An Empirical Study on Inconsistent Changes to Code Clones at Release Level

Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, Ahmed E. Hassan
Queen's University
Kingston, Ontario, Canada
Email: {nicbet, swy, walid, bram, ahmed}@cs.queensu.ca, ying.zou@queensu.ca

***Abstract*—Current research on code clones tries to address the question whether or not code clones are harmful for the quality of software. As most of these studies are based on the fine-grained analysis of inconsistent changes at the revision level, they capture much of the chaotic and experimental nature inherent to any ongoing software development process. Conclusions drawn from the inspection of highly fluctuating and short-lived clones are likely to exaggerate the ill effects of inconsistent changes. To gain a broader perspective, we perform an empirical study on the effect of inconsistent changes on software quality at the release level. Based on a case study on two open source software systems, we observe that only 1% to 3% of inconsistent changes to clones introduce software defects, as opposed to substantially higher percentages reported by other studies. Our findings suggest that developers are able to effectively manage and control the evolution of cloned code at the release level.**

***Keywords*-Software Engineering; Maintenance management; Reuse models; Scalability, Maintainability**

I. Introduction

A code clone is a part of the source code that is identical, or at least highly similar, to another part (clone) in terms of structure and semantics. Related clones belong to the same clone group, and the evolution of a particular clone group over the course of multiple versions of a software system forms a clone genealogy. In the past, cloning was generally considered to be harmful [1]–[7], due to the belief that changes to one clone need to be propagated to all related clones, thus increasing both the maintenance effort and the likelihood of introducing defects. Recently, a series of studies established cloning as a reasonable software engineering method for common software development problems [8]–[11]. It is not clear yet which of these two visions prevails, or whether the right vision depends on the software system at hand.

The on-going debate about the harmfulness of cloning has sparked many empirical studies on the trade-offs of cloning. Typically, inconsistent changes of clones are analyzed in a single snapshot of the software system, or between very small development increments (e.g., revisions) [5], [11]–[15]. Thus far, all studies on code cloning seem to focus on the impact of cloning on developers, such as the developers' ability to consistently update all clones or to understand the full extent of a clone group.

Although fine-grained studies are indispensable, we argue that they paint a too negative image of the world. The primary stakeholder in software development is not the developer, nor the architect or tester, but the end user, who installs an "official software release", then runs the software. Since many code clones are only temporary phenomena used for experimentation [11] and seem to disappear over time [15], we hypothesize that the end user faces far less (inconsistently changed) code clones than developers do, and hence experiences far less ill-effects caused by cloning. In other words: code clones are not harmful to end-users.

In this paper, we present an empirical study on inconsistent changes to code clones in two large open source software systems, at the level of releases. We identify six challenges inherent to studying clones at such a coarse-grained level, and address three research questions:

Q1) What are the characteristics of long-lived clone genealogies at the release level?

On average, a clone group survives 6.79 releases and contains 2.34 clones.

Q2) What is the effect of inconsistent changes to code clones when measured at the release level?

The percentage of inconsistent changes that lead to software defects ranges between 1.26% (6 defects out of 462 inconsistent changes) and 3.23% (2 defects out of 62 inconsistent changes).

Q3) What types of cloning patterns are observed at release level?

The *replicate and specialize*, *API* and *language idioms* are the most frequent patterns of cloning.

The rest of the paper is organized as follows: Section II situates our work in the context of prior clone detection research. We present and motivate our three research questions in Section III, then present the design of our case study in Section IV. Section V reports the results of our case study, after which we summarize the challenges involved in release-level clone detection and tracking (Section VI). Finally, Section VII discusses threats to validity and Section VIII presents the conclusion of this paper.

II. Related Work

Recently, multiple excellent surveys on code clones and detection tools have been published, in particular [16]–[18]. We focus on the most closely related work in characterizing the potential threats and virtues of code clones, and in analyzing the evolution of clones over time.

Kapser et al. [9] distill eleven patterns of code clone usage, all of which have both positive and negative consequences on software quality. These patterns show that cloning is often used in practice as a principled engineering method. However, many other works [1]–[4] argue about the negative impact of code clones on software quality. Recently, for example, Lozano et al. [7] show that methods with clones require more maintenance effort than methods without. This effort increases with the number of code clones. Juergens et al. [6] find, after manual inspection of clones in four industrial and one open source system, that inconsistent changes to clones are very frequent and can lead to significant numbers of faults in software.

Johnson [19] was the first to study the evolution of clones (between two versions of the GCC compiler). Over the years, similar studies have been performed on longer-lived and larger systems, such as a large telecommunication system [15] and the Linux kernel [20]. The former study surprisingly finds that a significant number of clones disappear automatically from a system over time, and that most clones are never changed after their creation: Programmers seem to be aware of the clones in a system. Geiger et al. [5] conjecture that the more clones are shared between files, the more these files are changed together, which would suggest that clones are consistently updated. No statistically significant results could be obtained, however.

Kim et al. [11] were the first to map clones in different versions of the source code to each other, in order to study how clone groups evolve over time (genealogy), and more in particular to analyze inconsistent changes inside genealogies. They report that up to 72% of the clone groups in the investigated system disappear within 8 commits in the source code repository. Up to 38% of the clone groups are always changed consistently. Aversano et al. [12] extend these findings, and report that most of the cloned code is consistently maintained in the same commit or shortly after. In contrast, Krinke [14] reports that in the systems he analyzed half of the changes to code clone groups are inconsistent and that corrective changes following inconsistent changes are rare. Finally, Bakota et al. [13] present a technique to map clones in different versions of the source code based on machine learning techniques.

This paper studies code clones at the release level to investigate the effect of inconsistent changes on software quality, in particular on the number of bugs in the clones. This has previously only been done at the revision level [6], [11]–[14], [21], as up until now studies at the release level focused on the evolution of clones instead of on (the impact of) inconsistent changes [5], [8], [15], [19], [20]. Looking at the release level enables us to factor out the effect of temporary code clones on software quality.

III. Research Questions

Contrary to previous studies, we focus on the impact of inconsistent changes on code clones at the release level. For this study, we formulate the following three research questions:

Q1) What are the characteristics of long-lived clone genealogies at the release level?

Previous research has shown that many clones are short-lived [11] or "automatically" disappear over time [15]. Hence, we are interested in the quantitative characteristics of clone genealogies at the release level, i.e., the number of clone groups, their average size and their average lifetime. Such characteristics play an important role for the management of these genealogies.

Q2) What is the effect of inconsistent changes to code clones when measured at the release level?

Measuring the effect of inconsistent changes at a fine-grained level, such as the revision level, might overestimate the real impact of inconsistent changes, as many clones are only short-lived [9], [11], [12]. Studying the effect of inconsistent changes at the release level filters out clones due to experimentation and refactoring, and allows focusing on the nature and impact of long-lived, inconsistent changes.

Q3) What types of cloning patterns are observed at the release level?

If short-lived clones, as identified in previous research [11], are experimental by nature, can we find specific stereotypes of clones that survive multiple releases? Since different types of clones have different consequences for the maintenance effort and overall quality of a software system, it is important to study which kinds of clones prevail at the release level.

IV. Study Design

This section presents the design of our case study to analyze inconsistent changes at the release level. An overview of our approach is shown in Figure 1. We first download all considered releases of the two open source software systems that we studied. For each individual release, we use a clone detection tool to identify cloned source code parts. We then transform the identified code clones into an abstract representation that allows us to track code clones between releases. For each resulting clone genealogy, we identify all changes that were not consistently propagated to all cloned

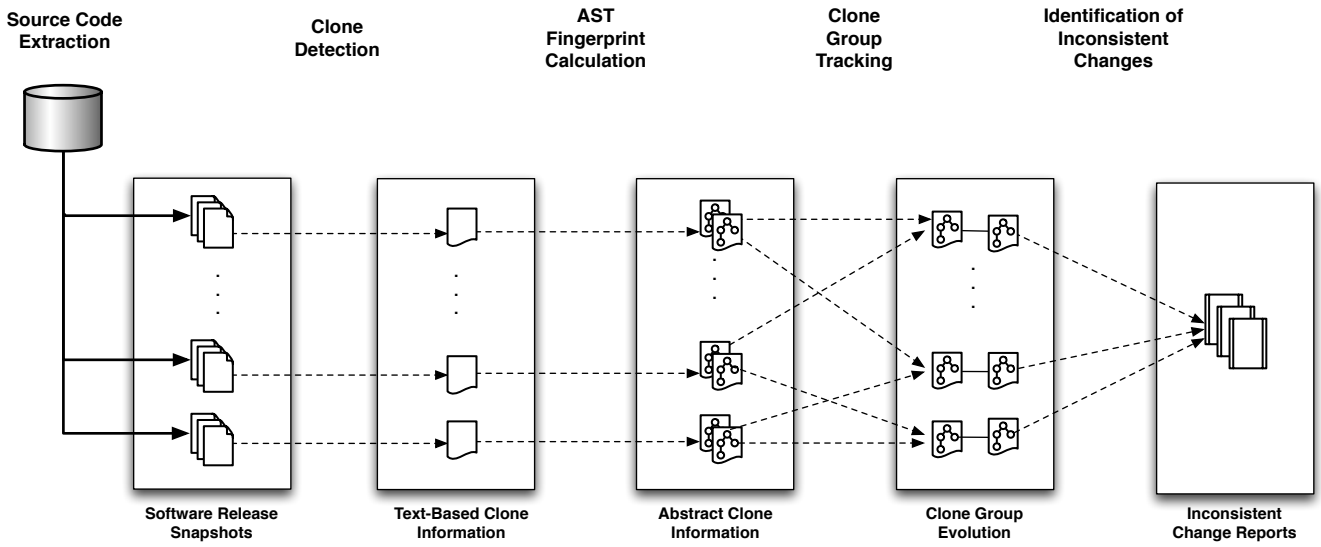

Figure 1. Overview of our approach to study inconsistent changes of code clones at release level.

parts and manually inspect them to investigate whether they introduced errors into the software. Additionally, we perform a manual classification of clone genealogies into eleven categories [9]. This section elaborates on each of these steps.

A. Choice of Subject Systems

We chose two open source software systems of different size and application domain as subjects for our case study:

- `Apache Mina`[1] is a network application framework, designed for easier development of network applications in `Java`. We chose `Apache Mina`, as due to the similarity between network protocols, the system is likely to contain much duplicated code.
- `JEdit`[2] is a popular graphical text editor that has been used as a case study subject in related work [7], [8].

We chose projects from differing contexts to help counter a potential bias of our case study towards any specific kind of software system. For each project, we capture all minor and major releases during a certain period of time. For `Apache Mina` we extract 22 releases between October 2006 and October 2007. For `jEdit` we extract 50 releases between December 2000 and May 2004. A more detailed overview of our subject systems is presented in Table I. The average length of a release cycle is 51 days for `Apache Mina` and 36 days for `jEdit`.

B. Clone Detection within Releases

There are four conventional approaches for clone detection: `text-based`, `token-based`, `syntax-based` and `metrics-based`. Based on existing surveys of techniques [16], [18], [22], [23] and previous studies on the evolution of code clones [12], [13], we opted for a `syntax-based` clone detection tool [2]. This kind of tool identifies clones that form syntactic units by identifying similar subtrees in the abstract syntax trees (AST) of a program. `Syntax-based` clone detection techniques can identify exact clones (with or without differences in comments) and clones with small differences in expression, type or identifier nodes. Some techniques are able to identify clones with larger differences, such as swapped lines or unrelated statements inside the clones. In general, the precision of the clone detection results is high, but at the expense of a lower recall.

We used the SimScan `syntax-based` clone detection tool for our case studies as it is a clone detection tool for Java systems that has successfully been used before by other researchers [8], [12]. SimScan is based on the `ANTLR` parser generator framework and is robust against reformatting, renaming of identifiers and type names, insertions and deletions of additional code in clones, and reordering of statements and declarations.

Table I
OVERVIEW OF THE OPEN SOURCE CASE STUDY SUBJECTS.

	Apache Mina	**jEdit**
#Releases	22	50
Start release	0.8.3	3.0.4
Start date	Oct. 01, 2006	Dec. 03, 2000
End release	1.1.3	4.2.13
End date	Oct. 02, 2007	May 14, 2004
Lines of code	11,908–15,463	47,500–88,305
Shortest release cycle	15 days	1 day
Longest release cycle	96 days	69 days
Average release cycle	51 days	36 days

[1] `http://mina.apache.org/`, last checked June 2009
[2] `http://http://www.jEdit.org/`, last checked June 2009

As is typical for `syntax-based` clone detection tools, it shows minor deficiencies in scalability and recall but it yields satisfactory results for the purpose of our study. The output generated by the tool is extremely detailed and easy to parse.

C. Clone Tracking between Releases

To study the evolutions of clones in a clone group, we need to track clone groups across different versions. A clone genealogy [11] for a particular clone group and version of the source code is a directed acyclic graph which connects the clone group with all corresponding clone groups in the next studied version of the source code, and recursively connects those clone groups to the corresponding ones in the following version.

Various techniques have been used before to map clone groups in different source code versions to each other [8], [11], [13], [14]. We chose to use the recent *clone region descriptors* (CRD) technique [8], which shows promising results for the tracking of code clones in evolving software. A clone region descriptor is a lightweight, abstract representation of a clone region in an AST that combines syntactic, structural and lexical information (Figure 2). A CRD locates a code clone region based on its location in the abstract syntax tree, e.g., "the for loop inside method `a()` of class `B` in the default package". To discriminate between similar code entities in the same containing entity (e.g., two for-loops in the same method), so-called corroboration metrics are used to distinguish the conflicting code entities.

While traversing the abstract syntax tree of a class, we record all entities on the path from the root of the AST to the largest child node that contains a code clone region. The recorded information contains the type of the node (e.g., *method declaration* or *finally* region of a try-catch block) and contextual information about the node (e.g., the method's signature or the caught `Exception`). This extended path information forms the CRD for a single clone region.

To track clone groups over two different versions `A` and `B`, we compare every clone group in version `A` to every clone group in version `B`. If any CRD of a clone in a clone group i in version `A` matches to the CRD of a clone in a clone group j in version `B`, we know that the clone in i and the clone in j are the same. Due to the transitivity of the equals relation we can then infer that clone group i is related to clone group j.

Finally, we compute if any code region corresponding to a clone was changed between two consecutive versions. For this purpose, we analyze the source code of the corresponding clone regions and do a textual comparison. If a change is detected, we generate a marked-up report of the textual differences between the two versions of the clone, suitable for human interpretation.

D. Manual Inspection of Inconsistent Changes

We perform a manual inspection of each inconsistent change to a clone genealogy. Three authors of this paper carried out this inspection independently and compared the results, in order to account for possible human error. For each inspected inconsistent change, we evaluate whether or not the change should have been applied to all parts of the source code clone, i.e., whether the change introduced a bug into the software system. For this, we track for every inconsistent change if a change with similar semantics was applied to another part of the same clone group at a later point in time. If this was the case, we consult the commit messages of the version archives, as well as the projects' bug tracking repositories to check if the observed changes resulted in software defects.

E. Classification of Clone Genealogies

Cloning of source code can occur due to different reasons. Kapser et al. identified eleven types of cloning, each having distinct patterns, purpose, as well as short and long term related management issues [9]. In order to better understand the clone genealogies of each project and to assess the overall risks, we have three authors of this paper act as human oracles, who independently perform a manual inspection of the source code and classify each clone genealogy discovered into one of eleven categories.

We use Fleiss' Kappa – a statistical method that measures the agreement of multiple raters on a categorization of several observations into various classes – to evaluate our consensus on the classification.

V. Case Study Results

This section presents the findings of our case study with respect to our three research questions. After a quantitative analysis of the code clones and code clone groups in the two studied software systems (Q1), we present our findings on the impact of inconsistent changes to clone genealogies (Q2). To better understand the observations of our case study, we conduct a classification of clone groups (Q3).

Q1) What are the characteristics of long-lived clone genealogies at the release level?

We detect a total of 1,387 groups of code clones in 22 releases of `Apache Mina` and a total of 11,160 groups of code clones in 50 releases of `jEdit`. The generation of clone genealogies via clone group tracking reduces the set of 1,387 unrelated groups of code clones to 306 clone genealogies for `Apache Mina`, and the set of 11,160 unrelated groups to 818 genealogies for `jEdit`.

For these genealogies, we then measure the average lifetime and size. Our findings are presented as beanplots in Figure 3. Beanplots are a boxplot alternative for summarizing and comparing the distribution of different data sets [24]. The beanplot shows the individual observations of our two

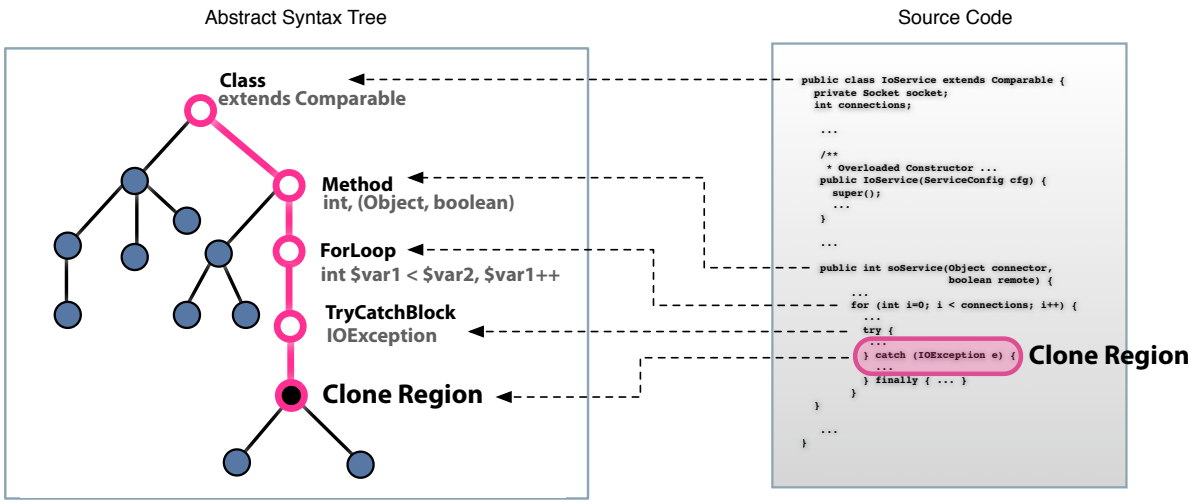

Figure 2. CRD generation using abstract syntax trees.

systems in the form of two asymmetrical, one-dimensional density plots and additionally displays estimated densities (white lines) as well as the averages of each set (black lines) and for both sets combined (dotted lines).

For `Apache Mina`, 79.7% (244 out of 306) of clone genealogies have a lifetime that spans multiple releases. We measure the average lifetime of a genealogy in `Mina` to be 4.59 releases and the average size of a genealogy are 2.56 clones. The smallest clone genealogy observed consisted of a single clone (as the cloned parts were within the smallest syntactical unit observable by our approach), and the largest one consisted of 14 clones.

For `jEdit`, 91.2% (746 out of 818) of clone genealogies have a lifetime that spans multiple releases. The average lifetime of a genealogy in `jEdit` is 9.00 releases and the average size of a genealogy are 2.13 clones. The largest clone genealogy in `jEdit` contains 166 cloned parts and is created by the BeanShell parser class (`org.gjt.sp.jedit.bsh.Parser`), which contains large amounts of automatically generated parsing code.

Overall, we find the average lifetime of clone genealogies across releases to be 6.79 releases and the average size to consist of 2.34 cloned parts.

Q2) What is the effect of inconsistent changes to code clones when measured at the release level?

To study inconsistent changes we need to look at the evolution of the clone genealogies obtained by the previous step. As we aim to study the relation between inconsistent changes and bugs, we distinguish between *reformatting changes* such as code beautification and *syntactical changes* that modify the actual source code.

For `Apache Mina` we record a total of 679 inconsistent changes. Of these, 6 changes were flagged as inconsistent reformatting changes. For `jEdit`, we record a total of 85 inconsistent changes, of which 10 were flagged as inconsistent reformatting changes. We then discard inconsistent reformatting changes and perform a detailed manual inspection of the remaining 748 inconsistent changes.

During the manual inspection of the inconsistent changes, we found that a number of clone groups generated by `SimScan` are false positives. Such false positive clone groups have clones with very similar syntactical structure, but are otherwise unrelated. In order to keep our study sound, we decided to ignore clone genealogies generated by these false positive groups and all the inconsistent changes they account for, respectively. For `Apache Mina`, we found 2 false positive clone groups that account for a total of 13 changes. For `jEdit`, we found 74 false positive clone groups that account for a total of 277 changes.

In the remaining total of 458 inconsistent changes (`Apache Mina` and `jEdit` together), we found seven inconsistent changes that led to software defects: two in `Apache Mina` and five in `jEdit`. We describe these changes in the following:

- **[Mina]** A bug fix (#JIRA-186) was applied between release 0.9.2 and 0.9.3 to the `putString()` method of the `ByteBuffer` class, in order to fix a defect that caused abnormal program termination when an UTF-8 formatted string was used as input for a buffer. However, this change was not reflected in the `getString()` method. Developers fixed this problem by applying a similar bug fix to the `getString()` method between version 0.9.3 and 0.9.4.
- **[Mina]** Developers introduced a call to `fireExceptionCaught()` in the `doFlush()` method of the `SocketIOProcessor` class between version 0.9.5 and 1.0. This call would

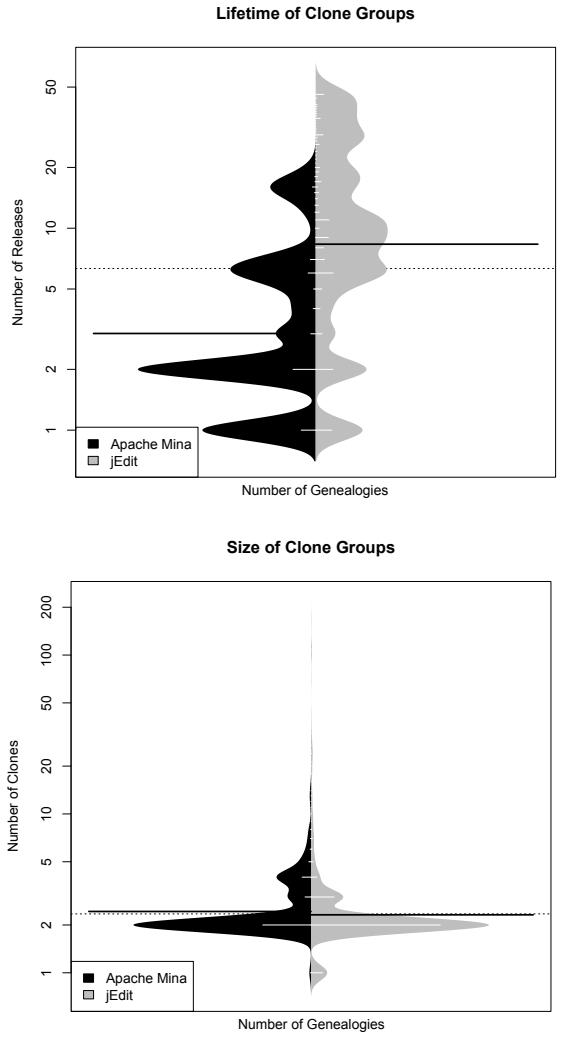

Figure 3. Distribution of clone group lifetime (in # of releases) and clone group size (in # of clones) for `Apache Mina` and `jEdit`.

notify event listeners that an error was found during execution and that the method could successfully handle it (#JIRA-273, #JIRA-283). However, this new behavior of `doFlush()` was not reflected in the rest of the clone group. This was fixed in a later update between 1.0.0 and 1.0.1.

- **[jEdit]** The coordinates to display a user interface element are changed for the `getToolTipLocation()` method of the `BrowserView` class between 4.0-pre3 and 4.0-pre4. However, this change is not applied to the cloned `getToolTipLocation()` methods of other classes in this clone group. This was fixed later between 4.0-pre4 and 4.0-pre5.
- **[jEdit]** An audible beep event was removed from the `goToNextFold()` method of the `EditTextArea` class between 4.0-pre3 and 4.0-pre4. However, removing the beep from the cloned method `goToPrevFold()` in the same class was missed. This was fixed later between 4.0-pre5 and 4.0-pre6.
- **[jEdit]** The `EnhancedMenuItem` class gets an extra shortcut for Mac OSX between 4.0-pre9 and 4.1-pre1, but this shortcut is not introduced to the cloned classes `MarkersMenuItem` and `EnhancedCheckBoxMenuItem`. A later bug fix between 4.1-pre11 and 4.2-pre1 corrects this.
- **[jEdit]** A bug fix to hide the welcome screen when `jEdit` was started with the `-nosettings` switch was applied to the `newView()` method of the main class between 4.0-pre3 and 4.0-pre4. However, the developers missed to apply the fix to another overloaded version of the same method. This was corrected between 4.0-pre5 and 4.0-pre6.
- **[jEdit]** A request for focus in the constructor of the `EditAbbrevDialog` class was removed between 4.0-pre3 and 4.0-pre4, but was missed to be removed from the cloned class `VFSFileChooserDialog`. This was fixed in a later patch between 4.2-pre2 and 4.2-pre3.

For each of the seven defects encountered, we measured the time between the introduction of defects by an inconsistent change to a clone genealogy and the change that fixes them. The results are presented in Table II. For `Apache Mina`, all defects were fixed very quickly with no intermediate defective releases. For `jEdit`, three defects were fixed quickly within on intermediate defective release, and two defects were present in the software over a very long period of time.

Overall, we find that only 1.26% of inconsistent changes in `jEdit` and 3.23% of inconsistent changes in `Apache Mina` led to the introduction of a bug into the software. This contrasts the findings of studies performed at fine-grained levels such as commit-level, which report a substantially higher fraction of bug introducing inconsistent changes [6], [12], [14].

In addition to the software defects described above, we encountered two instances of what we believe to be undetected bugs introduced by inconsistent changes. For both occasions we were not able to manually relate any bug report to the source code region affected. However, further inquiry

Table II
TIME BETWEEN THE INTRODUCTION OF A DEFECT AND ITS FIX.

Project	Defective Release	Fixed Release	#Releases	#Days
Mina	0.9.3	0.9.4	0	30
Mina	1.0.0	1.0.1	0	61
jEdit	4.0.4	4.0.5	0	13
jEdit	4.0.4	4.0.6	1	27
jEdit	4.1.1	4.2.1	10	321
jEdit	4.0.4	4.0.6	1	27
jEdit	4.0.4	4.2.3	19	530

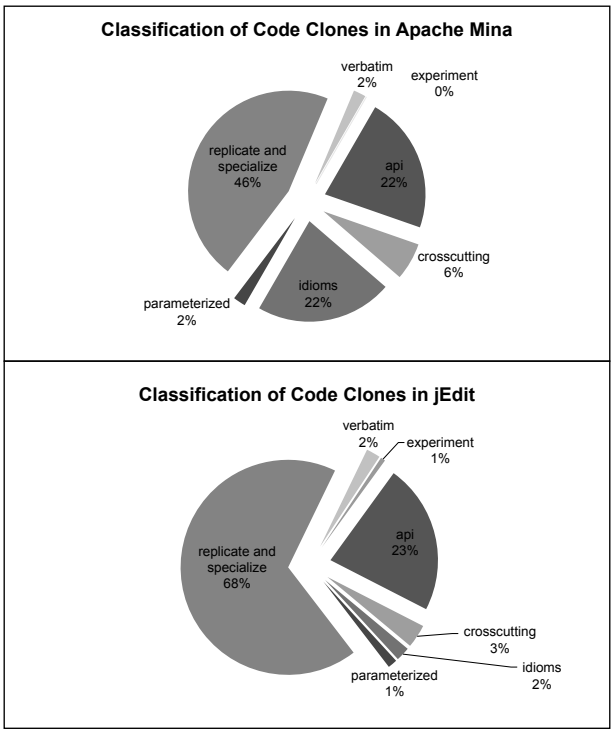

Figure 4. Classifications of clone groups in `Apache Mina` and `jEdit` show a high amount of replicate and specialize clones.

was impossible, as the suspicious parts disappeared in later versions due to refactoring, without being noticed in the meanwhile.

The observations from our manual inspection indicate that a substantial amount of clones (41% for `Apache Mina` and 64% for `jEdit`) in long-lived clone genealogies evolve independently. However, even changes to independently evolving clones occasionally need to be carried out consistently. These changes exhibit a high risk of introducing a bug into the system.

Q3) What types of cloning patterns do we observe at the release level?

In order to understand the dominant types of cloning patterns that we can observe from long-lived clone genealogies at release level, we performed a classification of the encountered clone genealogies into different categories of cloning [9]. For the three judges and eleven categories, we measured an inter-rater agreement of $\kappa = 0.271$ at $p < 0.001$. This result shows a statistically significant and fair level of agreement, considering the low number of judges and high number of categories [25]. While discussing our ratings, we found that most disagreements rooted in subtle semantics of the source code, which blurred the borders between categories. Kapser et al. observed a similar problem in an experiment that showed the difficulty of defining and classifying code clones [26].

The results of our classification of clone genealogies are presented in Figure 4. For both systems, the majority (46% for `Apache Mina` and 68% for `jEdit`) of long-lived code clones were found to belong to the *replicate and specialize* cloning pattern. In this form of cloning, existing code with similar functionality is copied and customized to implement a new functionality of the software. We found that changes to these types of clones in both systems are usually carried out inconsistently because the cloned parts evolve independently.

The second largest pattern of cloning we found in both systems is the *API* cloning pattern, which describes the cloning of a series of program steps, pre-determined by the usage of a specific interface.

A special kind of cloning forms the third largest class of code clones in `Apache Mina`: cloning due to *language idioms*. `Mina` as a network API that makes heavy use of `Java` exception handling, iterators and data structures, which involve tedious, yet frequently needed code fragments.

The remaining clone types observed in both projects were *verbatim* copy and pasting, cloning to implement *crosscutting concerns* and *parameterized cloning*. We found only one instance of cloning due to *experimentation* in `jEdit` when developers introduced different parameter types during the transition of version 3.x to 4.x. This confirms our belief that studying clones at the release-level filters out temporary clones, which are done for experimentation.

Of the seven inconsistent changes that introduced software defects, we found three bug introducing inconsistent changes to clone groups of class *replicate and specialize*, two bug introducing inconsistent changes to clone groups of class *API*, one bug introducing inconsistent change to a clone group of class *experimentation* and one bug introducing inconsistent change to a clone group of class *verbatim*.

In all cases, our observations confirm the risks and long-term issues identified by Kapser et al. [9] for the types of cloning we encountered. Overall, our findings show a dominance of *replicate and specialize* cloning among both projects. These clones typically evolve independently and exhibit a risk of errors introduced by reduced awareness of their presence in the source code. As we could not observe a high fraction of errors introduced to these clone groups through inconsistent changes, we believe that the developers of both projects are aware of these long-lived clones in their software systems and are able to effectively manage their independent evolution.

VI. Challenges of Inter-Release Clone Detection

Several researchers have studied code clones in different software releases [5], [8], [15], [19], [20] or between evenly-spread source control commits [13], [14]. Although none of these papers elaborate on the challenges encountered, our case studies taught us that performing clone detection

between fragmented releases or versions is not straightforward. Harder et al. [21], for example, point out that too long intervals between analyzed code versions hide too many details. This section reports all challenges with clone detection and tracking at release level we had to address in our case study, only some of which have been reported before. For each challenge, we briefly discuss its rationale, consequences for clone detection, and how we addressed the challenge.

C1. Discrete Change Information

Rationale: A series of contiguous changes committed to a source code repository provides a continuous view on the evolution of a software system. As soon as the repository information is sampled by leaving out particular versions or only focusing on milestone versions (releases), information is lost about some individual refactorings and added features. The lower the sampling rate, the longer the intervals between selected versions, the more information is lost.

Consequence: Discrete changes might hide important detailed patterns, i.e. the accuracy of the analysis degrades. For example, a particular clone group could have been left in an inconsistent state for months before the inconsistency eventually is fixed, but if the inconsistent period lies completely between two successive versions, it will not be discovered. It is important to consider the required accuracy for a study.

Addressed: We explicitly conduct our study at the release level to rule out temporary and experimental changes.

C2. Huge Changes between Releases

Rationale: In addition to hiding important information, discrete changes with long intervals between them exhibit more large restructurings at once than contiguous changes do. Instead of observing gradual re-engineering steps with occasional big-bang restructuring, large restructurings such as file renaming or restructuring of inheritance hierarchies become the norm rather than the exception.

Consequence: Clone detection tools need to be more robust to large restructurings such as copying and moving of clones, with more powerful mapping strategies between corresponding clones in different versions. Harder et al. [21] suggest to define and measure the correctness of mapping in order to provide the clone detection user with feedback about the robustness of the clone detection process.

Addressed: We adopt the CRD technique for clone tracking, which has shown high accuracy before [8].

C3. Irregular Release Schedule

Rationale: Whereas individual source code changes typically occur at a constant rate, with intervals typically going from seconds to a few days, releases tend to have periodic major releases and irregular minor releases (for example to fix security or stability issues). The time interval between releases ranges from days to months.

Consequence: The large variation in time interval between releases not only makes the consequences of challenges C1 and C2 worse, but has an impact on the interpretation of analysis results. A metric like the average number of clones per release does not make sense if one considers two distant major releases and two minor releases closely following the second major release.

Addressed: The release schedule of `jEdit` and `Mina` is regular and ranges from one day to three months.

C4. Parallel Releases

Rationale: The versioning numbering of releases is not necessarily chronological. Often a maintenance release for an older version of a software system (e.g., 3.1.4) is released at the same time as a new feature release of a newer version (e.g., 4.2). Typically, parallel releases share bug fixes but differ in the major features they provide, whereas sequential releases like 3.1.3 and 3.1.4 share major features but differ in bug fixes.

Consequence: The main consequence is that the release schedule has to be well-understood in order to determine between which versions corresponding clones should be mapped, i.e. within versions such as 3.x and 4.x or between chronological releases such as 3.1.4 and 4.2, and to consider whether it makes sense to compare the last 3.x release with 4.0.

Addressed: The release schedule of `jEdit` is sequential, whereas the release schedule of `Mina` is parallel starting from the 1.0.x and 1.1.x series. We resolved this minor issue with `Mina` manually.

C5. Selective Releases

Rationale: Official releases usually are cleaned up versions of the contents of the source code repository on a particular moment in time. Temporary or experimental files are removed, together with any confidential or private files.

Consequence: The missing files in official releases influence clone detection results. For example, it does not make sense to generalize findings about the number of *experimentation* variation clone groups [9] by only considering yearly releases. One should consider the goal of the study before selecting releases.

Addressed: We choose 22 releases from `Mina` and 50 releases from `jEdit`. The absence of temporary or experimental files does not affect our study.

C6. Traceability of Releases

Rationale: Whereas contiguous source code changes are typically explicitly linked to change log and bug report data, releases are only linked to the repository they originate from via their name (e.g., 2.1 alpha 1). Especially in older source control systems, it is hard to find the actual revisions and change logs of the specific versions of the files in the release.

Consequence: Unless more modern source control sys-

tems are used, recovering the traceability of releases usually requires a lot of manual browsing through source code and bug repositories and explicit combination of information from the different repositories.

Addressed: To overcome this challenge, we have 3 people manually inspecting change logs and bug reports individually, to make the results objective and accurate.

VII. Threats to Validity

In addition to addressing the challenges described in the previous section, we identified the following threats to the validity of our research.

- *Hidden impact of intermediate clones on end users.* Even though the impact of clones at the release level seems to be limited, clones in intermediate (developer) versions of the source code still might have a hidden impact on the software quality of the official releases, and hence on the end user. For example, maintenance problems with intermediate clones could have sparked a huge rewrite of important components, leading to bugs (not directly related to clones) and delayed releases.

- *Robustness of clone detection technique.* Our approach relies on the quality of the underlying clone detection tool to detect the clones inside a release. We countered this threat by a careful selection and evaluation of clone detection techniques. We settled on using the `SimScan` tool, which was used in previous studies in this research area [12], [27]. These studies report a good performance and accuracy of the tool.

- *Robustness of clone region descriptors.* Although CRDs greatly improve the robustness of finding clone regions in evolving source code, this technique is not without problems. CRDs encode the position of a cloned source code region as the position in the abstract syntax tree of a source code. However, if the source code is changed in a way such that nodes along the path from the root node to the subtree of the clone region are altered, tracking of the genealogy is lost. We tried to counter this problem by using transitivity as described in Section IV-C.

- *Lack of domain-specific knowledge.* The authors are not experts in network API programming, nor in the design of a text editor. Hence, our manual inspections might miss important insights, due to a lack of required domain-specific knowledge. We address this threat by having three different authors with diverse backgrounds and knowledge perform the inspection and discuss the results.

- *Generalizability.* As case study subjects, we picked two open source Java projects. Although we chose both projects to avoid potential biasing of our study towards any specific kind of software domain or size, our findings may not generalize to other open source projects of different nature. Due to the study of open source systems, where a large amount of developers freely contribute to the development of a projects, our findings may not generalize to an industrial setting. This threat can only be countered by doing additional case studies, especially on the effect of inconsistent changes in industrial projects, which are part of our future work. As many cloning patterns are specific to a certain programming language, our findings might not generalize beyond Java projects.

VIII. Conclusions

This paper presents an empirical study on inconsistent changes to code clones at release level, in order to evaluate the impact of these changes on the quality of software releases. Whereas previous work on clone detection is essential for the understanding of the immediate effects cloning has for day-to-day development processes, our study focuses on the impact of clones on the software quality of official releases, as perceived by the end user.

In both studied projects (`Apache Mina` and `jEdit`), we observe the presence of long-lived, yet small, clone genealogies. In contrast to traditional fine-grained analyses, we discover only a fraction of 1.26% to 3.23% of inconsistent changes to introduce software errors. The extensive manual inspection and categorization of clone genealogies suggests a number of possible explanations for this effect. The dominance of the *replicate and specialize* class of cloning means that the clones were meant to evolve independently, and hence cause many inconsistent changes.

As the number of defects through inconsistent changes, when observed at the release level, is substantially lower compared to observations at the revision level, we find that, for the two studied systems, the indirect effect of clones on the end-user seems limited. This confirms earlier findings [15] that developers are able to successfully manage inconsistent changes, even for long-lived clone genealogies.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback on earlier revisions of this paper.

References

[1] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1995, p. 86.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," *Software Maintenance, IEEE International Conference on*, vol. 0, p. 368, 1998.

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

[4] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, *Pattern matching for clone and concept detection*. Norwell, MA, USA: Kluwer Academic Publishers, 1996, pp. 77–108.

[5] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger, "Relation of code clones and change couplings," in *In Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE), number 3922 in LNCS*. Springer, 2006, pp. 411–425.

[6] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 485–495.

[7] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *24th IEEE International Conference on Software Maintenance (ICSM 2008)*. Beijing, China: IEEE, October 2008, pp. 227–236.

[8] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 158–167.

[9] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," *Reverse Engineering, Working Conference on*, vol. 0, pp. 19–28, 2006.

[10] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 61–82, 2006.

[11] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 187–196.

[12] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 81–90.

[13] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, Oct. 2007, pp. 24–33.

[14] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 170–178.

[15] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *ICSM '97: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1997, p. 314.

[16] R. Koschke, "Identifying and removing software clones." in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 15–36. [Online]. Available: http://dblp.uni-trier.de/db/series/springer/Mens2008.html#Koschke08

[17] C. Roy and J. Cordy, "A survey on software clone detection research," Queen's University, Kingston, Canada, Tech. Rep. 541, September 2007.

[18] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[19] J. H. Johnson, "Substring matching for clone detection and change tracking," in *ICSM '94: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1994, pp. 120–126.

[20] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755 – 765, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0B-46NXPDF-2/2/2e25956432f5ada61e0b045a49f49fe0

[21] J. Harder and N. Göde, "Modeling clone evolution," in *IWSC '09: Proceedings of the 4rd International Workshop on Software Clones*, Kaiserlautern, Germamy, March 2009.

[22] S. Bellon, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, 2007, member-Koschke, Rainer and Member-Antoniol, Giulio and Member-Krinke, Jens and Member-Merlo, Ettore.

[23] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," in *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 153–162.

[24] P. Kampstra, "Beanplot: A boxplot alternative for visual comparison of distributions," *Journal of Statistical Software, Code Snippets*, vol. 28, no. 1, pp. 1–9, 2008.

[25] J. Sim and C. C. Wright, "The kappa statistic in reliability studies: Use, interpretation, and sample size requirements," *Physical Therapy*, vol. 85, no. 3, pp. 257–268, March 2005.

[26] C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, and P. Weißgerber, "Subjectivity in clone judgment: Can we ever agree?" in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/970

[27] E. Duala-Ekoko and M. P. Robillard, "Clonetracker: tool support for code clone management," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 843–846.