Revealing Missing Bug-Fixes in Code Clones in Large-Scale Code Bases

Martin Pöhlmann, Elmar Juergens
CQSE GmbH, Germany
{poehlmann,juergens}@cqse.eu

***Abstract*—If a bug gets fixed in duplicated code, often all duplicates (so called clones) need to be modified accordingly. In practice, however, fixes are often incomplete, causing the bug to remain in one or more of the clones. In this paper, we present an approach to detect such incomplete bug-fixes in cloned code. It analyzes a system's version history to reveal those commits that fix problems. It then performs incremental clone detection to reveal those clones that became inconsistent as a result of such a fix. We present results from a case study that analyzed incomplete bug-fixes in six industrial and open source systems to demonstrate the feasibility and defectiveness of the approach. We discovered likely incomplete bug-fixes in all analyzed systems.**

***Keywords*-code clones, clone detection, incomplete bug-fixes, evolution analysis, repository analysis**

I. Introduction

"I had previously fixed the identical bug [...], but didn't realize that the same error was repeated over here." [1] This excerpt of a commit message is a common verdict of developers who unveil inconsistencies in duplicated (cloned) code. As research in software maintenance has shown, most software systems contain a significant amount of code clones [2]. During maintenance, changes often affect, and thus need to be carried out, on all clones.

If developers are not aware of the duplicates of a piece of code when they make a change, the resulting inconsistencies often lead to bugs [3]. Awareness of clones in a system is especially important, if a developer fixes a bug that has been copied to different locations in the system. Those clones that are not fixed continue to contain the bug. Many studies have reported discovery of errors in clones in practice, often due to incomplete bug-fixes [3, 4, 5, 6, 7, 8, 9].

To avoid such incomplete bug-fixes, *clone management* [2] approaches have been proposed to alert developers of the existence of clones when they perform changes. However, while such approaches can possibly ease future maintenance, they are of no help with those incomplete bug-fixes that have happed in the past. How can we detect such inconsistent bug-fixes that are already part of the source code of a system?

One approach to detect incomplete bug-fixes in cloned code is to search for clones that differ from each other, e.g. in modified or missing statements. These differences could hint at incomplete bug-fixes. Several clone detection approaches exist that can detect clones with differences (so called type-3 clones) [10]. Unfortunately for the precision of this approach, however, not every difference between a pair of clones hints at a bug. In many cases, a developer copy & pastes a piece of code and modifies it intentionally, since the new copy is required to perform a slightly different function.

During the past five years, we have inspected clones in numerous industrial and open-source systems. Most of them contain substantial amounts of clones—including type-3 clones. Searching for incomplete bug-fixes by manually inspecting type-3 clones is inefficient, simply because many of the differences were introduced intentionally, often already during the creation of the clone. To reveal incomplete bug-fixes more efficiently, we ideally require an approach that can (at least to a certain degree) differentiate between intentional and unintentional differences between clones.

This paper proposes a novel approach to reveal inconsistent bug-fixes. It iterates though the revision history of a system and classifies changes as bug-fixes, if the commit message contains specific keywords like *bug* or *fix*. It then tracks the evolution of code clones between revisions to detect clones which become inconsistent *as consequence of a fix*. Our assumption is that such inconsistencies have a high likelihood of being unintentional. The case study that we have performed for this paper confirms this assumption.

Furthermore, in contrast to clones detected on a single system version, this approach provides information on which change, by which author and for which defect, caused the difference between the clones. From our experience, this information substantially supports developers in judging if and how to resolve differences between clones.

Problem: Bug-fixes in cloned code are often incomplete, causing the bugs to remain in the system. We lack approaches to efficiently reveal such incomplete bug-fixes.

Contribution: This paper presents a novel approach for detecting missing bug-fixes in code clones by combining clone evolution analysis with information gathered from the version control system.

We have implemented the approach based on the incremental clone detection functionality of the open-source program analysis toolkit ConQAT[1]. We have evaluated it in a case study on six industrial and open-source systems in Java and C#. The results of the case study show that the approach is feasible and does reveal missing bug-fixes.

[1] http://conqat.org

II. Related Work

This section gives an overview of approaches to reveal incomplete or missing bug-fixes. We distinguish between work concerning system evolution and clone detection.

A. Evolution-based

Kim et al. [11] proposed a tool called *BugMem*, which uses a database of bug and fix pairs for finding bugs and suggesting fixes. In particular, this system-specific database is built via an on-line learning process, since each revision of the version control system is scanned for a commit message hinting at a bug-fix. As suggested by Mockus and Votta [12] they use the terms "Bug" or "Fix" for identifying bug-fixes, as well as reference numbers to issue-tracking software like Bugzilla. For each fix-commit, the changeset data is extracted and separated into *code-with-bug* and *code-after-fix*. These code regions are normalized to generalize identifiers and stored in the database.

We use the same method for scanning the version history for interesting terms, but refrain from including references to bug tracking reports, since some systems track both bugs and feature requests with such tools. For including them, further work is required to distinguish bugs and requests, as well as making the data available offline.

Zimmermann et al. [13] took a similar approach by mining data from a version control system for a change recommendation system. Their goal is to suggest in an IDE changes and fixes [14] in the manner of shopping applications: "Programmers who changed these functions also changed...". The precision of meaningful suggestions is 26%. From a user-perspective the main difference to our approach is, that the recommendation tool tries to prevent bugs by suggesting changes upon modification of files in the IDE. Yet, the amount of wrong recommendations is rather high.

B. Clone-based

Juergens et al. [3] inspected a set of gapped clones for incomplete bug-fixes using their open source tool suite ConQAT. They proposed an approach for identifying gapped code clones using an algorithm based on suffix-trees and evaluated it on several large-scale systems. The results of this study show an average precision of 28% for detecting unintentional inconsistent clones. Nevertheless, the tool reported for all but one system over 150 inconsistent clones, which is a large amount for initial analyses.

The approach proposed in this paper also builds upon ConQAT, but uses another technique for detecting inconsistent code clones using an index-based algorithm in conjunction with evolution analysis. Hence, we compare our approach to that from Juergens et al. in terms of reported inconsistencies, precision and execution time.

C. Combined – Clone-evolution-based

APPROX of Bazrafshan et al. [15] is a tool for searching arbitrary code fragments in multiple versions and branches for similar fragments to find missing fixes. The search is based on code clone detection, but limited to search for code fragments similar to a search term. In contrast to our approach APPROX requires developers to know beforehand which code snippet contains a bug-fix and is of interest.

Duala-Ekoko and Robillard [16] created an extension for the Eclipse IDE, which reads a clone report from SimScan[2] and tracks the location of the clones automatically as code changes in the editor or between revisions. The approach focuses more on getting an overview about all related clones while editing a file, since they provide automated edit propagation to other clone siblings as well.

In contrast, Kim et al. [17] analyzed clone genealogies by combining CCFinder [18] with a clone tracking approach. In a case study they showed that only up to 40% of all clones are changed consistently during system evolution. Canfora et al. [1] used another clone detection tool as well as other study objects and reproduced the results from Kim et al. with about 43% of all clones being consistently modified. In detail, the inconsistencies sum up to 67% whereas 14% of these were propagated later to become consistent again.

Also Göde and Rausch [19] analyzed the evolution of three open source systems during a time frame of five years. For this, they used an iterative clone detection and tracking algorithm, and showed as well, that 43.1% of all changes to clones are inconsistent with 16.8% being unintentional inconsistent. Again, the total amount of reported inconsistencies is with 131 clones quite high and includes lots of false positives with respect to unintentional inconsistencies. As we go further and filter the revisions by commit message, we significantly reduce the amount of false positives.

Geiger et al. [20] presented an approach for identifying interesting correlations between code clones and change couplings mostly with respect to different subsystems. Change couplings are files that are committed at roughly the same time, from the same author and with the same commit message [21]. Nevertheless, they conclude that a correlation is too complex to be easily expressed and more information is needed to identify harmful clones. In our approach we will not correlate change coupling, but use a prediction of which commit is a fix based on its commit message.

III. Revealing missing Bug-Fixes

This section provides an in-depth explanation of how the analysis process of the proposed approach works. As depicted in Fig. 1 the process is an iteration over the available revisions of the version control system in order to simulate the source code evolution. For each iteration, several steps are performed to identify incomplete bug-fixes in code clones. At the end of each cycle, the iterator is queried for the next revision and a new detection starts. As soon as no newer revision is available, the bug detection results are reported and the analysis process terminates.

A. Get Next Revision

Upon the start of the analysis the version control system is queried for all or a subset of available revisions. During the

[2] http://blue-edge.bg/simscan

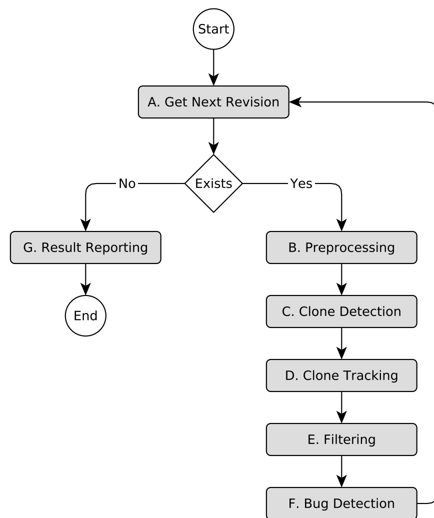

Fig. 1: Overview of the iterative bug detection process

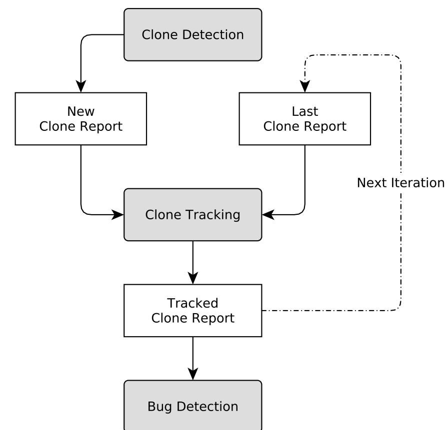

Fig. 2: Clone tracking dataflow process

iteration loop, the revisions are checked out in chronological order and metadata containing the commit message is handed over to the next steps.

B. Preprocessing

First, the source code is read from the disk into memory. Regular expressions are used as include and exclude filters for omitting generated and test code, since the former does not contain manual modifications and the latter is not interesting for production use. We further strip unnecessary statements, like package identifiers or include statements as these are unlikely to contain bugs. Finally, the source code is normalized into a generic representation which is insensitive to method names, variable names and literals.

C. Clone Detection

For detecting code clones, we use an algorithm using a hash index [22], which allows incremental updates of the clone data with high performance. For each revision, we gather the list of altered, added and deleted files and remove all data from the index, that belongs to these files. Afterwards, these files are added to the index again, but with updated content. Thus, we can keep most of the data and update just a small fraction depending on the changeset size.

Detection is configured to keep clones from crossing method boundaries. Hence, we can minimize the amount of semantically not meaningful code clones. Moreover, we do not take gapped clones (type-3) into account, which contain statement additions, removals and modifications after normalization. Including gapped clones in this step will not allow us to determine if the inconsistency in such a clone is related to a bug-fix.

D. Clone Tracking

This step performs the actual evolution analysis for code clones. For this, the reported clones from the clone detection step are mapped to those from the previous iteration cycle (*cf.* Fig. 2). Modifications that are performed inconsistently between clones, turns a group of ungapped cloned into a group of gapped clones. Moreover, these gapped clones are marked as *modified in this revision*.

The clone tracking is also performed by ConQAT and roughly follows the approach proposed by Göde and Koschke [23]. It first calculates the edit operations of a code clone between two consecutive revisions. Then it propagates these edit operations to the clones of the current revision, so the clone positions are updated accordingly. Afterwards, the updated clones are mapped to those from the the current detection step: First, those clones are matched, which positions do not differ. Second, a fuzzy coverage matching is performed on the remaining clones, that determines whether one clone covers an other and reports clones with modifications and gaps, that are of interest for the proposed bug detection approach.

E. Filtering

All clones without gaps are filtered, because they cannot contain incomplete bug-fixes. We also remove clones that differ too much with regards to their length. The threshold we choose is 50% around the average length of all clone instances of a group.

For example, a group of clones with two instances of length 23 and one of length 8 has an average clone length of 18. As the length of the shortest instance lies outside the 50% interval around this average length ($[9, 27]$), it is removed from the group. The other instances remain, because they lie within the interval. If all clones of a group lie outside this interval, the entire group is discarded.

F. Bug Detection

Clones are classified in those that contain an incomplete bug-fix and those that do not, as outlined in Fig. 3. To achieve this, the version control system is first queried for the commit message of the current revision. In the message we search for terms like *fix*, *defect* or *bug* that may indicate a bug-fix. If such a term is found, the whole commit and its modifications are seen as bug-fix commit. Mockus and Votta [12] proposes

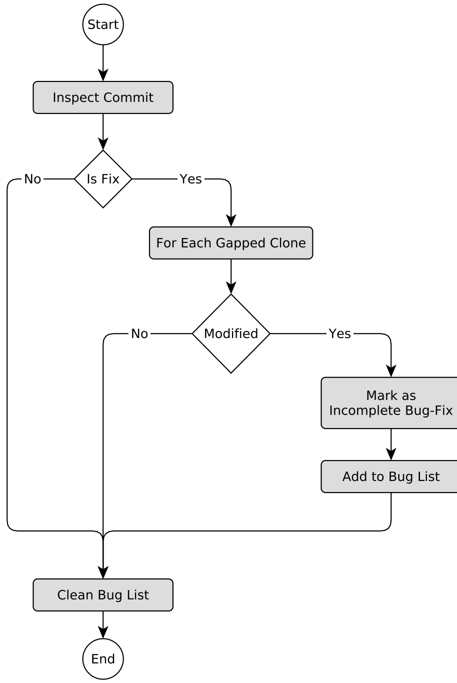

Fig. 3: Incomplete bug-fix detection flow

an even more extensive approach for finding fixes in commit messages. In our case, system specific terms were sufficient.

Afterwards, the list of code clones is searched for those that were marked as *modified in this revision* in the clone tracking step. For those, the approach suggests that the clone contains an incomplete fix, as the commit is a bug-fix and the clone was inconsistently modified in this revision. Consequently, it will be added to the global list of all incomplete bug-fixes. Finally, we also need to clean this list of incomplete bug-fixes as soon as a clone became consistent again or vanished. We continue with a new iteration loop, as long as newer revisions are available.

G. Result Reporting

After the revision iteration terminates, the result reporting is the last step. It writes all incomplete bug-fixes into a XML file for manual inspection with the ConQAT Clone Workbench. The report contains details about the location of clone instances in the source code, which includes file name, start and end line, as well as the position of gaps. Moreover, also information about the revision that caused the clone to become inconsistent are stored, namely the commit message and the revision identifier.

H. Performance Optimization: Revision Compression

After analyzing the version history of some software systems, we found that only a small percentage of the commits represent a bug-fix. In the studied systems roughly 25%. We can exploit this for a notable performance improvement, since we only need to inspect each commit that represents a bug-fix. All revisions between bug-fixes can be compressed into a single composite revision, as depicted in Fig. 4. These

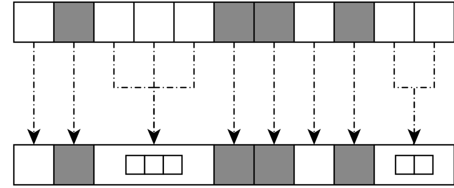

Fig. 4: Compressing non-bug-fix revisions (white) into single commits. Revisions including a bug-fix (gray) are not compressed.

compressed revisions are created by appending the commit messages and merging the changesets of altered files.

The general performance improvement can be described as follows: Let R be the total number of commits and F be the amount of fixes in a system (determined from the commit message) with of course $F \leq R$. Then we have to inspect R revisions without compression. With revision compression at most $2 * F + 1$ revisions have to be inspected, whereas R is still an upper limit that cannot be exceeded. For a system with a ratio of one fix per four commits, we can skip at least 50% of all revisions.

IV. Case Study

This section presents a case study which examines the amount of fixes in code clones of industrial and open source software systems. Moreover, it evaluates how well bugs can be detected with the proposed clone evolution approach and compares it to gapped clone detection on a single system revision.

A. Research Questions

We investigate the following five research questions:

RQ 1: Can fixes be determined from commit messages?
The first question serves as fundamental analysis of how fixes can be detected solely from analyzing commit messages. It analyzes if a set of a few keywords can be chosen in a manner that they identify a commit as fix.

RQ 2: Which amount of code clones is affected by fixes?
If inconsistent changes to clones do not occur in software systems, further analyses do not make sense.

RQ 3: How many inconsistently fixed clones qualify as bug candidates?
This question investigates whether the proposed approach is appropriate for detecting bugs and how many false positives are returned. For a toolkit in production use, a high precision in detecting incomplete bug-fixes is desireable.

RQ 4: What is the impact of the commit changeset size on the bug-finding precision?
Commits with a lot of modified files are likely to contain refactorings, feature additions or branch merges besides the actual fix. We suspect that more false positives are reported and try to give evidence by analyzing the precision with respect to limited changeset sizes.

TABLE I: List of the analyzed software systems

System	Organization	Language	History (years)	Size (kLOC)	Commits
A	Munich Re	C#	1.5	81.4	823
B	Munich Re	C#	1.5	370.6	638
C	Munich Re	C#	5	652.7	7483
D	Aol	Java	1.5	47.5	1449
Banshee	Novell	C#	7	165.6	8097
Spring	VMWare	Java	4	417.6	5034

RQ 5: How does evolution-based bug-detection compare to gapped clone detection on a single revision?
Finding incomplete bug-fixes can also be achieved by gapped clone detection. Hence, the question arises if the overhead of analyzing history information is justified compared to gapped clone detection in terms of precision and the amount of reported inconsistently fixed clones needed to be inspected by a developer or quality assurance engineer.

B. Study Objects

The case study was performed on six real-world software systems as listed in Table I. The reason for choosing these projects was on the one hand the requirement of having an evolved version history, on the other hand we need access to the version control system even for non-open-source projects. Thus, we relied on own contacts for industry code. In contrast, *Banshee* and *Spring* are available as open source systems and maintained by Novell and VMWare.

All systems are written by different teams, have individual functionality and evolved independently. They also differ in size and age. System A, B and C are owned by Munich Re, but are developed and maintained by different suppliers. They are written in C# and used for damage prediction and risk modeling. System D is an Android application developed by AOL. The two open source applications are the popular open source cross-platform audio player *Banshee*[3] written in C# by more than 300 contributors and the Java enterprise application framework *Spring*[4] developed by over 50 contributors. All systems are actively developed and in production use.

C. Determining Bug-Fixes — RQ 1

Design and Procedure: This question explores how well fixes can be determined from the commit messages of a version control system. To answer it, we manually inspected commit messages from all study objects and identified reoccurring terms which are used in bug-fix commits. These keywords were compiled to a regular expression so that it matches any of the terms.

Results: The manual inspection of all six systems yielded the following list of keywords suitable for identifying bug-fixes: *Fix, Bug, Defect, Correct*. As system A, B and C are developed by German engineers, some of the commit messages are written in German as well. This yields an extended set of keywords also containing the German words *Fehler*, *Defekt*, *behoben* and *korrigiert*. Additionally, the word *correct* seems to be often misspelled by developers as *corect*, so we also took this variant into account.

[3]http://banshee.fm
[4]http://springsource.org/spring-framework

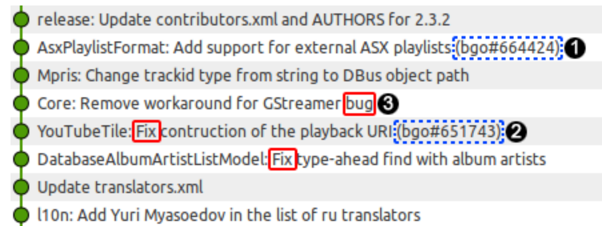

Fig. 5: Excerpt of commit logs from Banshee

TABLE II: Total amount and percentage of identified fixes

System	Commits	Fixes	Fixes (%)
A	823	194	23.6
B	638	203	31.8
C	7483	1754	23.4
D	1449	326	22.5
Banshee	8097	2016	24.9
Spring	5034	648	12.9

We decided against identifying commits as fixes solely from the presence of a reference to a bug-tracking software. As shown in Fig. 5, such references are mostly given by a number code representing the identifier of a bug or feature request in the issue-tracker. In the example, only the second commit with a bug-tracking identifier (❷) is a bug-fix, while the first one (❶) references a feature request. Thus, we suspect to threaten precision by generally taking these identifiers into account. All in all, we can compile the following regular expression to match a commit message against:

```
(?is).*(fix|bug|defe(c|k)t|fehler|behoben|
co(r|rr)ect|ko(r|rr)igier).*
```

Table II summarizes the amount of fixes detected by the above regular expression in each analyzed system. According to this result, almost every fourth commit is a fix.

Discussion: The results show that detecting bug-fixes from commit messages works reasonably well, at least for the systems we studied. Of course, the list of keywords used for detection varies depending on the software system and the language the developers speak. The proposed terms provide an initial starting point. But as soon as the bug detector is used on other systems, the terms need to be customized and tailored.

Still, it might happen that a commit is falsely identified as bug-fix, as depicted in Fig. 5 (❸). However, this seems to be a rare problem and in case of the above example the modification was at least related to a bug-fix.

D. Amount of Incomplete Fixes — RQ 2

Design and Procedure: The second research question investigates the amount of code clones that is affected by fixes and became inconsistent thereby. To answer it, we counted both, the total amount of incomplete fixes during project evolution, as well as those that were still present in the latest revision of the system history. Therefore, our bug detection

TABLE III: Evolution of incomplete bug-fixes

System	Total Incomplete Fixes	Still Present
A	48	28
B	60	50
C	108	61
D	26	15
Banshee	112	21
Spring	35	23

TABLE IV: Results of the bug detection for each system

System	Total Clones	Inconsistently Fixed Clones	Bug Candidates	Precision	Time (min)
A	200	28	11	0.39	2.7
B	765	50	9	0.18	23
C	778	61	23	0.38	116
D	170	15	6	0.40	5
Banshee	165	21	5	0.25	119
Spring	518	23	4	0.17	127

toolkit has been slightly modified to deliver these statistics. The configuration remained as described in Section III with a minimal clone length of 7 statements. Additionally, groups of clones with more than three instances were filtered, because for them the tracking approach is unreliable. Including these clones remains for future work.

Results: Table III shows for each system the amount of incomplete bug-fixes in code clones. For all systems less fixes are present in the last revision than occurred in total. This is due to corrected inconsistencies or completely removed clones. Moreover, a code clone can also be affected by more than one bug-fixed and appear multiple times in the above statistic.

Discussion: Depending on the system, we were able to reduce the total amount of code clones to a small fraction compared to all gapped clones, as shown by Table VI later in the case study.

Comparing the low amount of incomplete fixes for Banshee, which are still present in the last revision, to all inconsistencies found during the analysis shows that the detection algorithm is not stable with regards to big refactorings. The Banshee developers partially restructured the project with regards to sub components, which caused some detected bugs to be lost during tracking. Even gathering renamed files from the version control system will not completely eliminate this issue, since code may be exchanged between files as well. Also system C suffered from this in a minor extent.

Moreover, The Banshee Git repository contains lots of branching on the main development line, but the analysis loops through all commits in a sequential order provided by the *jGit* library. Thus it might switch between branches for consecutive runs and causing some bugs to disappear, due to tracking issues. An adapted evolution analysis is needed for this case, that traverses merged branches separately. This requires some major rework for the entire revision iteration and bug detection process, which is left for future work.

E. Detection Precision — RQ 3

Design and Procedure: This question investigates bug-detection precision. The inconsistently fixed clones, which were gathered with RQ 2, were manually inspected by the researcher and separated in false positives and bug candidates. For answering the question, we calculated the precision of the bug detector according to Equation 1.

$$precision = \frac{\#\ bug\ candidates}{\#\ inconsistently\ fixed\ clones} \qquad (1)$$

The decision if an inconsistently fixed clone qualifies as bug candidate was made upon comparing the source code of clones using the ConQAT clone workbench for Eclipse and manual inspection of the commit message and modifications. This is shown in Fig. 6. We used a Laptop with a 2.2 GHz Quadcore CPU and 4 GB of RAM running a 32 Bit Ubuntu Linux with Oracle JDK 7 for the detection.

Our goal is to cost-effectively find missing bug-fixes. We are thus willing to sacrifice recall for higher precision and leave analysis of recall (if feasible at all) for future work.

Results: Table IV summarizes the total amount of code clones, which were detected during history evolution and present in the last revision, as well as the inconsistently fixed clones. Additionally, it lists the amount of bug candidates resulting from manual inspection, the precision calculated with Equation 1 and the time the detection took in minutes.

Besides Spring, the detection algorithm identifies roughly 10% of all clones as inconsistently fixed. The manual inspection of the researcher revealed that 17% to 40% of these reported clones are classified as bug candidates. The execution times varies with regard to the project size and the amount of revisions chosen for evolution analysis. Still, all analyses were performed in less than three hours.

Discussion: The lowest result with respect to precision shows system B and Spring. The latter has a very low rate of bug-fixes in general and just 23 bugs were reported out of over 5000 revisions with more than 400.000 lines of code per revision. According to the documentation it has very strict guidelines[5] for third party contributions with respect to coding style, unit-testing and patch submission and similar rules seem to apply for internal work as well.

For system B, lots of false positives were introduced by big commits that *"fixed"* coding style related issues. We did not count these as bugs. The same problem also decreases the results for system C. RQ 4 tries to alleviate this problem by limiting the amount of modified files per commit in order to be recognized as bug-fix.

F. Changeset Size Impact — RQ 4

Design and Procedure: This question analyzes how the size of the commit changeset influences the bug-finding precision. Analogous to RQ 3, we answered this question by inspecting the returned inconsistently fixed clones and determining the precision according to Equation 1. Therefore, the detection toolkit was executed with the same parameters as described for RQ 3. Additionally, a minimum and

[5]https://github.com/SpringSource/spring-framework/wiki/Contributor-guidelines

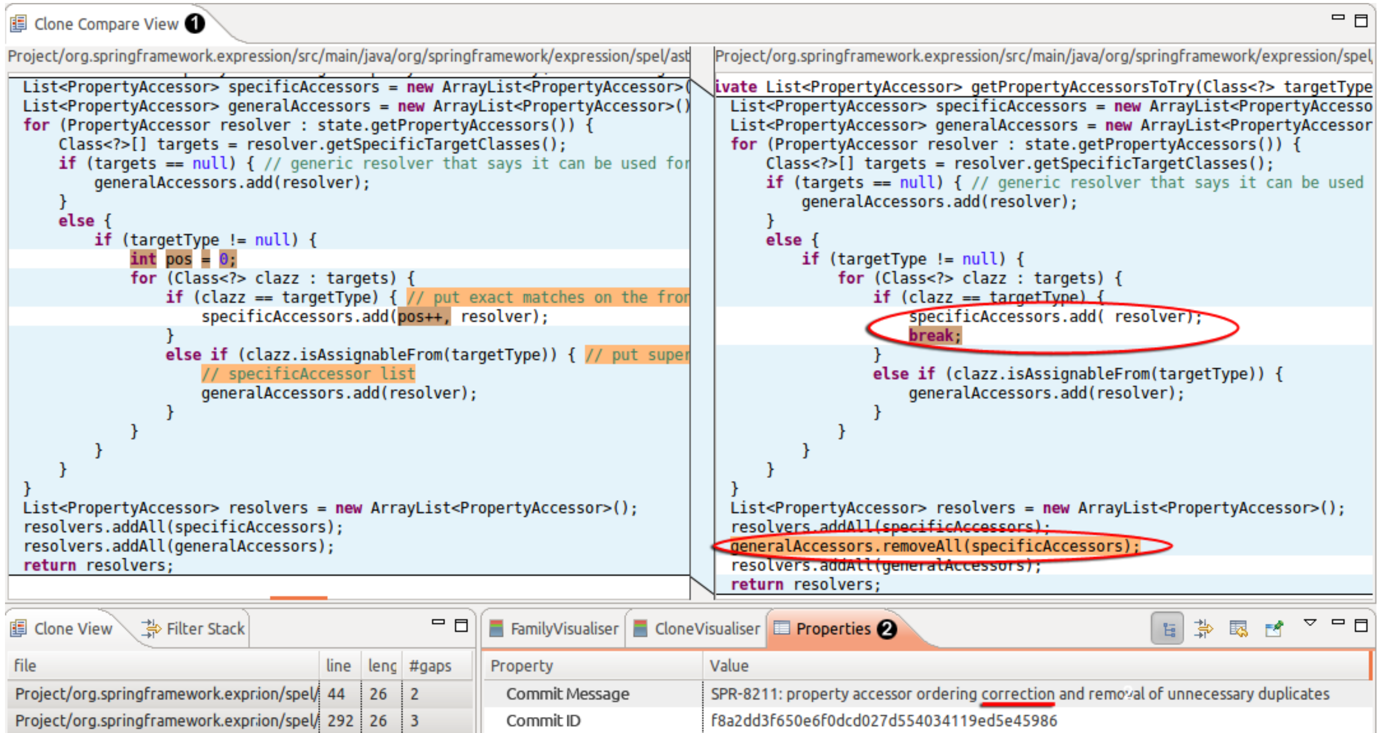

Fig. 6: Screenshot of the ConQAT clone workbench including a clone compare view (❶) and metadata about the commit this inconsistency was introduced by (❷)

maximum threshold to the changeset size of a commit is applied, which is evaluated at the time of revision compression. The changeset size is limited with window of size 5 sliding from 1 to 21. This results in the following intervals: $[1, 5], [6, 10], [11, 15], [16, 20], [21, \infty[$

Results: The results are summarized in Table V and grouped by the limit interval. One can clearly see that the precision in intervals $[1, 5]$ and $[6, 10]$ is almost two times higher than the average precision from Table IV and ranges from 30% to 60%. In contrast, the precision drops off significantly for commits with changeset sizes larger than 10. As discussed, the style related fixes in system B which lowered the results for RQ 3 fall into this category. Nevertheless, the systems performing worse before did not catch up to the other systems in terms of precision, but an increase is still noticeable.

It is worth a remark that the sum of the reported bugs or bug candidates of each system may not be equal to the results from Table IV, since a clone can be altered by fixes of different changeset sizes. Thus, some clones are listed multiple times.

Discussion: Applying a maximum limit to the changeset size of at most 10 altered files, the precision of the approach could almost be doubled and bugs are detected with an acceptable precision of 30% to 60%. We consider this sufficient for use in real-world assessments.

G. Gapped Clone Detection Comparison — RQ 5

Design and Procedure: This research question compares the evolution based bug-finding approach to the less time-consuming gapped clone detection. To answer it, we run a

TABLE V: Results with limits applied to the changeset size

Limit	System	Inconsistently Fixed Clones	Bug Candidates	Precision
$[1, 5]$	**A**	11	8	0.73
	B	20	5	0.25
	C	35	13	0.37
	D	11	5	0.45
	Banshee	12	4	0.33
	Spring	17	5	0.29
$[6, 10]$	**A**	4	2	0.50
	B	5	2	0.40
	C	11	9	0.82
	D	3	1	0.33
	Banshee	4	1	0.25
	Spring	3	1	0.33
$[11, 15]$	**A**	0	—	—
	B	1	0	0.00
	C	5	1	0.20
	D	0	—	—
	Banshee	2	0	0.00
	Spring	0	—	—
$[15, 20]$	**A**	3	1	0.33
	B	0	—	—
	C	0	—	—
	D	1	0	0.00
	Banshee	0	—	—
	Spring	0	—	—
$[21, \infty[$	**A**	7	0	0.00
	B	26	3	0.12
	C	5	2	0.40
	D	0	—	—
	Banshee	3	0	0.00
	Spring	0	—	—

TABLE VI: Results for finding bugs with gapped clone detection

System	Reported Clones	Inspected (%)	Bug Candidtates	Precision
A	42	42 (100%)	13	0.31
B	219	109 (50%)	26	0.23
C	192	96 (50%)	25	0.26
D	60	60 (100%)	15	0.25
Banshee	34	34 (100%)	8	0.24
Spring	166	83 (50%)	17	0.20

gapped clone detection with ConQAT on the study objects and filter the returned clones to contain at least one modification. The parameters from RQ 3 are used again with the following parameters being chosen especially for the gapped clone detection according to Juergens [24]: The gap ratio must be at most 20% and the edit distance has a maximum of 5 edits.

Finally, we determined the precision of finding bugs in this set of clones with Equation 1. The results are then compared to those of RQ 3 and RQ 4. Due to the large amount of reported inconsistently fixed clones for some systems, we just inspected a percentage of the reported clones for bug candidates, which are randomly chosen from the entire result set.

Results: Table VI presents the results for the gapped clone detection executed for each of the study objects. Compared to the results from the evolution based approach, one can see that the overall precision is more homogeneously ranging from 20% to 30%. Hence, there is no significant difference to the evolution analysis without taking changeset sizes into account. Nevertheless, the detection reported 2 to 6 times more clones we have to manually inspect. As positive side effect, also more bug candidates were detected. As for the enhanced approach with limits applied to the changeset size, the gapped detection performs clearly worse in terms of precision.

Discussion: The results of the evolution based approach are gained with a time consuming analysis that took over two hours for some systems. Hence it is valid to ask if a simple gapped clone detection, which only takes two minutes to execute, does the job of finding inconsistent clones with similar precision.

Just by taking the results from the basic evolution based approach one may concede this point. Nevertheless, the gapped detection was performed with additional parameters that already filtered lots of false positives. Not applying those filters increases the size of inconsistently changed clones for system A from 42 to 309. For the evolution-based approach, we did not apply these filters and may gain further precision by applying them. Furthermore, when it comes to comparison with limited changeset sizes, the precision is clearly higher than for gapped clone detection. Thus, we consider the long execution times as justified.

Besides precision, there are other valid arguments for favoring the history based approach: First, we can gain important information about the fix from the corresponding commit messages. Furthermore, in an continuous scenario, we just have to update the clone index for altered files and can thus return new results in almost real-time. Related to the future extensions, the revision information can also be used to get knowledge about the person who introduced the inconsistency.

It is noteworthy that the gapped detection unveiled some bugs, we did not encounter before, but vice-versa the evolution based approach reported some bug candidates not found by the gapped approach as well. Hence, also a combination of both methods could be beneficial.

H. Threats to Validity

This section gives an overview of internal and external threats to validity of the case study and how we tried to mitigate them.

Internal Validity: The main error source for the case study may be determining if an inconsistently fixed clone is a bug candidate, since the researcher has no deep knowledge of how the analyzed systems are built and components work together. We tried to mitigate the threat by inspecting the results twice and concluded with the same results. For future work we also want to verify the results by developers.

For answering the research questions, we did not take recall into account. Yet, this is no problem with regards to the aim of the proposed toolkit. The key requirement is to find bugs with high precision combined with context information. This set of tool-reported bugs should contain as less false positives as possible to minimize manual inspection efforts. As long as the time spent searching for bugs is justified by the bugs we find, we do not mind how many we miss: the time invested into finding bugs paid off.

Another group of threats concerns the program evolution. Depending on the version control system used, fixes that happen on feature branches are not visible on the main branch after being merged. All systems that were imported from Subversion and Microsoft Team Foundation server suffer from this problem, whereas the Git based systems Banshee and Spring do not. Similarly, we will not detect clones that were newly created and a fix was applied to the code before committing to the version control system again. These problems are more or less technical restrictions that cannot be prevented and thus the set of reported bugs may be smaller than the actual set of inconsistent fixes that were applied to code clones.

A further problem may be clone false positives, which are code regions that are syntactically similar to each other but contain no semantic similarity. Examples are *e.g.* lists of getters and setters with different identifiers. We included those false positives in the results of the case study and counted them as not representing a bug candidate. Doing so, we penalize the precision of the bug-finding tool.

Finally, the list of keywords for identifying bugs may not be exhaustive. Again, our aim is not to find all possible bugs, but a subset with a high precision. Moreover, adding new keywords for other systems is easy.

External Validity: The systems chosen for the case study as study objects may not represent an average software system. Yet, for closed-source systems, we are limited to existing industry contacts. However, all systems are developed by

different teams and for different purposes as described in Section IV-B. Moreover, RQ 1 and RQ 3 showed that they also have different characteristics in terms of bug evolution and amount of code clones. We are thus convinced that we have no strong bias in the results.

V. Conclusion and Future Work

This paper contributed to the analysis of the evolution history of code clones with the goal to find incomplete bug-fixes. A novel approach has been proposed that inspects commit messages for terms indicating a bug-fix in conjunction with unveiling gapped clones from evolution analysis.

We have performed a study on six real-world open source and industrial software systems for evaluating this approach. The results clearly show that inconsistent fixes—although varying in number—are a problem common to many software systems. The proposed toolkit helps revealing this missing bug-fixes in code clones with an acceptable precision of 30% to 60%. Compared to gapped clone detection, which has a precision of 20% to 30%, the evolution analysis produces more precise results. Moreover, bugs are not only reported, but with commit messages and inconsistent clone pairs valuable context information is provided.

The approach is suitable for both first-time analyses of a system which may even be performed by persons not familiar with the system as well as continuous analyses. The latter is supported by the index-based clone detection backend, which supports fast incremental updates.

For future work, the approach can be extended, to gain further precision or performance improvements. We plan to do a combined approach of gapped clone detection and evolution analysis with some kind of weighting of the found incomplete fixes. Also the content of altered source code can be taken into account. Added null-checks, catched exceptions or additional if-clauses are highly suspect to represent missing bug-fixes if applied inconsistently. However, this requires additional research and goes beyond the scope of this paper.

Further performance improvements can be gained by keeping the normalized source code in memory between consecutive iterations and just update modified files. An analogous method is already used for updating the clone index and needs to be applied here as well, since disk operations are one essential bottle neck for large-scale system analyses.

Acknowledgment

The authors would like to thank Munich RE Group and AOL for supporting this study. 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.

References

[1] G. Canfora, L. Cerulo, and M. Di Penta, "Tracking your changes: a language-independent approach," *Software, IEEE*, vol. 26, no. 1, pp. 50–57, 2009.

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

[3] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 485–495.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. on Softw. Eng.*, 2006.

[5] E. C. Lingxiao Jiang, Zhendong Su, "Context-based detection of clone-related bugs," in *Proc. of ESEC/FSE '07*, 2007.

[6] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *Proc. of CSMR '07*, 2007.

[7] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, 2009.

[8] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *Proc. of ICSM '07*, 2007.

[9] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. on Softw. Eng.*, 2004.

[10] 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.

[11] S. Kim, K. Pan, and E. E. Whitehead Jr, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 35–45.

[12] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, pp. 120–130.

[13] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004, pp. 563–572.

[14] T. Zimmermann, N. Nagappan, and A. Zeller, "Predicting bugs from history," *T. Mens, S. Demeyer (Eds.), Software Evolution, Springer*, pp. 69–88, 2008.

[15] S. Bazrafshan, R. Koschke, and N. Gode, "Approximate Code Search in Program Histories," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 109–118.

[16] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 158–167.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 187–196, 2005.

[18] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions on*, vol. 28, no. 7, pp. 654–670, 2002.

[19] N. Göde and M. Rausch, "Clone Evolution Revisited," *Softwaretechnik-Trends*, vol. 30, no. 2, pp. 60–61, 2010.

[20] R. Geiger, B. Fluri, H. Gall, and M. Pinzger, "Relation of code clones and change couplings," *Fundamental Approaches to Software Engineering*, pp. 411–425, 2006.

[21] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Software Maintenance, 1998. Proceedings. International Conference on*, 1998, pp. 190–198.

[22] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010, pp. 1–9.

[23] N. Göde and R. Koschke, "Incremental clone detection," in *Workshop Software-Reengineering (WSR'09)*, 2009, pp. 219–228.

[24] E. Juergens, "Why and How to Control Cloning in Software Artifacts," 2011, Dissertation, Technische Universität München.