Figure 3, left example). We observed those extractions frequently during preliminary experiments in practice and assume that they represent refactorings of existing code, when a new class is extracted or code is moved to a utility class. To also detect extractions, we set θ_2 smaller than θ_1. Setting θ_2 smaller than θ_1 still detects file extensions.

Move Detection. For the move detection, we calculate the similarity of each added file f and all files $f' \in F'$ at revision $n - 1$ that have common chunks with f. The incremental clone index allows to retrieve these files efficiently. We mark the file f' with the highest clone similarity to f as origin.

Parameters. We selected the following parameters:

Chunk length: We set the chunk length to 25 after preliminary experiments showing that setting the chunk length is a trade-off between performance and recall of the clone heuristic: The smaller the chunk length, *i. e.*, the less tokens a chunk consists of, the higher the chance that there are hashing collisions in the clone index because shorter normalized token sequences appear more often in big systems. This makes the calculation of the clone similarity more expensive as it queries the index how many chunks from file f also appear in other files f'. On the other side, it holds that the larger the chunk size, the smaller the recall: A file consisting of few tokens, *e. g.*, an interface with only two method declarations, does not form a chunk of the required length and will not appear in the clone index. As no clone similarity can be calculated, it cannot be detected as move.

Normalization: We chose a conservative token-based normalization, which does not normalize identifier, type keywords, or string literals, but normalizes comments, delimiters, boolean-,character-, number literals and visibility modifier. We chose a conservative normalization as we are interested in content similarity rather than type 3 clones.

Thresholds After preliminary experiments on the case study objects (Section 5), we set $\theta_1 = 0.7$ and $\theta_2 = 0.6$. The evaluation will show that those thresholds yield very good results independent from the underlying system.

4.5 Name-based Heuristic

As a second heuristic, we use a name-based approach: To detect for each added file $f \in F$, whether it was moved, the name-based heuristic performs the following two actions:

1. It extracts all files $f' \in F'$ from the system that have the same name and sorts them according to their path similarity with f. (We use the term *name* for the file name, *e. g.*, `A.java` and *path* for the file location including its name, *e. g.*, `src/a/b/A.java`.)

2. It chooses $f' \in F'$ with the most similar path and the most similar content using cloning information.

Sorting the paths. The heuristic extracts all files $f' \in F'$ from the previous revision that have the same name and are therefore considered as origin candidates for moves. The origin candidates are sorted based on path similarity to f. For example when f has the path `/src/a/b/A.java` and the two candidates are f' =`/src/a/c/A.java` and f'' =`/src/x/y/A.java`, then f' will be considered more similar. Similarity between two paths is calculated with the help of a diff algorithm: With f' and f'' being compared to f, all three paths are split into their folders, using `/` as separator. The diff algorithm returns in how many folders f' and f'' differ from f. Path f' is more similar to f than f'' if it differs in less folders from f. If f' and f'' differ in the same number from f, the sorting is determined randomly.

Comparing the content. The heuristics iterates over all origin candidates, sorted according to descending similarity. Starting with f' with the most similar path to f, it compares the content of f' and f: It calculates the *clone similarity* between f' and f, which is the same metric as used in Section 4.4. To determine whether f' and f are similar enough to be detected as move/copy we use thresholds $\theta_1 = 0.5$ and $\theta_2 = 0.5$. We determined both thresholds with preliminary experiments. In comparison to the *clone heuristic*, we use lower thresholds as we have the additional information that the name is the same and the paths are similar.

For this heuristic, we use a separate, in-memory clone index in which we only insert file f and the current candidate f' of the iteration. Hence, we can choose a very small *chunk size* without loosing performance drastically. After preliminary experiments, we set the chunk size to 5. As discussed in Section 4.4, the clone heuristic cannot detect moves and copies of very small files, *e. g.*, interfaces or enums, as the chunk size needs to be large to ensure performance. The name heuristic addresses this problem by using the name information to reduce the possible origin candidates and applies the more precise clone detection with small chunk size only on very few candidates. Hence, this heuristic can detect those small files that are missed by the clone heuristic.

5. EVALUATION

The evaluation analyzes four aspects of the approach which will be addressed by one research question each: Completeness, correctness, performance, and relevance in practice. Section 5.1 shows the research questions which are answered with one case study each (5.4-5.7).

5.1 Research Questions

RQ1 (Completeness): What's the recall of the heuristic approach? We evaluate if the heuristics retrieve all moves in the history of a software system. If the heuristics succeed to reconstruct the complete history of a source code file, they can provide accurate information for all static code analyses mining software histories.

RQ2 (Correctness): What is the precision of the approach? We determine how many false positives the approach produces, *i. e.*, how many files were determined as move although being newly added to the system. To provide a correct foundation for evolution mining static code analyses, the number of false positives should be very low.

RQ3 (Relevance): How many implicit moves does the approach detect? We evaluate how much history information our approach reconstructs. This indicates how many errors are made when applying static code analyses of software repositories without our approach.

RQ4 (Performance): What is the computation time of the algorithm? We measure how fast our approach analyzes large repositories with thousands of commits.

Ordering of the heuristics. We also evaluate whether it is best to use only the clone or only the name heuristic, or both in either ordering. The ordering influences correctness and completeness of the approach. As the heuristics are applied sequentially, the second heuristic processes only those files that were not detected as moves by the first heuristic. Hence, the first heuristic should have the highest precision because the second heuristic cannot correct an error of the first heuristic. The recall of the first heuristic is not the major decision criteria for the ordering as a low recall of the first heuristic can be compensated by the second heuristic.

5.2 Setup

For evaluation, we change the design of our algorithm (Figure 2) and remove the sequential execution of extracting explicit move information and applying the two heuristics. Instead, we execute all three in parallel. Figure 4 visualizes the evaluation setup. This gives us the opportunity to evaluate the results of both heuristics separately and use the information extracted from the repository as gold standard.

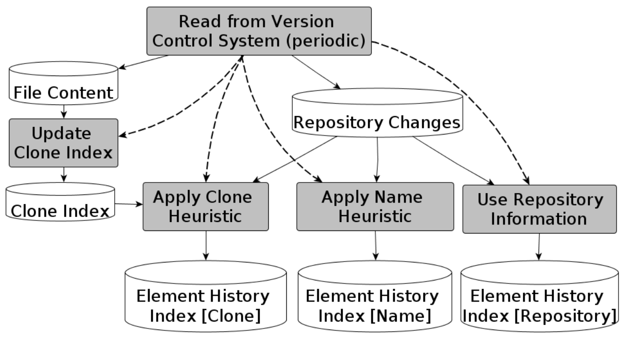
Read from Version Control System (periodic)
File Content
Repository Changes
Update Clone Index
Clone Index
Apply Clone Heuristic
Apply Name Heuristic
Use Repository Information
Element History Index [Clone]
Element History Index [Name]
Element History Index [Repository]

Figure 4: Setup for Evaluation.

5.3 Study Objects

Research questions 1-4 require different study objects. To evaluate recall and precision, we need systems with a large number of explicit moves, providing a solid gold standard. To show the relevance of our approach, we need systems with a large number of implicit moves. To select suitable case study objects, we run our setup on the systems shown in Table 1. All study objects use SVN as repository. Amonst others, the table shows the number of commits which is usually smaller than the difference between end and start revision as, in big repositories with multiple projects, not all commits affect the specific project under evaluation. Running our analysis first reveals how many explicit and implicit moves each repository has. The number of implicit moves is the result of our heuristics, which might be error-prone, but still sufficient to select the study objects. Table 2 shows that ConQAT, ArgoUML, Subclipse, and Autofocus have the largest numbers of explicit moves. ArgoUML and Autofocus also have many implicit moves.

ConQAT has a large number of explicit and a small number of implicit moves, as all developers develop in Eclipse and all major moves within the repository were done with the *svn move* command. ArgoUML has a high number of explicit moves. However, the significant number of implicit moves results from a CVS to SVN migration. Most implicit moves are dated prior to revision 11199, the beginning of the migration.[2] Although Autofocus 3 is primarily developed in Eclipse, there is still a significant number of implicit moves. Our manual inspection reveals that some major moves of packages (such as restructuring the `mira/ui/editor` package or moves from the `generator` to the `common` package) were not recorded. For Subclipse, only a small number of implicit but a large number of explicit moves were found, matching our initial expectation that Subclipse developers pay attention to maintaining their repository history as they develop a tool with the purpose of repository integration into Eclipse. Vuze (former Azureus) is also a system that was migrated from CVS to SVN.[3] Besides many implicit moves due to the migration, there are not many explicit moves in the new SVN. For all other systems, we do not have additional information about their history.

[2]An ArgoUML contributor states that this commit was done by the cvs2svn job and he himself did a manual cleanup in rev. 11242.

[3]http://wiki.vuze.com/w/Azureus_CVS

Table 1: Case Study Objects

Name	Language	Domain	History [Years]	Size [LOC]	Revision Range	# Commits
ArgoUML	Java	UML modeling tool	25	370k	2-19910	11721
Autofocus 3	Java	Software Development Tool	3	787k	18-7142	4487
ConQAT	Java	Source Code Analysis Tool	3	402k	31999-45456	9309
jabRef	Java	Bibliography reference manager	8	130k	10-3681	1521
jHotDraw7	Java	Java graphics GUI framework	7	137k	270-783	435
Subclipse	Java	Eclipse Plugin for Subversion	5	147k	4-5735	2317
Vuze (azureus)	Java	P2P file sharing client	10	888k	43-28702	20762

Table 3: Recall

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.84	0.96	0.92	0.98
ArgoUML	0.89	0.97	0.97	0.98
Subclipse	0.91	0.98	0.99	0.99
Autofocus	0.73	0.82	0.85	0.86

Table 2: Explicit and Implicit Repository Moves

Name	Explicit	Implicit
ConQAT	8635	191
ArgoUML	3571	1358
Subclipse	573	38
Autofocus 3	3526	1135
JabRef	6	34
JHotDraw 7	298	175
Vuze	73	860

5.4 Completeness (RQ1)

5.4.1 Design

To evaluate the completeness of the heuristics, we use the recall metric. We calculate the recall based on the information extracted from the repository which gives us only an approximation of the true recall: Among all explicit moves, we calculate how many are found by our heuristics. However, the heuristics' true recall over a long history cannot be calculated as there is no source of information about how many files were moved. Even asking developers would not provide a complete picture if the history comprises several years. The experimental setup (Section 5.2) allows us to run the heuristics in parallel to extract the repository information. We chose those open source systems from Table 1 that contain the largest numbers of explicit moves, *i. e.*, ConQAT, ArgoUML, Subclipse and Autofocus. With their explicit move information, those systems create a solid gold standard for evaluation.

5.4.2 Results

Table 3 shows the recall for all possible orderings of the two heuristics (as explained in Subsection 5.1). The sequential execution of the name heuristic before the clone heuristic (annotated in the column header with *Name-Clone*) yields the most promising results, with a recall of above 98% for ConQAT, ArgoUML, and Subclipse. Only for the study object Autofocus, the recall is slightly lower with 86%. This is due to one limitation of the current approach: its failure to detect reverts in the repository. The approach can detect moves or copies only if the origin is present in the previous revision. If some part of the system was deleted and later readded (reverted), our approach cannot detect this. As Autofocus contains some reverts, the recall is slightly lower. The name-heuristic itself already achieves a recall of above 82% for all systems, indicating that many moves are done without renaming.

5.5 Correctness (RQ2)

5.5.1 Design

The correctness of the approach is evaluated with the precision metric. The precision of the approach cannot be evaluated solely based on explicit move information as the repository does not provide any information whether implicit moves detected by a heuristic are true or false positives. Hence, we evaluate the precision two-fold: First, we calculate a lower-bound for the precision based on the explicit move information assuming that all implicit moves are false positives. This assumption is certainly not true, but it allows us to automatically calculate a lower bound and make a first claim. The calculation is done on the same study objects as in the completeness case study because it requires a large number of explicit moves. Second, in a survey, we gather developer ratings from ConQAT and Autofocus developers for a large sample of implicit moves and use them to approximate the true precision of the approach.

Challenges in Evaluating the Precision. Although the precision is mathematically well defined as the number of true positives divided by the sum of true and false positives, defining the precision of our approach is not trivial: In case of a merge as shown in Figure 1, the origin predecessor of an added file is not unique. However, our heuristics are designed such that they will output only one predecessor. Hence, in case of a merge, it is not decidable which is the *correct* predecessor. In the absence of explicit move information or a disagreement between the two heuristics, there is no automatic way to decide which result is a true or false positive. For calculating the lower bound of the precision, we assumed the disagreement to represent two false positives.

Developer survey. We chose the two study objects ConQAT and Autofocus because their developers were available for a a survey. Table 4 shows the participation numbers as well as the average years of programming experience and the average years of programming for the specific project. As the participants had many years of programming experience and were core developers of the respective system under evaluation, we consider them to be suitable for evaluation.

Table 4: Experience of Survey Subjects [years]

System	Developers	∅ Program. Experience	∅ Project Experience
ConQAT	6	13.8	5.9
Autofocus	3	17	4.3

Table 5: Lower Bound for Precision

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.91	0.98	0.92	0.97
ArgoUML	0.73	0.76	0.71	0.72
Subclipse	0.92	0.98	0.99	0.99
Autofocus	0.75	0.74	0.72	0.72

Table 6: Precision based on Developers' Rating

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.975	0.925	0.95	0.95
Autofocus	1.0	0.90	0.90	0.90

Table 7: Statistical Approximation of Precision

System	Clone	Name	Clone-Name	Name-Clone
ConQAT	0.997	0.998	0.996	0.998
Autofocus	1.0	0.97	0.97	0.97

We showed the developers a statistic sample of implicit moves and moves for which a heuristic led to a different result than recorded in the repository. We randomly sampled the moves such that the developers evaluated n moves detected by the clone heuristic, n by the name heuristic and n by each of the sequential executions. However, as those four sample sets intersect, the developers rated less than $4n$ samples. For ConQAT, we set $n = 40$ and for Autofocus, we set $n = 20$. To evaluate a sample, developers were shown the diff of the file and its predecessor and were allowed to use any information from their IDE. The developers rated each move as *correct* or *incorrect*, indicating also their level of confidence from 0 (*I am guessing*) to 2 (*I am sure*). Each move was evaluated by at least three developers. In case of disagreement, we chose majority vote as decision criteria. As we expected most moves to be correct, we added m files with a randomly chosen predecessor in the survey to validate the developer's attention while filling out the survey. For ConQAT we chose $m = 20$, for Autofocus $m = 10$.

5.5.2 Results

Lower bound. Table 5 shows a lower bound for the precision. The accuracy of the lower bound depends on the number of implicit moves as they were all assumed to be false positives. Hence, the more implicit moves a system has, the lower the lower bound.

The systems ConQAT and Subclipse show a very high lower bound for the precision with above 90% for all scenarios. This means that among all explicit moves, the heuristics performed correctly in at least nine out of ten cases. The execution of name before clone heuristic, which has led to the highest recall, also leads to the highest lower bound of the precision (97% for ConQAT, 99% for Subclipse). For ArgoUML and Autofocus, the lower bound drops as both systems have a large number of implicit moves which were counted as false positives (see Table 2). However, even if all implicit moves were false positives, the precision would still be 71% or 72% respectively.

Developers' Rating. Table 6 shows the precision for the sample moves in the survey based on the developers' rating. For both survey objects, the clone heuristic performs better than the name heuristic, which achieves a precision of 92,5% for ConQAT and 90% for Autofocus 3. Developers occasionally disagreed with the name heuristic for the following reasons: In both systems, there are architectural constraints that different parts of the system need to have one specific file that has the same name and the almost same context. In ConQAT, these are the `Activator.java` and `BundleContext.java` which appear exactly once for each ConQAT bundle. The developers commented that those samples were clones but not necessarily a copy as the developers could not decide which predecessor is the right one. For Autofocus, developers commented on similar cases: Although the newly added class was very similar to the suggested predecessor, they did not consider it as a copy because it was added in a very different part of the system and should not share the history of the suggested predecessor.

For ConQAT, the clone heuristic achieves a precision of 97.5%, which is clearly above the precalculated lower bound of 91% In some cases, the clone heuristic outputs a different result than the repository due to the following reason: In ConQAT's history, the developers performed two major moves by copying the system to a new location in multiple steps. Each step was recorded in the repository and did not change a file's content. The clone heuristic chose a file as predecessor, which was in fact a pre-predecessor (an ancestor) according to the repository. In other words, the clone heuristic skipped one step of the copying process. However, developers still considered these results as correct. For Autofocus, the developers never disagreed with the clone heuristic, hence, it achieves a precision of 100%.

All samples with a random origin were evaluated as *incorrect*, showing that the participants evaluated with care and did not only answer *correct* for all cases. In terms of inter-rater-agreement, the Autofocus developers gave the same answers on all samples, the ConQAT developers agreed except for two cases when the majority vote was applied. Considering the confidence on judging whether a file was moved or copied from the suggested predecessor, the Autofocus developers had an average confidence value (per developer) between 1.77 and 2.0. The ConQAT developers indicated on average a confidence between 1.86 and 2.0.

Statistical Approximation. We combine the results from the survey on implicit moves and the repository information on explicit moves to calculate a statistical, overall approximation of the heuristics' precision: For all moves that were detected by a heuristic and that were explicitly recorded in the VCS, we use a precision of 100% as we assume the explicit move information to be correct. For all implicit moves, we use the precision resulting from the developer survey. We weight both precisions by the relative frequency of explicit and implicit moves. Table 7 shows the final results. On both survey objects, the clone and the name-based heuristic achieve a very high precision of above 97% in all cases with only minor differences between a single execution of a heuristic or both heuristics in either order. As the best recall was

Table 8: Relevance

Name	Implicit head moves	Head files	Information gain
ConQAT	114	3394	3.3%
ArgoUML	741	1904	38.9%
Subclipse	31	872	3.5%
Autofocus 3	1067	4585	23.2%
JabRef	30	628	4.7%
JHotDraw 7	145	692	20.9%
Vuze	666	3511	18.8%

Table 9: Execution times of the algorithm

Name	Commits	Total [h]	∅ per commit [s]
ConQAT	9309	2.1	0.83
ArgoUML	11721	9.6	2.9
Subclipse	2317	2.8	4.3
Autofocus 3	4487	6.5	5.2
JabRef	1521	2.4	5.6
JHotDraw 7	435	0.9	7.6
Vuze	20762	29.5	5.1

achieved by the execution of the name heuristic before the clone heuristic and differences in precision are only minor, we conclude that this is the best execution sequence.

5.6 Relevance (RQ3)

5.6.1 Design

To show the relevance of our approach in practice, we evaluate for all systems in Table 1 how many implicit moves exist. We calculate the number of implicit moves for a file in the head revision divided by the total number of head files to show the information gain of our approach. This fraction can be interpreted in two ways–either as average number of moves per file in the head revision that would not have been detected without our approach or as percentage of all files in the head that had on average one undetected move in their history. This quantifies the error rate of current state of the art repository analyses would have without our approach.

5.6.2 Results

Table 8 shows the approach's information gain. The number of implicit moves found in head files is smaller than the overall number of implicit moves: files deleted during the history do not appear in the head but their moves still count for the overall number of implicit moves. The table shows that for some systems significant origin information would be lost without our approach: For ArgoUML, *e. g.*, the information gain is 39% which mainly results from the loss of history information during the migration from CVS to SVN. The same applies to Vuze, for which our approach yields to an information gain of 19%. Without our approach, the origin information of the CVS history would have been lost in both cases. Also the information gain on other systems, such as jHotDraw (21%) or Autofocus (23%), reveals that our approach reconstructs a large amount of missing information. However, our approach's relevance is not limited to CVS-to-SVN conversions but also applies to any other VCS change. Naturally, the relevance of our approach varies with different systems. For well-maintained histories such as ConQAT or Subclipse, the information gain of our approach is less significant. As we restricted the information gain to files in the head, we ignored implicit moves for deleted files.

5.7 Performance (RQ4)

5.7.1 Design

We measure the algorithm's performance with the total execution time and the average execution time per commit. The experiments were run on a virtual server with 4GB RAM and 4 CPU cores (Intel Xeon). The approach is parallelized only between projects but not within a project. We evaluate the performance of the experimental setup (running heuristics in parallel, Figure 4) which denotes an upper bound of the intended execution of the algorithm (running heuristics sequentially).

5.7.2 Results

Table 9 shows the total execution time and the average per commit. Many systems were analyzed in about two hours. For the longest-lived system (Vuze, 20762 commits in 10 years), the approach took a little more than a day to finish. The average time per commit ranged from below one second up to eight seconds per commit. We consider the execution times of the algorithm feasible for application in practice.

Compared to the reported performance of Lavoie's algorithm, [22], our analysis runs significantly faster: To analyze two versions of a system, Lavoie reports between 24 and 600 seconds depending on the size of the system (between 240kLoc and 1.2 MLoc). With our systems being in the same size range, we can compare two versions on average in at most 8 seconds (jHotDraw). Our approach benefits from the incremental clone detection. However, we did not conduct an explicit comparison to a non-incremental approach as this has already been done in [13].

6. DISCUSSION AND FUTURE WORK

After presenting the results, we discuss applications and limitations of our approach, deriving future work.

Applications. Our approach is designed to incrementally reconstruct the history of a source code file to provide complete history data. This is relevant for all static code analyses that mine the software evolution, such as type change analysis, signature change analysis, instability analysis [16] or clone genealogy extractors, clone stability analyses, or defect prediction based on change frequency. Furthermore, for monitoring software quality continuously in long-lived projects, our approach enables to show a precise and complete history for all quality metrics such as structural metrics, clone coverage, comment ratio, or test coverage.

Limitations. Our approach cannot detect reverts when a file was readded from a prior revision. Future work might solve this problem by keeping all data in the clone index. However, it remains debatable whether a complete file's history should contain the revert or stop at the reverting add as the developer had deleted the file on purpose.

We designed our approach for the main trunk of the repositories, excluding branches and tags. Including branches and tags into our approach requires a cross linking between the main trunk and other branches. Otherwise, the name-based heuristic produces wrong results when a file is added to a second branch because it will detect the file in the first branch to have a more similar path than the file in the trunk. Future work is required to establish the cross linking and to integrate this information with the heuristics.

Ongoing work comprises the comparison of our approach to the tracking mechanism of Git. For future work, we plan to conduct a quantitative and quality comparison.

7. THREATS TO VALIDITY

The recall is based only on the explicit move information and, hence, provides only an approximation of the real recall. However, to our best knowledge, there is no other way of evaluating the recall as even developers would not be able to tell you all moves, copies, and renames if the history of their software comprises several years. To make the recall approximation as accurate as possible we chose systems with a large number of explicit moves in their history.

To get a statistical sample to approximate the overall precision of the approach, we conducted a developer survey for ConQAT and Autofocus. As ConQAT is our own system, one might argue that the developers do not provide accurate information to improve the results of this paper. However, the first author of this paper was not a developer of ConQAT and did not participate in the case study.

We evaluated the use of our tool only on SVN and Java. Generally, our approach is independent from the underlying VCS and programming language. It can be also used with other VCS such as Git or TFS.

8. CONCLUSION

We presented an incremental, commit-based approach to reconstruct the complete history of a source code file, detecting origin changes such as renames, moves, and copies of a file during evolution. With a precision of 97-99%, the approach provides accurate information for any evolution-based static code analysis such as code churn or code stability. A recall of 98% for almost all systems indicates that the heuristics can reliably reconstruct any missing information. A case study on many open source systems showed that our approach succeeds to reveal a significant amount of otherwise missing information. On systems such as ArgoUML for example, 38.9% files had an incomplete history as recorded in the repository. Without our approach, current static code analyses that rely on file mapping during evolution would have made an error on more than every third file. In contrast to many state of the art tools, our approach is able to analyze every single commit even in large histories of ten thousand revisions in feasible runtime.

9. REFERENCES

[1] Git. `http://www.git-scm.com/`. [Online; accessed 2013-12-03].

[2] G. Antoniol, M. D. Penta, and E. Merlo. An Automatic Approach to Identify Class Evolution Discontinuities. In *IWPSE '04*, 2004.

[3] M. Asaduzzaman, C. Roy, K. Schneider, and M. D. Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *ICSM'13*, 2013.

[4] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt. A Framework for Incremental Quality Analysis of Large Software Systems. In *ICSM'12*, 2012.

[5] G. Canfora, L. Cerulo, and M. D. Penta. Identifying Changed Source Code Lines from Version Repositories. In *MSR'07*, 2007.

[6] Davies, J. and German, D. M. and Godfrey, M. W. and Hindle, A. Software bertillonage: finding the provenance of an entity. In *MSR'11*, 2011.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00*, 2000.

[8] N. Göde and J. Harder. Clone stability. In *CSMR'11*, 2011.

[9] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *IWPSE'02*, 2002.

[10] M. Godfrey and L. Z. Using origin analysis to detect merging and splitting of source code entities. *Software Engineering, IEEE Transactions on*, 31(2), 2005.

[11] Godfrey, M. W. and German, D. M. and Davies, J. and Hindle, A. Determining the provenance of software artifacts. In *IWSC'11*, 2011.

[12] L. Heinemann, B. Hummel, and D. Steidl. Teamscale: Software Quality Control in Real-Time. In *ICSE '14*, 2014.

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

[14] J. Hunt and W. Tichy. Extensible language-aware merging. In *ICSM'02*, 2002.

[15] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *ISSRE'96*, 1996.

[16] M. Kim and D. Notkin. Program Element Matching for Multi-version Program Analyses. In *MSR'06*, 2006.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE'05*, 2005.

[18] S. Kim, K. Pan, and E. J. Whitehead, Jr. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *WCRE'05*, 2005.

[19] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Recovering the Evolution Stable Part Using an ECGM Algorithm: Is There a Tunnel in Mozilla? In *CSMR'09*, 2009.

[20] J. Krinke. Is Cloned Code More Stable than Non-cloned Code? In *SCAM'08*, 2008.

[21] J. Krinke. Is cloned code older than non-cloned code? In *IWSC'11*, 2011.

[22] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou. Inferring Repository File Structure Modifications Using Nearest-Neighbor Clone Detection. In *WCRE'12*, 2012.

[23] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*, 2008.

[24] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE'05*.

[25] Q. Tu and M. W. Godfrey. An Integrated Approach for Studying Architectural Evolution. In *IWPC'02*, 2002.

[26] F. Van Rysselberghe and S. Demeyer. Reconstruction of Successful Software Evolution Using Clone Detection. In *IWPSE'03*, 2003.