Electronic Communications of the EASST
Volume 60 (2013)

Proceedings of the
Seventh International Workshop on
Software Quality and Maintainability
-
Bridging the gap between end user expectations,
vendors' business prospects,
and software engineers' requirements on the ground
(SQM 2013)

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

Martin Poehlmann and Elmar Juergens

19 pages

Guest Editors: Eric Bouwers, Yijun Yu
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

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

Martin Poehlmann[1] **and Elmar Juergens**[2]

[1]poehlmann@cqse.eu
[2]juergens@cqse.eu
CQSE GmbH, Germany
http://www.cqse.eu

Abstract: When a bug is fixed in duplicated code, it is often necessary to modify all duplicates (so-called clones) accordingly. In practice, however, fixes are often incomplete, which causes the bug to remain in one or more of the clones. This paper presents an approach that detects such incomplete bug-fixes in cloned code by analyzing a system's version history to reveal those commits that fix problems. The approach 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 our approach. We identified likely incomplete bug-fixes in all analyzed systems.

Keywords: software clones, software maintenance, static analysis, software evolution

1 Introduction

"I had previously fixed the identical bug [...], but didn't realize that the same error was repeated over here" [CCD09]. This excerpt of a commit message is common among 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 [Kos07]. During maintenance, changes often affect all clones and must therefore be carried out on all instances of these 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 [JDHW09]. 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 discovering errors in clones in practice, often due to incomplete bug-fixes [JDHW09, LLMZ06, LJ07, ACD07, TCAD09, BFG07, YMNC04].

To avoid such incomplete bug-fixes, *clone management* [Kos07] approaches have been proposed to alert developers of the existence of clones when performing changes. However, while such approaches may ease future maintenance, they are of no help for incomplete bug-fixes that occurred in the past. How can we detect such inconsistent bug-fixes that are already part of a system's source code?

One approach to detect incomplete bug-fixes in cloned code is to search for clones that differ from each other in terms of, for example, modified or missing statements. These differences could hint at incomplete bug-fixes. While several approaches are able to detect clones with differences (so called type-3 clones) [RCK09], not every difference between a pair of clones hints at a bug. In many cases, a developer will copy & paste a piece of code and modify it intentionally, since the new copy is required to perform a slightly different function.

Over the past five years, we have inspected clones in numerous industrial and open-source systems and found that most of these systems contain substantial numbers 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 have been introduced intentionally, often during the creation of the clone. The ideal approach would (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. This approach iterates through the revision history of a system and classifies changes as bug-fixes, if the commit message contains specific keywords such as *bug* or *fix*. It then tracks the evolution of code clones between revisions to detect clones that have become inconsistent *as the result of a fix*. Our assumption is that such inconsistencies have a high likelihood of being unintentional. The case study that we conducted for this paper confirms this assumption.

Furthermore, in contrast to clones detected on a single system version, this approach provides information about which change, made by which author and for which defect, caused the difference between the clones. Based on our experience, this information substantially helps developers to determine whether and how to resolve differences between clones.

This work targets the problem of incomplete bug-fixes in cloned code, which causes the bugs to remain in the system. There is currently a lack of approaches that efficiently reveal such incomplete bug-fixes. For this, we have developed 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 implemented the approach based on the incremental clone detection functionality of the open-source program analysis toolkit ConQAT.[1] We evaluated the approach in a case study on six industrial and open-source systems written in Java and C#. The results of the case study show that the approach is feasible and does reveal missing bug-fixes.

2 Related Work

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

[1] http://conqat.org

2.1 Evolution-based

Kim et al. [KPW06] proposed a tool called *BugMem*, which uses a database of bug and fix pairs to find bugs and suggest 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 that hints at a bug-fix. As suggested by Mockus and Votta [MV00] they used the terms ”Bug” or ”Fix” to identify 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 are stored in the database.

We used the same method for scanning the version history for interesting terms, although we refrain from including references to bug tracking reports, since some systems track both bugs and feature requests with such tools. In order for them to be included, further work would be required to distinguish bugs and feature requests, and to make the data available offline.

Zimmermann et al. [ZWDZ04] took a similar approach by mining data from a version control system for a change recommendation system. Their goal was to suggest changes and fixes within the IDE [ZNZ08] in the manner of shopping applications: “Programmers who changed these functions also changed...”. With a precision of 26%, however, the amount of meaningful suggestions is rather low. From a user perspective, our approach differs from Zimmermann’s mainly in that the recommendation tool tries to prevent bugs by suggesting changes upon modification of files in the IDE.

2.2 Clone-based

Juergens et al. [JDHW09] 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 over 150 inconsistent clones for all but one system, 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 involving 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.

2.3 Combined – Clone-evolution-based

APPROX, by Bazrafshan et al. [BKG11], is a tool for searching among arbitrary code fragments in multiple versions and branches for similar fragments in order to find missing fixes. The search is based on code clone detection, but is limited to searching 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 [DR07] 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 of all related clones while editing a file, since it provides automated edit propagation to other clone siblings.

In contrast, Kim et al. [KSNM05] analyzed clone genealogies by combining CCFinder [KKI02] with a clone tracking approach. In a case study, they showed that a maximum of 40% of all clones are changed consistently during system evolution. Canfora et al. [CCD09] used another clone detection tool, as well as other study objects, and reproduced the results from Kim et al. with approximately 43% of all clones being consistently modified. In detail, the inconsistencies sum up to 67%, 14% of which were propagated later to become consistent again.

Göde and Rausch [GR10] analyzed the evolution of three open-source systems over a five-year period. For this, they used an iterative clone detection and tracking algorithm, and showed that 43.1% of all changes to clones are inconsistent, with 16.8% being unintentionally inconsistent. Again, the total amount of reported inconsistencies is quite high, with 131 clones, and includes many 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. [GFGP06] 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 approximately the same time, from the same author, and with the same commit message [GHJ98]. Nevertheless, Geiger et al concluded that a correlation is too complex to be easily expressed and more information is needed to identify harmful clones. Our approach does not correlate change coupling, but instead uses a prediction of which commit is a fix based on its commit message.

3 Revealing missing Bug-Fixes

This section provides an in-depth explanation of how the analysis process of the proposed approach works. As depicted in Figure 1, the process is an iteration over the available revisions of the version control system in order to simulate the source code evolution. Several steps are performed for each iteration 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.

3.1 Get Next Revision

At the start of the analysis, the version control system is queried for all or a subset of available revisions. During the iteration loop, the revisions are checked out in chronological order and metadata containing the commit message is handed over to the next steps.

Git[3] is used as backend for the revision iteration, as it supports import from various other version control systems. Hence, the approach is virtually independent of the version control system which the code is managed with. Moreover, as a distributed version control system,

[2] http://blue-edge.bg/simscan
[3] http://git-scm.com

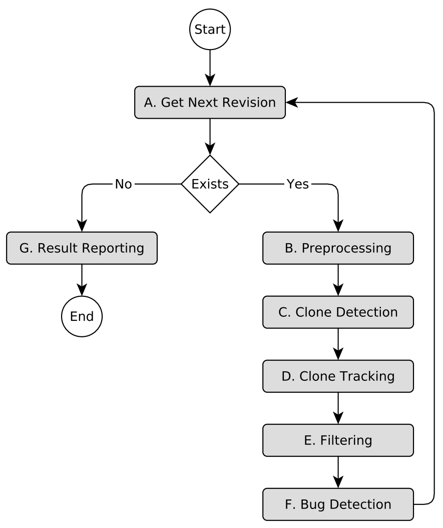


Figure 1: Overview of the iterative bug detection process.

the entire version history is stored locally and does not require communication via a network connection.

3.2 Preprocessing

Firstly, 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, such as package identifiers or include statements, as these are unlikely to contain bugs. Finally, the source code is normalized into a generic representation that is insensitive to method names, variable names and literals.

3.3 Clone Detection

Code clones are detected with an algorithm provided by ConQAT using a hash index [HJHC10], 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 from the index all data, that belongs to these files. These files are later added to the index again, but with updated content, which enables us to retain most of the data and update just a small fraction depending on the changeset size.

The detection is configured to keep clones from crossing method boundaries (shaped detection). This enables us to minimize the amount of semantically non-meaningful code clones. Moreover, we do not take into account gapped clones (type-3) that contain statement additions, removals, and modifications after normalization. Including gapped clones in this step will not enable us to determine whether the inconsistency in such a clone is related to a bug-fix.

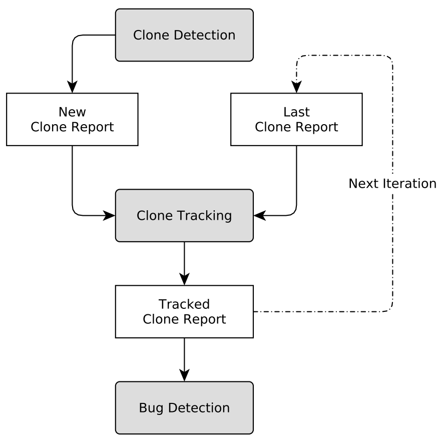


Figure 2: Clone tracking dataflow process.

3.4 Clone Tracking

This step performs the actual evolution analysis for code clones. The reported clones from the clone detection step are mapped to those from the previous iteration cycle (*cf.* Figure 2). Modifications that are performed inconsistently between clones turn a group of ungapped clones 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 [GK09]. It first calculates the edit operations of a code clone between two consecutive revisions and then propagates these edit operations to the clones of the current revision so that the clone positions are updated accordingly. The updated clones are then mapped to those from the current detection step: Firstly, those clones whose positions do not differ are matched. Secondly, a fuzzy coverage matching is performed on the remaining clones; this determines whether one clone covers another and reports clones with modifications and gaps that are of interest for the proposed bug detection approach.

3.5 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. We selected a threshold of 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.

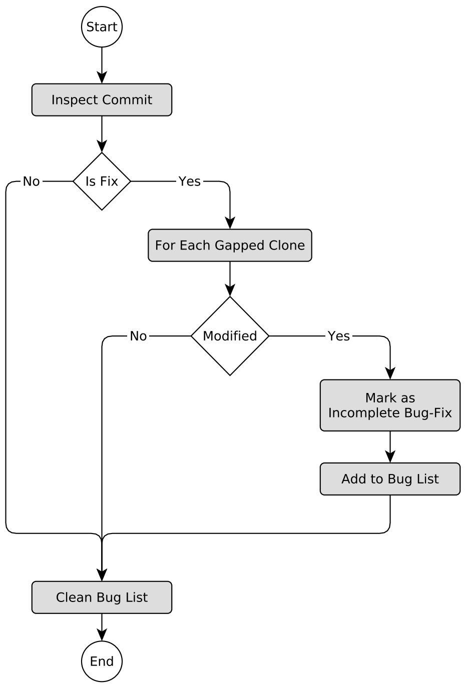


Figure 3: Incomplete bug-fix detection flow.

3.6 Bug Detection

We distinguish between clones that contain an incomplete bug-fix and those that do not, as outlined in Figure 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 that may indicate a bug-fix, such as *fix*, *defect*, or *bug*. If such a term is found, the whole commit and its modifications are seen as bug-fix commit. Mockus and Votta [MV00] proposed an even more extensive approach for finding fixes in commit messages. In our case, a customizable list of system specific terms was sufficient.

The list of code clones is then searched for those clones that were marked as *modified in this revision* in the clone tracking step. Our approach suggests that such clones contain an incomplete fix, as the commit is a bug-fix and the clone was inconsistently modified in this revision. These clones are then 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 either becomes consistent again or vanishes. We continue with a new iteration loop, as long as newer revisions are available.

3.7 Result Reporting

The final step, after the revision iteration terminates, is the result reporting. It writes all incomplete bug-fixes into an 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 the file name, the start and end line, as well as the position of gaps. Information about

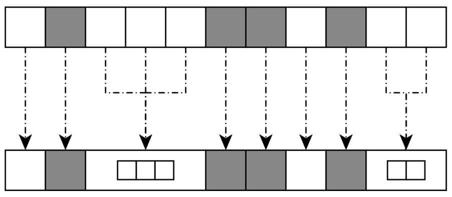


Figure 4: Compressing non-bug-fix revisions (white) into single commits. Revisions including a bug-fix (gray) are not compressed.

the revision that caused the clone to become inconsistent is also stored, namely the commit message and the revision identifier.

3.8 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 approxiumately 25%. We can exploit this for a notable performance improvement, since we only need to inspect the commits that represent a bug-fix. All revisions between bug-fixes can be compressed into a single composite revision, as depicted in Figure 4. These 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 $F \leq R$. We then inspect R revisions without compression. With revision compression a maximum of $2 * F + 1$ 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.

4 Case Study

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

4.1 Research Questions

We investigate the following four research questions:

RQ 1 *How many code clones are affected by fixes?*
If inconsistent changes to clones do not occur in software systems, further analyses do not make sense.

RQ 2 *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, high precision in detecting incomplete bug-fixes is desirable.

RQ 3 *What impact does the commit changeset size have on the bug-finding precision?*
Commits with a large number 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 provide evidence by analyzing the precision with respect to limited changeset sizes.

RQ 4 *How does evolution-based bug detection compare to gapped clone detection on a single revision?*
Gapped clone detection can also be used to find incomplete bug-fixes. This arises the question of whether the overhead of analyzing history information is justified compared to gapped clone detection in terms of precision and the number of reported inconsistently fixed clones needed to be inspected by a developer or quality assurance engineer.

4.2 Study Objects

The case study was performed on six real-world software systems as listed in Table 1. These projects were chosen because they have an evolved version history and also because we needed access to the version control system, even for non-open-source projects. Therfore, we relied on own contacts for industry code. In contrast, *Banshee* and *Spring* are available as open-source systems and are 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. Systems 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 cross-platform audio player *Banshee*[4] written in C# by more than 300 contributors and the Java enterprise application framework *Spring*[5] developed by over 50 contributors. All systems are actively developed and in production use.

4.3 Amount of Incomplete Fixes — RQ 1

Design and Procedure The first research question investigates the amount of code clones that were affected by fixes and consequently became inconsistent. 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. Our bug detection toolkit has been slightly modified to deliver these statistics. The configuration remained as described in Section 3, with a minimal clone length of 7 statements. Additionally, groups of clones with more than three instances were filtered, because precursory studies showed that the tracking approach can be unreliable if modifications occur in more than one instance. Including these clones remains for future work.

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

Table 1: 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

Table 2: Total amount and percentage of commits containing bug-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

To identify bug-fix commits, the version history of all six systems was manually inspected, yielding the following list of keywords hinting at bug-fixes: *Fix, Bug, Defect, Correct*. As Systems A, B, and C are developed by German engineers, some of the commit messages are also written in German, which extended the list of keywords to include the German translations *Fehler*, *Defekt*, *behoben*, and *korrigiert*. Furthermore, the word *correct* has frequently been misspelled by developers as *corect*, so we also took this variant into account. We decided against identifying commits as fixes solely from the presence of a reference to a bug-tracker issue number, because feature requests were managed in the same tracker software for each of the system.

To compare execution times, 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 throughout the case study.

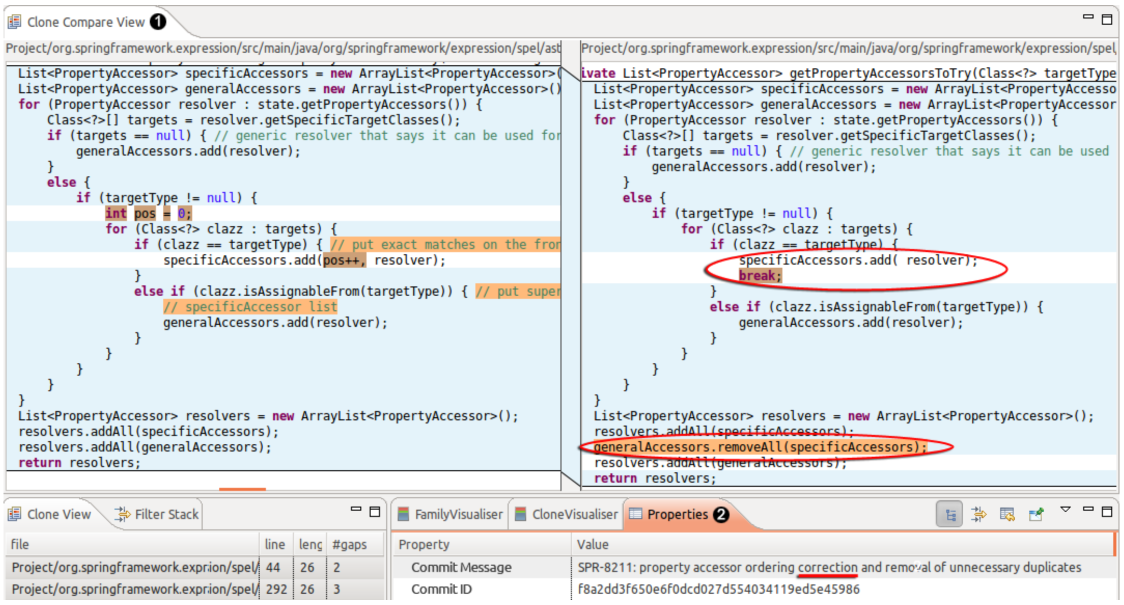
Results Table 2 summarizes the amount of fixes detected with the mentioned keywords in each analyzed system. According to this result, almost one in four commits is a fix.

Table 3 shows the number of incomplete bug-fixes affecting code clones for each system. For all systems, fewer 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-fix and appear multiple times in the above statistic.

Discussion Table 3 compares the low number of incomplete fixes, which are still present in the last revision, with all inconsistencies found during the analysis. For Banshee, it shows that the detection algorithm is not stable with regards to large refactorings. Some detected bugs were lost

Table 3: 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



Project/org.springframework.expression/src/main/java/org/springframework/expression/spel/ast

```
List<PropertyAccessor> specificAccessors = new ArrayList<PropertyAccessor>(
List<PropertyAccessor> generalAccessors = new ArrayList<PropertyAccessor>()
for (PropertyAccessor resolver : state.getPropertyAccessors()) {
    Class<?>[] targets = resolver.getSpecificTargetClasses();
    if (targets == null) { // generic resolver that says it can be used for
        generalAccessors.add(resolver);
    }
    else {
        if (targetType != null) {
            int pos = 0;
            for (Class<?> clazz : targets) {
                if (clazz == targetType) { // put exact matches on the fron
                    specificAccessors.add(pos++, resolver);
                }
                else if (clazz.isAssignableFrom(targetType)) { // put super
                    // specificAccessor list
                    generalAccessors.add(resolver);
                }
            }
        }
    }
}
List<PropertyAccessor> resolvers = new ArrayList<PropertyAccessor>();
resolvers.addAll(specificAccessors);
resolvers.addAll(generalAccessors);
return resolvers;
```

Project/org.springframework.expression/src/main/java/org/springframework/expression/spel

```
ivate List<PropertyAccessor> getPropertyAccessorsToTry(Class<?> targetType
  List<PropertyAccessor> specificAccessors = new ArrayList<PropertyAccesso
  List<PropertyAccessor> generalAccessors = new ArrayList<PropertyAccessor
  for (PropertyAccessor resolver : state.getPropertyAccessors()) {
      Class<?>[] targets = resolver.getSpecificTargetClasses();
      if (targets == null) { // generic resolver that says it can be used
          generalAccessors.add(resolver);
      }
      else {
          if (targetType != null) {
              for (Class<?> clazz : targets) {
                  if (clazz == targetType) {
                      specificAccessors.add( resolver);
                      break;
                  }
                  else if (clazz.isAssignableFrom(targetType)) {
                      generalAccessors.add(resolver);
                  }
              }
          }
      }
  }
  List<PropertyAccessor> resolvers = new ArrayList<PropertyAccessor>();
  resolvers.addAll(specificAccessors);
  generalAccessors.removeAll(specificAccessors);
  resolvers.addAll(generalAccessors);
  return resolvers;
```

file	line	leng	#gaps
Project/org.springframework.exprion/spel/	44	26	2
Project/org.springframework.exprion/spel/	292	26	3

Property	Value
Commit Message	SPR-8211: property accessor ordering correction and removal of unnecessary duplicates
Commit ID	f8a2dd3f650e6f0dcd027d554034119ed5e45986

Figure 5: Screenshot of the ConQAT clone workbench including a clone compare view (❶) and metadata about the commit this inconsistency was introduced by (❷).

during tracking because the Banshee developers partially restructured sub components. Even gathering renamed files from the version control system will not completely eliminate this issue, since code may be exchanged between files as well. System C also suffered from this to 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. Therefore, it might switch between branches for consecutive runs, causing some bugs to disappear, due to tracking issues. In this case, an adapted evolution analysis is needed, that traverses merged branches separately. This requires some major rework for the entire revision iteration and bug detection process, which is an area for future work.

Table 4: 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

4.4 Detection Precision — RQ 2

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

$$precision = \frac{\#\, bug\ candidates}{\#\, inconsistently\ fixed\ clones} \tag{1}$$

The decision whether 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 manually inspecting the commit message and modifications (see Figure 5). In case the manual comparison revealed that the inconsistency seems to be intentionally introduced, the clone is discarded as false positive.

Our goal is to cost-effectively identify missing bug-fixes and, to do this, we are willing to sacrifice recall for higher precision and leave analysis of recall (if feasible at all) for future work.

Results Table 4 summarizes the total amount of code clones detected during history evolution and present in the last revision, as well as the number of inconsistently fixed clones. Table 4 also lists the amount of bug candidates resulting from manual inspection, the precision calculated with Equation 1, and the time the detection took in minutes.

Apart from Spring, the detection algorithm identifies approximately 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 vary depending on the project size and the amount of revisions chosen for evolution analysis. However, 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. The documentation contains very strict guidelines[6] for third party contributions with respect to coding style, unit-testing and patch

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

submission, and similar rules seem to apply for internal work as well.

For system B, many false positives were introduced by big commits that *"fixed"* coding style-related issues. We did not count these as bug-fixes. The same problem also decreases the results for System C. RQ 3 tries to alleviate this problem by ignoring commits with many modified files.

4.5 Changeset Size Impact — RQ 3

Design and Procedure This question analyzes how the size of the commit changeset influences the bug-finding precision. Analogous to RQ 2, 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 2. We also applied a minimum and maximum threshold to the changeset size of a commit, which is evaluated at the time of revision compression. The changeset size is limited with a 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 5 and grouped by the limit interval. The table shows that the precision in intervals $[1,5]$ and $[6,10]$ is almost twice as high than the average precision from Table 4 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 that lowered the results for RQ 2 fall into this category. Nevertheless, the systems that performed worse previously did not catch up to the other systems in terms of precision, although there was a noticeable increase.

The sum of the reported bugs or bug candidates of each system may not be equal to the results from Table 4, since a clone can be altered by fixes of different changeset sizes. Therefore, some clones are listed multiple times.

Discussion By limiting the changeset size to a maximum of 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.

Nonetheless, different development practices and policies such as committing changes just once per day or merging upstream work in a single revision may cause some incomplete bug-fixes to be missed. However, the results in Table 5 show that for the analyzed systems the gain of precision outweighs this loss.

4.6 Gapped Clone Detection Comparison — RQ 4

Design and Procedure This research question compares the evolution-based bug-finding approach to the less time-consuming gapped clone detection. To answer the question, we ran a gapped clone detection with ConQAT on the study objects and filtered the returned clones to contain at least one modification. The parameters from RQ 2 are used again with parameters chosen especially for the gapped clone detection according to Juergens [Jue11]; namely, the gap ratio must be at most 20% and the edit distance can have a maximum of 5 edits.

Table 5: 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	—	—

Finally, we determined the precision of finding bugs in this set of clones with Equation 1 and compared the results to those of RQ 2 and RQ 3. Due to the large amount of reported inconsistently fixed clones for some systems, we only inspected some of the reported clones for bug candidates, which were randomly chosen from the entire result set.

Results Table 6 presents the results for the gapped clone detection executed for each of the study objects. Compared to the results from the evolution-based approach, the overall precision is more homogeneous, ranging from 20% to 30%. Hence, there is no significant difference to

Table 6: Results for finding bugs with gapped clone detection.

System	Reported Clones	Inspected (%)	Bug Candidates	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

the evolution analysis unless changeset sizes are taken into account. Nevertheless, the detection reported 2 to 6 times more clones that we had to inspect manually. As a positive side effect, more bug candidates were detected. Compared to the enhanced approach with limits applied to the changeset size, the gapped detection clearly performs worse in terms of precision.

Discussion The results of the evolution-based approach were gained via a time-consuming analysis that took over two hours for some systems. Therefore, it is valid to ask whether a simple gapped clone detection, which only takes two minutes to execute, can identify inconsistent clones with similar precision.

Based solely on 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 many 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, compared with limited changeset sizes, the precision is clearly higher than for gapped clone detection. Therfore, we consider the long execution times to be justified.

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

Although the gapped detection did unveil some bug candidates, that we had not encountered before, the evolution-based approach also reported some bug candidates not found by the gapped approach. Therefore, a combination of both methods could be beneficial.

4.7 Threats to Validity

This section provides an overview of the internal and external threats to the 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 whether an inconsistently fixed clone is a bug candidate, since the researcher has no in-depth knowledge of how the analyzed systems are built and how the components work together. We inspected the results twice in an attempt to mitigate the threat and reached the same conclusion both times. For future work, we would also like to verify the results by developers.

We did not take recall into account when answering the research questions. Nevertheless, this does not present a problem with regard 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 few false positives as possible in order 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 in finding bugs paid off.

Without knowing the recall, however, statistical tests of the evaluation results have little use, especially since we minimized the set of inconsistent clones to gain higher precision. Hence, we refrained from backing up RQ 3 and RQ 4 with statistical tests.

Another group of threats concerns the program evolution. Depending on the development practices and version control system used, fixes that happen on 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 do not detect clones that were newly created and had a fix applied to the code before re-committing to the version control system. These problems are more or less technical restrictions that cannot be prevented, which means that 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; that is, code regions that are syntactically similar to each other but contain no semantic similarity. Examples include 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. In doing so, however, 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. Also, it is easy to add new keywords for other systems.

External Validity The systems chosen for the case study as study objects may not represent an average software system. For closed-source systems, however, we are limited to existing industry contacts. Nonetheless, all systems are developed by different teams and for different purposes, as described in Subsection 4.2. Moreover, RQ 2 showed that they also have different characteristics in terms of bug evolution and amount of code clones, so we are convinced that we have no strong bias in the results.

5 Conclusion and Future Work

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

To evaluate our approach, we performed a study on six real-world open-source and industrial software systems. The results clearly show that inconsistent fixes—although varying in number—are a problem common to many software systems. The proposed toolkit helps to reveal missing bug-fixes in code clones with an acceptable precision of 30% to 60%. The evolution analysis produces more precise results than gapped clone detection, which has a precision of 20% to 30%. Not only are bugs reported, but commit messages and inconsistent clone pairs provide valuable context information.

The approach is suitable for first-time analyses of a system, which may even be performed by persons not familiar with the system, and for 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 create a combined approach of gapped clone detection and evolution analysis with some kind of weighting of the incomplete fixes we identified. The content of altered source code can also be taken into account. Added null-checks, caught exceptions, or additional if-clauses are highly suspect to represent missing bug-fixes if applied inconsistently. However, such work requires additional research that is beyond the scope of the present paper.

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

Acknowledgements: 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". Responsibility for this article lies with the authors.

Bibliography

[ACD07] L. Aversano, L. Cerulo, M. Di Penta. How Clones are Maintained: An Empirical Study. In *Proc. of CSMR '07*. 2007.

[BFG07] T. Bakota, R. Ferenc, T. Gyimothy. Clone Smells in Software Evolution. In *Proc. of ICSM '07*. 2007.

[BKG11] S. Bazrafshan, R. Koschke, N. Gode. Approximate Code Search in Program Histories. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*. Pp. 109–118. 2011.

[CCD09] G. Canfora, L. Cerulo, M. Di Penta. Tracking your changes: a language-independent approach. *Software, IEEE* 26(1):50–57, 2009.

[DR07] E. Duala-Ekoko, M. P. Robillard. Tracking code clones in evolving software. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. Pp. 158–167. 2007.

[GFGP06] R. Geiger, B. Fluri, H. Gall, M. Pinzger. Relation of code clones and change couplings. *Fundamental Approaches to Software Engineering*, pp. 411–425, 2006.

[GHJ98] H. Gall, K. Hajek, M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings. International Conference on*. Pp. 190–198. 1998.

[GK09] N. Göde, R. Koschke. Incremental clone detection. In *Workshop Software-Reengineering (WSR'09)*. Pp. 219–228. 2009.

[GR10] N. Göde, M. Rausch. Clone Evolution Revisited. *Softwaretechnik-Trends* 30(2):60–61, 2010.

[HJHC10] B. Hummel, E. Juergens, L. Heinemann, M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. Pp. 1–9. 2010.

[JDHW09] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*. Pp. 485–495. 2009.

[Jue11] E. Juergens. Why and How to Control Cloning in Software Artifacts. 2011. Dissertation, Technische Universität München.

[KKI02] T. Kamiya, S. Kusumoto, K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on* 28(7):654–670, 2002.

[Kos07] R. Koschke. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.

[KPW06] S. Kim, K. Pan, E. E. Whitehead Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. Pp. 35–45. 2006.

[KSNM05] M. Kim, V. Sazawal, D. Notkin, G. Murphy. An empirical study of code clone genealogies. *ACM SIGSOFT Software Engineering Notes* 30(5):187–196, 2005.

[LJ07] E. C. Lingxiao Jiang, Zhendong Su. Context-Based Detection of Clone-Related Bugs. In *Proc. of ESEC/FSE '07*. 2007.

[LLMZ06] Z. Li, S. Lu, S. Myagmar, Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. on Softw. Eng.*, 2006. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2006.28

[MV00] A. Mockus, L. G. Votta. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on*. Pp. 120–130. 2000.

[RCK09] C. Roy, J. Cordy, R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009.

[TCAD09] S. Thummalapenta, L. Cerulo, L. Aversano, M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 2009.

[YMNC04] A. Ying, G. Murphy, R. Ng, M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. on Softw. Eng.*, 2004.

[ZNZ08] T. Zimmermann, N. Nagappan, A. Zeller. Predicting bugs from history. *T. Mens, S. Demeyer (Eds.), Software Evolution, Springer*, pp. 69–88, 2008.

[ZWDZ04] T. Zimmermann, P. Weibgerber, S. Diehl, A. Zeller. Mining version histories to guide software changes. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. Pp. 563–572. 2004.