ARTICLE IN PRESS

Science of Computer Programming ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

An empirical study on inconsistent changes to code clones at the release level

Nicolas Bettenburg*, Weiyi Shang, Walid M. Ibrahim, Bram Adams, Ying Zou, Ahmed E. Hassan

Queen's University, Kingston, Ontario, Canada

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Software engineering
Maintenance management
Reuse models
Clone detection
Maintainability
Software evolution

ABSTRACT

To study the impact of code clones on software quality, researchers typically carry out their studies based on fine-grained analysis of inconsistent changes at the revision level. As a result, they capture much of the chaotic and experimental nature inherent in any on-going software development process. Analyzing highly fluctuating and short-lived clones is likely to exaggerate the ill effects of inconsistent changes on the quality of the released software product, as perceived by the end user. 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 three open source software systems, we observe that only 1.02%–4.00% of all clone genealogies introduce software defects at the release level, as opposed to the substantially higher percentages reported by previous studies at the revision level. Our findings suggest that clones do not have a significant impact on the post-release quality of the studied systems, and that the developers are able to effectively manage the evolution of cloned code.

1. Introduction

Code clones are the source of heated debates among software maintenance researchers. Developers typically clone (copy) existing pieces of code in order to jumpstart the development of a new feature, or to reuse robust parts of the source code for new development. However, unless a clone is reused as is, developers quickly lose track of the link between the clone and the cloned piece of code, especially after some local modifications. Losing the links between clones increases the risk of inconsistent changes. These are code changes that are applied to only one clone, whereas they should propagate to all clones, such as defect fixing changes.

There is no consensus on whether the positive traits of cloning, such as effective reuse, outweigh its drawbacks, such as increased risk of deteriorated software quality. Many researchers consider clones to be harmful [3,6,14,21,22,27,36], due to the belief that inconsistent changes increase both maintenance effort and the likelihood of introducing defects. Yet, other researchers do not find empirical evidence of harm [39,47], or even establish cloning as a valuable software engineering method to overcome language limitations or to specialize common parts of the code [10,24–26]. It is not yet clear which of these two visions prevails, or whether the right vision depends on the software system at hand [15,43,47].

Empirical studies on code clones almost exclusively focus on the impact of cloning on developers, such as the developers' ability to keep track of all related clones in a clone group and their ability to consistently propagate changes to all clones. Many studies analyze inconsistent changes to clones and the general evolution (genealogy) of clone groups across very small

* Corresponding author. Tel.: +1 613 533 6802.
E-mail addresses: nicbet@cs.queensu.ca (N. Bettenburg), swy@cs.queensu.ca (W. Shang), walid@cs.queensu.ca (W.M. Ibrahim), bram@cs.queensu.ca (B. Adams), ying.zou@queensu.ca (Y. Zou), ahmed@cs.queensu.ca (A.E. Hassan).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.11.010

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

development increments (e.g., revisions) within a limited time interval [2,4,14,15,26,32,34,47]. Such studies provide insight into the developers' rationale for using code clones during refactoring or forward engineering, and the overhead of code clones on specific software development activities.

While studying the impact of code clones on developers is crucial for our understanding of code clones and the associated risks, we argue that it is equally important to understand the long-term impact of cloning on the quality of a software project as perceived by the primary stakeholders of a software system, i.e., the end user. Since many code clones are only introduced to quickly experiment with source code [26], and other clones gradually disappear over time because of on-going local changes [34], we conjecture that end users primarily perceive the effects of long-lived clones in a software system. The end user will likely not observe the actual cloning in the system, but instead perceive the effect of inconsistently changed clones in the form of software release defects.

In this paper, we conduct an empirical study on three large open source software systems on the relation of inconsistent changes to code clones with software quality, at the level of official releases. In particular, we address the following four research questions:

(Q1) What are the characteristics of long-lived clone genealogies at the release level?
(Q2) What is the effect of inconsistent changes to code clones on code quality when measured at the release level?
(Q3) How does the effect of inconsistent changes to code clones at the release level compare to finer-grained levels?
(Q4) Which cloning patterns are observed at the release level?

This paper is an extended version of an earlier conference paper [8]. The major differences are a discussion of the benchmarks we performed to select a clone detection tool, more details on our approach and the reports that are generated by our tool chain, and an additional case study and research question (`ArgoUML` in Q3).

The paper is organized as follows: Section 2 situates our work in the context of prior clone detection research. We present and motivate our four research questions in Section 3, then present the design of our case study in Section 4. Section 5 reports the results of our case study. Finally, Section 6 discusses threats to validity and Section 7 presents the conclusion of this paper.

2. Related work

Recently, multiple excellent surveys on code clones and detection tools have been published [29,40,44]. 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. [24] 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 [3,6,22,27] argue about the negative impact of code clones on software quality. Recently, for example, Lozano et al. [36] show that methods with clones require more maintenance effort than methods without. This effort increases with the number of code clones. Juergens et al. [21] find, after manual inspection of clones in four industrial and one open source systems, that inconsistent changes to clones are very frequent and can lead to significant numbers of faults in software.

Johnson [20] 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 the Linux kernel [1] and a large telecommunication system [34]. The latter study surprisingly finds that over time a significant number of clones automatically disappears from a system, and that most clones are never changed after their creation: Programmers seem to be aware of the clones in a system. Geiger et al. [14] conjecture that the larger the number of clones shared between files, the more likely that these files will change together. This conjecture would suggest that clones are consistently updated. However, no statistically significant results could be obtained.

Kim et al. [26] were the first to map clones in different versions of the source code to each other, so they could study the evolution of clone groups over time (genealogy), and analyze inconsistent changes inside genealogies. They report that up to 72% of the clone groups in an 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. [2] extend these findings, and report that most of the cloned code is consistently maintained in the same commit or shortly after. Similarly, Krinke [32] reports that half of the changes to code clone groups in the systems that he analyzed are consistent and that corrective changes following inconsistent changes are rare.

Recently, a number of clone evolution studies have emerged that suggest that the harmfulness of inconsistent changes is not a given fact, but depends on many different factors, such as the system at hand and the programming language [15,43,47]. Rahman et al. [39] found empirical evidence that the percentage of defects related to cloned code is very low (less than 15%) in the 4 applications that they studied, and that larger clone groups turn out to be less defect-prone than smaller ones.

This paper studies code clones at the release level to investigate the effect of inconsistent changes on the software quality as perceived by the end user, in particular the number of defects in the clones. Such a study of clones and defects has previously been done only at the revision level [2,4,15,17,21,26,32,39,47], as up until now studies at the release level focused on the evolution of clones instead of on (the impact of) inconsistent changes [1,10,14,20,34]. We believe that our work is

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

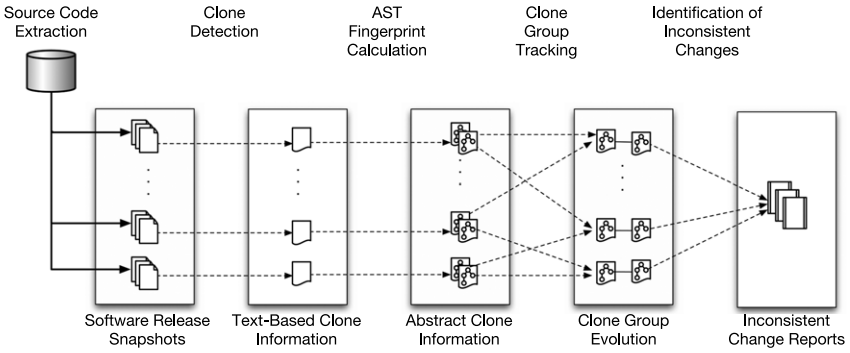


Fig. 1. Overview of our approach to study inconsistent changes of code clones at the release level.

complementary to the work of Rahman et al. [39], since we are interested in finding out of those defects that are related to clones, how many actually affect the end user in the form of post-release defects.

3. Research questions

This paper focuses on the impact of inconsistent changes to code clones at the release level. For this study, we formulate the following four research questions:

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

Previous research shows that many clones are short-lived [26] or "automatically" disappear over time [34], whereas other studies found that cloned fragments survive more than 1 year on average [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 in the management of these genealogies.

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

Measuring the effect of inconsistent changes on code quality at a fine-grained level, such as the revision level, might over-estimate the real impact of inconsistent changes, since some researchers found that many clones are short-lived [2,24,26]. Studying the effect of inconsistent changes at the release level filters out clones due to experimentation and refactoring, and allows us to focus on the nature and impact of long-lived, inconsistent changes.

(Q3) *How does the effect of inconsistent changes to code clones at the release level compare to finer-grained levels?*

There are strong indications that the harmfulness of code clones is application-dependent [15,43,47]. Project-specific development guidelines, the programming language in use and developer experience all influence the presence and impact of code clones. These are confounding factors for our findings for Q2: inconsistent changes to code clones might have a low (high) effect at the release level just because the number of inconsistent changes overall (even during development) is rather low (high). Hence, we explicitly need to compare the effect of inconsistent changes to code clones at the release level with the effect at finer-grained levels in a specific system.

(Q4) *Which cloning patterns are observed at the release level?*

If short-lived clones, as identified in previous research [26], are experimental in nature, can we find specific patterns of clones that survive multiple releases? Since different patterns of clones have different consequences for the maintenance effort and overall quality of a software system, it is important to study which patterns of clones prevail at the release level.

4. Study design

This section presents the design of our case study, which analyses inconsistent changes at the release level. Fig. 1 shows an overview of our approach. We first download all considered releases of the three open source software systems that we studied. For the finer-grained study in Q3, we used the date field of commits in the source code repository to identify the snapshots we need. For each individual release (or snapshot), 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 parts and manually inspect them to investigate whether they introduced errors into the software. For the purpose of this study we do not consider consistent changes (i.e., the exact same change carried out to all clones in a clone group), even though they too can potentially introduce errors into the software. Additionally, we perform a manual classification of clone genealogies into eleven categories [24]. This section elaborates on each of these steps.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

Table 1
Overview of the case study subjects.

	Apache Mina	jEdit	ArgoUML
#Releases	22	50	7
Start release	0.8.3	3.0.4	0.18
Start date	Oct. 01, 2006	Dec. 03, 2000	Apr. 30, 2005
End release	1.1.3	4.2.13	0.24
End date	Oct. 02, 2007	May 14, 2004	Feb. 12, 2007
Lines of code	11,908–15,463	47,500–88,305	118,656–165,261
Shortest release cycle	15 days	1 day	64 days
Longest release cycle	96 days	69 days	167 days
Average release cycle	51 days	36 days	112 days

4.1. Choice of subject systems

We chose three 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 easy development of network applications in Java. We chose Apache Mina, since we expect the system to likely contain a large amount of duplicate code due to the similarity between network protocols.
- jEdit[2] is a popular graphical text editor that has been used as a case study subject in related work [10,36].
- ArgoUML[3] is an open source UML modelling tool that has been used as a case study subject in related work [2,15,32,33,47]. We studied all sub-modules of ArgoUML.

We chose projects from differing contexts to help counter a potential bias of our case study towards any specific kind of software system. For Apache Mina and jEdit, 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. For ArgoUML, we extract 4 main releases in April 2005 (0.18), February 2006 (0.20), August 2006 (0.22) and February 2007 (0.24). As there was no official ArgoUML 0.19, 0.21 and 0.23 release, we extract minor releases 0.19.6 in September 2005, 0.21.2 in April 2006 and 0.23.4 in December 2006. Table 1 presents a detailed overview of our subject systems. The average length of a release cycle is 51 days for Apache Mina, 36 days for jEdit, and 112 days for ArgoUML.

4.2. Clone detection tools

Previous research in clone detection has produced a number of different techniques for the identification of duplicated source code. In the following, we describe the four most commonly used approaches for clone detection [29,40,44]:

- **Text-based** clone detection techniques work on the source code of the software system and use text transformation and normalization approaches like pattern matching and substring matching, or data mining techniques like latent semantic indexing (LSI) [12,37]. Text-based approaches are usually programming language-independent and scale well to large code bases containing millions of lines of code. However, most of these approaches are not robust to modifications of the cloned source code that are commonly carried out during software development, such as adding and deleting lines of code.
- **Token-based** clone detection techniques work on higher-level abstractions of the software system. Using a lexical analysis, the textual representation of the system's source code is transformed into token sequences, which are then surveyed for duplications. These techniques can be made robust to minor code modifications [22]. While token-based techniques are usually able to find a higher number of clones than other approaches, they report many false positives. As a result, additional manual verification of the clone detection results is needed, thus rendering these approaches less scalable for large-scale studies.
- **Syntax-based** clone detection techniques evaluate the similarity of source code blocks by calculating and comparing metrics on a syntax tree representation of the code [6,19,30]. These techniques usually produce a high level of precision in their results, but only moderate recall [7]. However, syntax-based approaches do not scale well to large-scale systems and typically require compilable source code, rendering them less valuable for measuring clones at the revision level.
- **Semantics-based** clone detection techniques work on the program dependency graph level of a software system and hence require a thorough reverse engineering of the software system under study [31,35]. While empirical studies on the performance of these approaches reported very good results in terms of precision and recall [41], these techniques are usually hard to implement and do not scale well to the size of real-world software projects [13].

1 http://mina.apache.org/, last checked June 2009.
2 http://www.jEdit.org/, last checked June 2009.
3 http://argouml.tigris.org/, last checked February 2010.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

Table 2
Tool versions and parameters used in our benchmarks and case studies.

Tool	Version	Parameters
CCFinderX	10.2.7.3	min_length=50, chunk_size=60M, block_shaper=2, minimum_size_of_token_set=12
Simian	2.2.24	failOnDuplication=true, ignoreCharacterCase=true, ignoreCurlyBraces=true, ignoreIdentifierCase=true, ignoreModifiers=true, ignoreStringCase=true, threshold=6
SimScan	R1 Integrated	equality_depth=4, upper_razor=1000000, significant_node_depth=4, full_check_group_size_comparisons=20, equality_group_consideration_ratio=20, equality_group_consideration_success_ratio=0, minimal_search_success_ratio=0, group_skips_for_stop=1000000, strictness=60, minimal_match_weight=60, minimal_common_sequence_size=5, minimal_common_sequence_ratio=40, correspondence_strictness=5, maximal_single_correspondences_ratio=50, similar_children_bridge_length=3, speed=3 [`Apache Mina/jEdit`], speed=4 [`ArgoUML`]

The goal of our research is to study code clones at the release level. Between releases, developers usually carry out numerous changes to different parts of a software system. As a result, the source code we observe at release R_{i+1} may be significantly different from the source code we observed in the previous release R_i. As opposed to past studies on code clones in evolving source code, which were carried out at a much finer granularity level, we need to choose a clone detection approach that is robust enough to changes carried out to clones, while retaining a sufficiently high level of precision and recall during detection across releases.

Based on existing surveys of clone detection techniques [29,40,44], as well as previous studies on the evolution of code clones [2,4], we selected three of the most popular tools for clone detection to test their applicability for detecting code clones in multiple releases of a source code project. These tools were `CCFinderX` [22], `Simian` [18] and `SimScan` [11].

In order to select the best tool out of these three, we developed and ran a benchmark suite. For better repeatability of our study, we present the versions and parameters used for all clone detection tools in Table 2. The benchmark contains clones generated by the 13 different developer editing actions that can generate clones [42], distributed across the four traditional categories (“types”) of clones [7,28]. We ran each clone detection tool on the benchmark code to measure their robustness to changes and their detection capabilities. Our results are summarized in Table 3, grouped by the clone type produced by carrying out each editing action. In the following, we will outline each of the clone types, the editing actions associated with them, and the results of our benchmarking process.

Type-1 clones are identical parts of the source code, except for differences in whitespace, layout and code comments. We carried out three different editing actions to produce Type-1 clones: (1) exact copying and pasting of code, (2) copying and pasting with additional changes to comments and whitespace, and (3) copying and pasting with additional reformatting of the source code by moving the opening and closing brackets. *All three tools were robust to these modifications.*

Type-2 clones are parts of the source code that are identical on a syntactical level, except for differences in data types, identifier names, literals, or the differences for Type-1 clones. To produce Type-2 clones, we carried out three different editing actions: (1) renaming function parameter names in copied code, (2) changing parameters into expressions, and (3) systematic renaming of identifiers. *`CCFinderX` and `SimScan` were robust to all three editing actions; `Simian` could not detect the clones produced by substituting parameters by expressions.*

Type-3 clones are similar parts of the source code in which statements might have been added or removed, in addition to the differences outlined for Type-2 clones. To produce Type-3 clones, we carried out seven different editing actions after copying code: (1) deletion within a line, (2) deletion of a whole line, (3) insertion within a line, (4) insertion of multiple lines, (5) modifications of multiple lines, (6) reordering of declarations, and (7) reordering of lines and statements. *`SimScan` was able to detect clones produced by all seven editing actions, `CCFinderX` was able to detect fragments of clones produced by five out of seven editing actions, and `Simian` detected parts of three of the seven code clones produced by the editing actions.*

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

Table 3
Comparison of the robustness of clone detection tools against different types of editing actions defined in the editing taxonomy by Roy et al. [42].

	Edit operation	SimScan	CCFinder	Simian
Type 1	Exact copy	+	+	+
	Editing comments and whitespace	+	+	+
	Formatting changes	+	+	+
Type 2	Changing types	+	+	+
	Syntactic renaming	+	+	+
	Substituting parameters with expressions	+	+	o
Type 3	Insertion within a line	+	o	o
	Deletion of a line	+	o	−
	Deletion within a line	+	o	o
	Insertion of one or more lines	+	−	−
	Reordering of declarations	+	o	o
	Changing one or more lines	+	o	−
	Reordering of statements	+	−	−
Type 4	Semantically equivalent control structure	−	−	−
Legend	+ *robust to editing actions and detects full clone* o *not robust to editing actions, but detects clone partially* − *not robust to editing actions, fails to detect clone*			

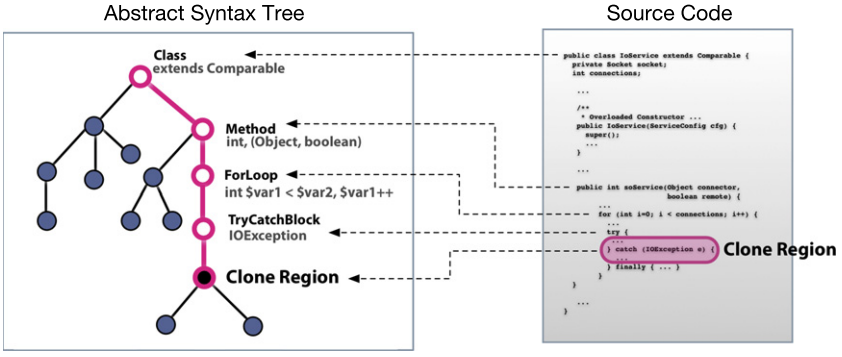


Fig. 2. CRD generation using abstract syntax trees.

Type-4 clones are parts of the source code that have the same functionality, but different implementation. To produce a Type-4 clone we replaced `for` control statements in the copied source code with `while` constructs and replaced the code inside the `for` loop with a semantically equivalent computation. *None of the clone detection tools was able to discover the clone produced by this editing action.*

Based on the results of our comparison, we decided to use the `SimScan` clone detection tool for the remainder of our case studies. `SimScan` is a clone detection tool for Java systems based on the `ANTLR` parser generator framework, and has previously been used by other researchers [2,10] in the area. The output generated by the tool is very detailed and easy to parse, making it suitable for automated analyses such as those used in our study. Similar to other `syntax-based` clone detection tools, `SimScan` exhibits scalability issues, which became apparent when we applied it on the code bases of larger projects like `jEdit` and `ArgoUML`. For this reason, we had to use a higher speed option for `ArgoUML` (Table 2), which ignores some clones that are too dissimilar. This option significantly reduced run-time overhead. We performed a manual inspection of clone detection results with both speed options, but this analysis showed no significant differences for the purpose of this study.

More recently, various techniques have been proposed to scale up clone detection, such as incremental clone detection algorithms [5,15,16] and web-scale techniques [45]. We believe that future studies require these techniques to perform large-scale clone evolution studies.

4.3. Clone tracking between releases

To study the evolution of clones in a clone group, we need to track clone groups across different versions. A clone genealogy [26] for a particular clone group and version of the source code is a directed acyclic graph that 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.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

Various techniques have been used to map clone groups in different source code versions to each other [4,10,26,32]. We chose to use the *clone region descriptors* (CRD) technique [10], which had been combined successfully with SimScan 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 (Fig. 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".

While traversing the abstract syntax tree (AST) 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*.

To discriminate between code entities with the same CRD (e.g., two for-loops in the same method), we use so-called corroboration metrics. The corroboration metric that we use is the ordering of code clones within the next larger containing entity. This is based on the heuristic that the order of code clones rarely changes in a particular containing entity. If the order does change, our tool may track the wrong code clones.

A greater risk when using CRDs for clone tracking is refactoring. If any node changes along the path of the abstract syntax tree from the cloned code region to the root node of the AST, we consider this as a "new" clone region. However, if there are any other clone regions in the clone group whose CRD path did not change due to refactoring, i.e., the clone group changed inconsistently, we are still able to track the clone group through the transitivity heuristic described earlier. Our method only loses track of a clone group when the CRDs of all clones in a clone group changed, but in that case the clones likely changed consistently.

Finally, to identify inconsistent changes, we filter out all clone groups in which there was no change at all or in which all clones changed together between two consecutive versions of the source code. The filtering uses textual comparison to analyze the source code of the corresponding clone regions, ignoring whitespace changes (but not comments). The resulting clone groups have at least one clone that did not change, i.e., they are likely the result of an inconsistent change. Manual analysis is needed to verify this. To support this manual analysis, we generate a marked-up report of the textual differences between the two versions of the clone, suitable for human interpretation.

Fig. 3 shows an annotated screenshot of a report for the `jEdit` project of a clone group that has been changed inconsistently. The first line of each clone contains the clone's CRD, i.e., the unique address of the clone in the source code. In this case, the first clone is located inside the actionPerformed() method of the ActionSetModelElementNamespace class in the org.argouml.uml.ui.foundation.core package. The second clone is located inside the actionPerformed() method of the ActionSetModelElementStereotype class in the same package. The clone group itself contains two clones. In one of them, code was added and removed since the previous version, but no corresponding changes were carried out in the other clone. Hence, we consider this to be an inconsistent change. The inconsistent change is shown in the report via colored and possibly crossed out text, as indicated on Fig. 3. The CRDs (reported in the gray box on top of each clone) are very helpful in identifying the source code location of a clone when doing manual inspection of inconsistent changes.

4.4. Manual inspection of inconsistent changes

We perform a manual inspection of each inconsistent change to a clone genealogy reported by our clone tracking tool. 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 defect into the software system. Most of the times, our own judgment is able to distinguish between inconsistent changes and false positives. In case of doubt, we first check the source code repository to find out if a change with similar semantics has been applied to another part of the same clone group at a later point in time. If so, the change is inconsistent for sure, otherwise the change was either intentional or it caused a defect. To help determine the right case, we consulted the commit messages of the version archives, as well as the projects' bug tracking repositories, mailing list archives and documentation to check if the change in question is mentioned. If no supporting evidence was found, we used our best judgment to interpret the change.

In order to account for possible human error, four authors of this paper carried out this inspection (using external data sources) independently, then compared the results. Voting was used in case of disagreement. In case of a tie (rare), each party tried to convince the other party of her view.

4.5. Classification of clone genealogies

Cloning of source code can occur due to different reasons. Kapser et al. identified eleven patterns of cloning, each having a distinct purpose, as well as short and long-term management issues [24]. In order to better understand the clone genealogies of each project and to assess the overall risks, four authors of this paper act as human oracles, who independently perform a manual inspection of the source code and classify each discovered clone genealogy into one of eleven categories. Again, voting was used in case of disagreement. In case of a tie (rare), each party tried to convince the other party of her view.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

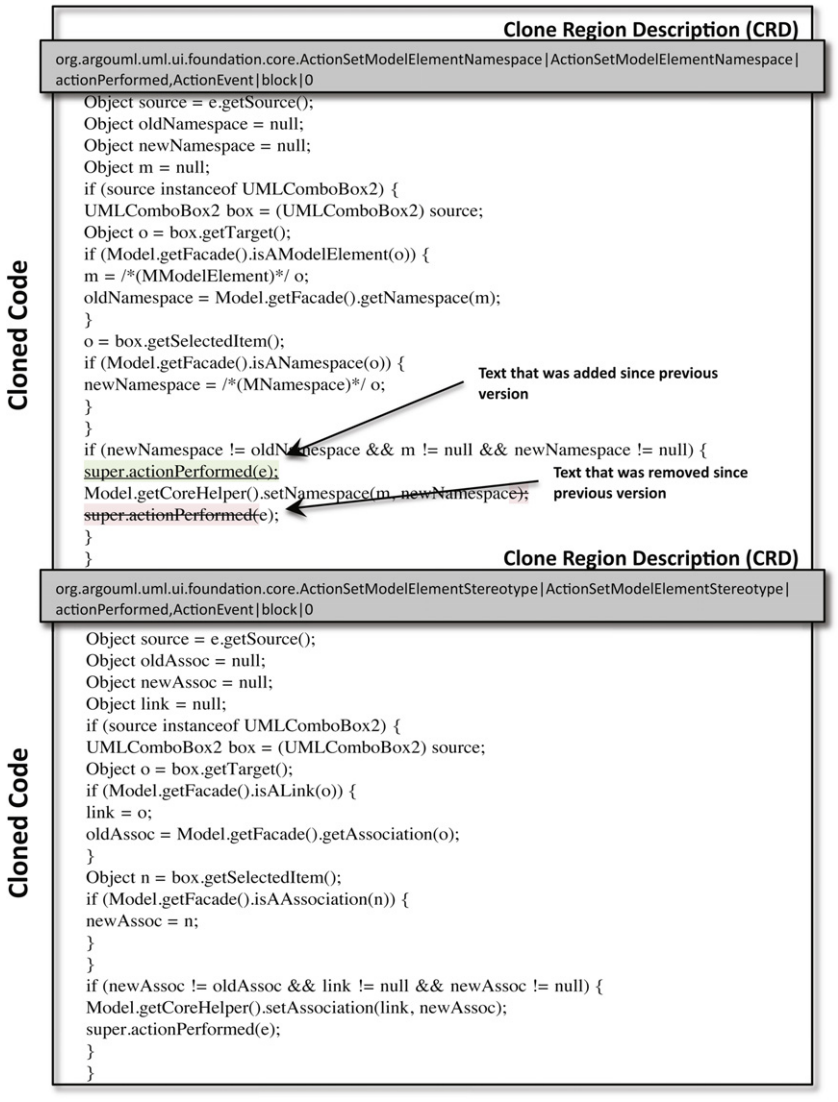


Fig. 3. Annotated screenshot for a report of an inconsistent change.

5. Case study results

This section presents the findings of our case study with respect to our four research questions. After a quantitative release level analysis of the code clones and code clone groups in `Apache Mina`, `jEdit` and `ArgoUML` (Q1), we present our findings on the impact of inconsistent changes to clone genealogies at the release level in these three systems (Q2). Then, we compare the impact of inconsistent changes for the release and revision levels on the largest of our subject systems, i.e., `ArgoUML` (Q3). To better understand the findings of our three case studies, we conduct a classification of clone groups (Q4). Table 4 summarizes our findings on inconsistent changes and defects.

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

We detect a total of 1,387 groups of code clones spread across 22 releases of `Apache Mina` and a total of 11,160 groups of code clones across 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`, the set of 11,160 unrelated groups to 818 genealogies for `jEdit` and the set of 6,430 unrelated groups to 1,574 genealogies for `ArgoUML`.

We then measure the average lifetime and size for these genealogies. Our findings are presented as kernel density plots in Figs. 4 and 5. A kernel density plot estimates the probability density function of a variable, and can be interpreted as a continuous form of a histogram. Fig. 4 shows the probability for genealogies of having a particular lifetime (in # releases),

Table 4
Summary of inconsistent changes and defects found in Apache Mina, jEdit and ArgoUML.

	Apache Mina	jEdit	ArgoUML (release)	ArgoUML (weekly)
Total #clone groups	1,387	11,160	6,430	74,509
Total #clone genealogies	306	818	1,574	1,619
#discarded genealogies	254	602	1,181	1,151
#false positive genealogies	2	74	2	2
#genealogies with incons. change	50	142	391	466
Total #incons. changes flagged	85	679	708	1,431
#reformatting incons. changes	10	6	247	464
#false positive incons. changes	13	277	4	4
#true positive incons. changes	62	396	457	963
#incons. changes with defect	2	5	4	19

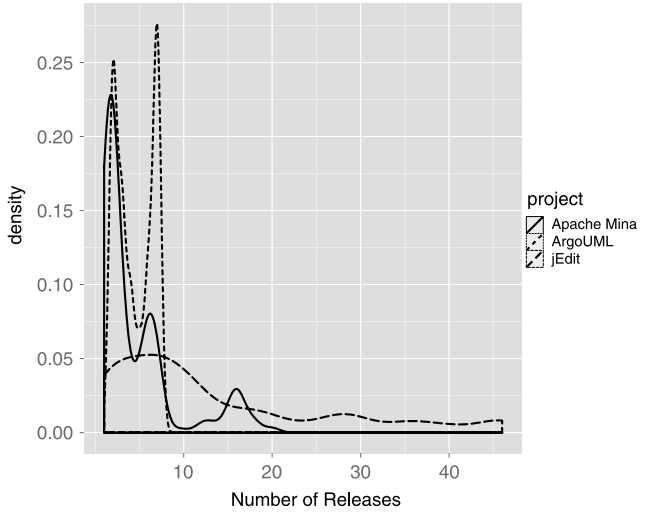


Fig. 4. Comparison of the distribution of lifetime (in # releases) of the genealogies for Apache Mina, jEdit and ArgoUML.

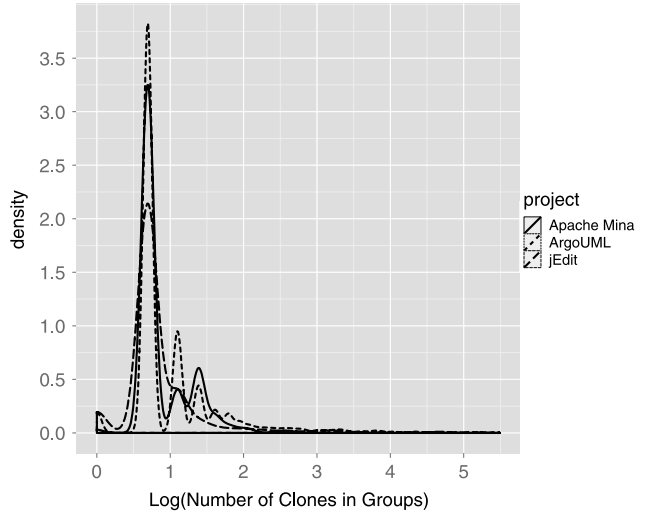


Fig. 5. Comparison of the distribution of average size (in # clones) of the genealogies for Apache Mina, jEdit and ArgoUML.

whereas Fig. 5 shows the probability for genealogies of having a particular average size (in #clones). Kernel density plots allow easy comparison of multiple distributions in one plot.

Fig. 4 shows that Apache Mina and ArgoUML have similar lifetime distributions up until 7 releases (maximum number of releases for ArgoUML). Afterwards, Apache Mina has a peak at 16 releases. jEdit has a maximum lifetime around 7,

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

after which the probability keeps on decreasing. Fig. 5 shows that most genealogies have an average clone group size of 2 clones. Apache Mina and ArgoUML have smaller peaks for 3 and 4 clones, whereas jEdit only has a slight peak for 3 clones.

For Apache Mina, 79.7% (244 out of 306) of all clone genealogies have a lifetime that spans multiple releases. The average lifetime of a genealogy in Mina is 4.59 releases and the average size of a genealogy is 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 the 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 is 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.

For ArgoUML, 80.7% (1,271 out of 1,574) of the clone genealogies have a lifetime that spans multiple releases. The average lifetime of a genealogy in ArgoUML is 3.8 releases and the average size of a genealogy is 4.16 clones. The largest clone genealogy in ArgoUML contains on average 262 cloned parts during its lifetime. It consists of generated parts of Java and IDL parsers.

Overall, we find that jEdit has clone genealogies with longer lifetimes than Apache Mina and ArgoUML, but a smaller average clone group size. The average lifetime of clone genealogies across releases is 5.80 releases and the average size is 2.95 cloned parts.

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

To study inconsistent changes at the release level, we need to look at the evolution of the clone genealogies obtained for Q1. As we aim to study the relation between inconsistent changes and defects, we distinguish between *reformatting changes*, such as code beautification, and *syntactical changes*, which modify the actual source code.

For Apache Mina, we record a total of 85 inconsistent changes, of which 10 were flagged as inconsistent reformatting changes. For jEdit we record a total of 679 inconsistent changes. Of these, 6 changes were flagged as inconsistent reformatting changes. For ArgoUML, we record 708 inconsistent changes in total, of which 247 are reformatting changes. We then discard inconsistent reformatting changes and perform a detailed manual inspection of the remaining 1,209 inconsistent changes (75 for Apache Mina, 673 for jEdit and 461 for ArgoUML).

During the manual inspection of the inconsistent changes, we found that a number of genealogies generated by SimScan are false positives. First, we discarded those genealogies without inconsistent changes. This eliminates 254 genealogies of Apache Mina, 602 genealogies of jEdit and 1,181 genealogies of ArgoUML. In the remaining genealogies, we identified false positive genealogies that have clones with very similar syntactical structure (for example, a for loop or switch/case-structure), but are otherwise unrelated. In order to keep our study sound, we decided to ignore clone genealogies generated by these false positive groups. For Apache Mina, we found 2 false positive genealogies that account for a total of 13 changes. For jEdit, we found 74 false positive genealogies that account for a total of 277 changes. For ArgoUML, we found 2 false positive genealogies that account for 4 changes.

In the remaining total of 915 inconsistent changes (62 for Apache Mina, 396 for jEdit and 457 for ArgoUML), we found eleven inconsistent changes that led to software defects: two in Apache Mina, five in jEdit and four in ArgoUML. 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 a 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 SocketIO Processor class between version 0.9.5 and 1.0.0. This call would 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 behaviour 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.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

Table 5
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
ArgoUML	0.16.1	0.18	1	222
ArgoUML	0.21.2	0.23.4	1	234
ArgoUML	0.21.2	0.22	0	110
ArgoUML	0.20	0.22	1	178

- **[jEdit]** A request for screen 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.
- **[ArgoUML]** In revision 8,854, a typo in a variable name was fixed. This typo was introduced right after cloning during the parameterization of the clone. The compiler did not warn developers for this typo, as there was an existing private variable with the same name. Our clone data only contains the bug fix for this defect, which means that the original inconsistent change causing the defect occurred before the time interval that we studied. This means that the typo, and the defect, appeared before release 0.18. Eventually, we found that this defect was introduced in release 0.16.1.
- **[ArgoUML]** A bug fix (Issue#: 4,162) was applied in revision 10,395 to check if an object's owner is NULL when evaluating simple OCL expressions. This change was not propagated to the other clones in the group within the two years of `ArgoUML` data that we analyzed. It took until revision 12,302 before the change was finally propagated.
- **[ArgoUML]** In revision 10,106, the tool tips of various GUI actions such as removing objects from a diagram were made more user-friendly by providing a short description of the actions. Many actions were implemented as clones, but not all of the clones saw their tool tip string updated at revision 10,106. Only later on, in revision 10,733, the remaining tool tip strings were updated.
- **[ArgoUML]** A bug fix (Issue#: 3,651) was applied in revision 9,287 to avoid testing if an object can be selected by the user if the object has been deleted. This bug fix was not propagated to all clones in the clone group. A later bug fix in revision 10,478 propagates the change to all other clones in the clone group consistently.

For each of the eleven 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 5. For `Apache Mina`, all defects were fixed very quickly with no intermediate defective release. For `jEdit`, three defects were fixed quickly within one intermediate defective release, and two defects were present in the software over a very long period of time. For `ArgoUML`, one defect was fixed with no intermediate release, and three defects were fixed with one intermediate defective release.

In addition to the aforementioned software defects, we encountered two instances in `jEdit` of what we believe to be undetected defects 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 was impossible, as the suspicious parts disappeared in later versions due to refactoring, without being noticed in the meanwhile.

Overall, we find that only 4.0% of the genealogies in `Apache Mina` (2 defects for 50 genealogies), 3.52% of the genealogies in `jEdit` (5 defects for 142 genealogies) and 1.02% of the genealogies in `ArgoUML` (4 defects for 391 genealogies) caused a defect in the software. This contrasts the findings of studies on other systems performed at fine-grained levels (such as commit-level). Those studies report a substantially higher fraction of defect introducing genealogies [2,21,39]. The next question directly contrasts release level findings to finer-grained findings for one and the same system.

(Q3) *How does the effect of inconsistent changes to code clones at the release level compare to finer-grained levels?*

We compared inconsistent changes at release and fine-grained levels in `ArgoUML`, since it is a larger system than `Apache Mina` and `jEdit`, and has been studied before in he context of clone detection [2,15,32,33,47]. Similar to those studies, we study the clone genealogies at the weekly level because of the large number of cloning results to analyse. We restrict ourselves to the time frame between major releases 0.18 and 0.24. We choose these two major releases, because SimScan does not support Java 5.0 code (0.24 was the last major `ArgoUML` release in Java 1.4).

Using the same approach as for Q1 and Q2, we detect a total of 74,509 clone groups spread across 94 weekly snapshots (653 days) and a total of 6,430 code clone groups in the 7 releases of `ArgoUML` that we studied. The generation of clone genealogies via clone group tracking reduces the set of 74,509 unrelated clone groups for the weekly level to 1,619 clone genealogies, and the set of 6,430 unrelated clone groups for the release level to 1,574 genealogies. After discarding clone

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

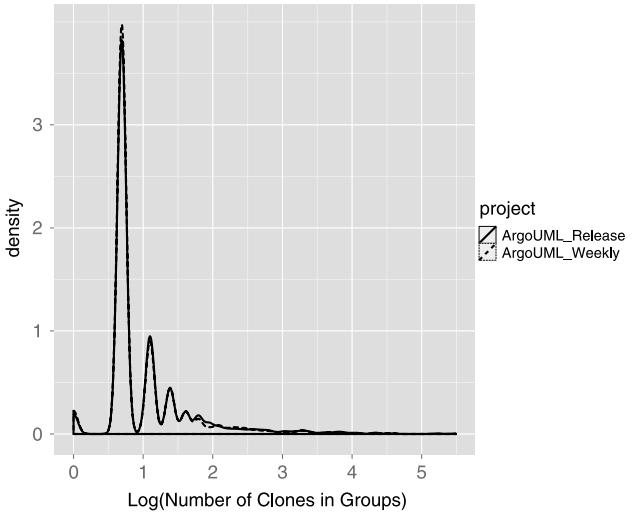


Fig. 6. Size of Clone Groups for the weekly snapshots of ArgoUML.

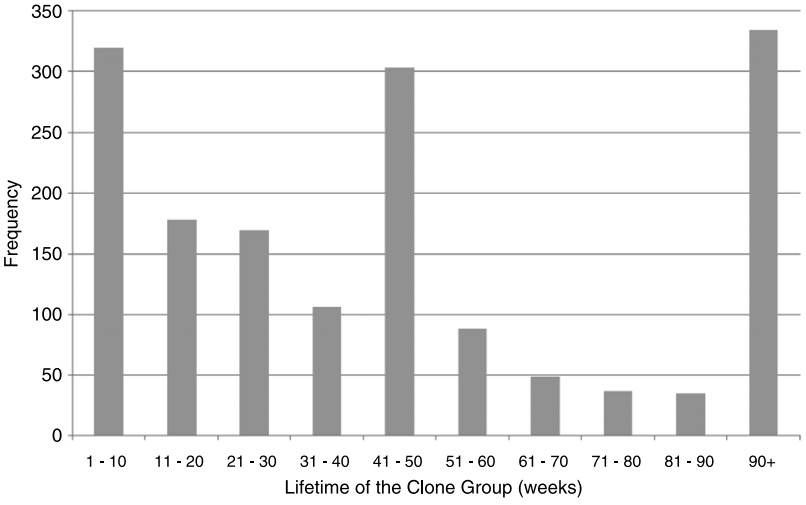


Fig. 7. Histogram of the lifetime of clone groups for the weekly snapshots of ArgoUML.

genealogies without inconsistent changes and false positive genealogies, we end up with more genealogies at the weekly level, because some genealogies are too short-lived, and our tool lost track of the links between some other clone groups. At the weekly level, the changes between clone groups are smaller, making it easier to link clone groups in subsequent snapshots to each other.

We then measure the average lifetime and the distribution of the clone size for the recovered genealogies at the weekly level. The distributions of the clone size at weekly and release level are presented as kernel density plots in Fig. 6. To study whether there is a statistical significance between the distribution of clone group size between weekly and release level, we carried out an unpaired, 2-sided t-test (Mann-Whitney). Based on our findings, we cannot refute the null hypothesis that the 2 distributions are statistically identical (i.e., there is no significant difference in the distribution of clone group size between weekly and release level at $p < 0.05$.)

Fig. 7 shows a histogram of the distribution of the lifetime of clone groups as a histogram. We can see that the lifetime of clone genealogies steadily decreases for longer lifetimes, except for a peak at the end and in the middle. The peak at the end contains all genealogies that were in the first studied snapshot and still exist in the last studied snapshot. For the weekly level, 20% (330 out of 1,619) of clone genealogies have a lifetime that spans across all studied weeks (96.5%, i.e., 1,563 out of 1,619, span across at least one week), with an average lifetime of 43.74 weeks and average genealogy size of 3.5 clones. If we compare to the release level, we see that slightly more than 24% (380 out of 1,574) of clone genealogies at the release level lived through the whole studied period (between release 0.18 and 0.24), and that the average genealogy size of 4.16 clones is larger as well.

Furthermore, the peak in the middle of the histogram seems strange. After analysis, we found that this peak coincides with the massive removal of source code that was reported earlier by Krinke [33]. During this period a large set of generated

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

files for a bytecode parser were deleted, containing many clone files. We did not discard these generated files from our analysis. Generated code is often (sometimes accidentally) changed or optimized by hand, or only partially re-generated. Such hard-to-find defects can be easily detected via inconsistent changes between clones in the generated code.

To study the differences between inconsistent changes and defects at the release level and revision level, we look at the evolution of the clone genealogies obtained at the release level and weekly level. For the weekly level, we record a total of 1,431 inconsistent changes (Table 4), of which 464 were flagged as inconsistent reformatting changes and 4 changes from 2 clone groups were false positives. For the release level, we record a total of 708 inconsistent changes. Of these, 247 changes were flagged as inconsistent reformatting changes and 4 changes from 2 clone groups were false positives. We then discard all inconsistent reformatting changes and the false positive data. We ended up with a total of 963 inconsistent changes for the weekly level and 457 inconsistent changes for the release level.

By detailed manual inspection of the 963 inconsistent changes from the weekly level, we found 19 inconsistent changes that led to either a defect or a bug fix: 15 defects/bug fixes appeared only in the weekly level and 4 defects appeared in both the release level and weekly level (described in Q2). We discuss four representative examples of weekly level-only inconsistent changes with a defect:

- **[Weekly level]** A bug fix (Issue#: 2,287) was applied in revision 8,085 to automatically select an `ArgoUML` diagram when the diagram is added to the `ArgoUML` explorer tree. However, this bug fix was not propagated to the other 2 clones in the clone group. A later inconsistent change was made to this clone group to make it consistent again in revision 8,129. This new change fixed 2 defects (Issue#: 1,833 and 3,177).
- **[Weekly level]** A bug fix (Issue#: 2,355) was applied in revision 9,533 to make file chooser dialogs used for saving `ArgoUML` diagrams remember the last directory that was opened. To solve this issue, file choosers are passed the path and filename of the file to save, without a file extension. This fix was introduced to one of the clones and was not propagated to the other clone. We believe this is a defect, although we did not find any bug report on it.
- **[Weekly level]** A bug fix (Issue#: 4,248) was applied in revision 10,646 to wrap certain source code regions by exception handling code. However, not all clones in the clone group were wrapped by the exception handling source code. To fix this, the change was propagated to the other clones in the clone group in revision 10,768.
- **[Weekly level]** In revision 8,422, a method invocation that performs action of a Java AWT event was moved to enable undo of this action. The clone of this piece of code was not changed in revision 8,422. To fix this issue, a later change was applied in revision 10,733.

Our comparison of inconsistent changes to clone groups at the weekly level and at release level indicates that most of the defects (15 out of 19) caused by inconsistent changes appear and disappear before a release is cut. More specifically, 4.08% of the genealogies that have inconsistent changes at the weekly level (19 out of 466) caused a defect, whereas only 1.02% of the genealogies that have inconsistent changes at the release level (4 out of 391) caused a defect. We believe that this is due to heavy testing before release, as well as the ability of developers to somehow keep track of clones [47]. However, as 4 of the defects caused by inconsistent changes appear at the release level, and hence affect the end user, developers might still benefit from clone management IDE support to pro-actively manage clone genealogies [9,10,38].

(Q4) *Which cloning patterns are observed at the release level?*

In order to understand the dominant 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 [24].

Four judges independently categorized all clone groups from the three subject systems into eleven categories. For the four 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 [46]. While discussing our ratings, we found that most disagreements rooted in subtle semantics of the source code, which blurred the borders between categories.

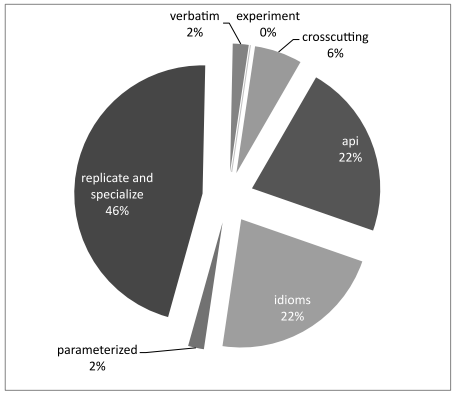
In particular, we found two main reasons for disagreements:

1. Different patterns of code clones are intertwined in one clone group. For example, in one clone some identifiers are modified (*parameterized code*), whereas another clone in the same group is specialized for the specific context of the clone (*replicate and specialize*). It is hard to determine the categorization of such a clone group.
2. Subtle semantics of the source code blur the borders between categories. For example, the *boiler-plating* and *parameterized code* cloning patterns are both modifying one clone as a template to solve similar problems. Even though changing data types in a clone is considered to be the *boiler-plating* pattern and changing identifiers or literals in a clone is considered to be the *parameterized code* pattern, clones that change both data types and identifiers are hard to categorize.

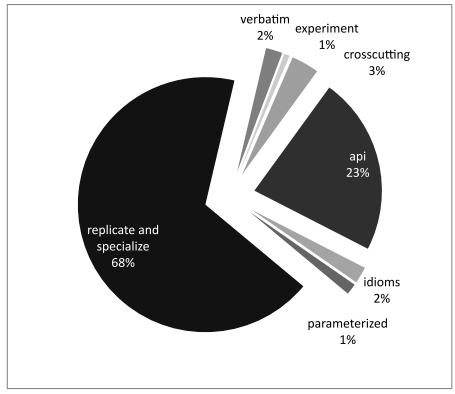
To solve our disagreements, we tried to follow the spirit of the cloning categories instead of following strict rules. If one clone group consists of different categories of cloning, we determined the majority to be the category of the clone group. Kapser et al. observed a similar problem in an experiment that showed the difficulty of defining and classifying code clones [23].

The results of our classification of clone genealogies are presented in Fig. 8. For all systems, the majority (46% for `Apache Mina`, 68% for `jEdit`, 44% for `ArgoUML`) at release level of long-lived code clones were found to belong to the *replicate*

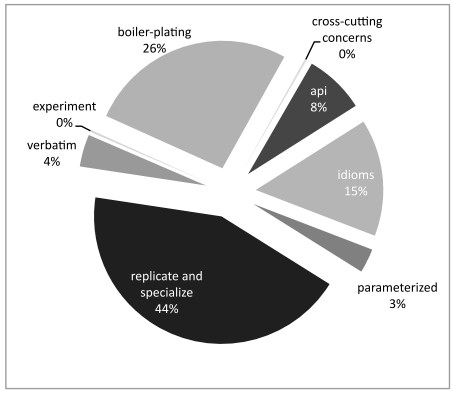
Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010



(a) Apache Mina.



(b) jEdit.



(c) ArgoUML at the release level.

Fig. 8. Classifications of clone groups in Apache Mina, jEdit and ArgoUML at the release level, according to the classification of Kapser et al. [24].

and specialize cloning pattern. In this form of cloning, existing code with similar functionality is copied and customized to implement new functionality for the software. We found that changes to such clones in the subject systems are usually carried out inconsistently because the cloned parts evolve independently.

The second largest pattern of cloning we found in jEdit and Apache Mina is the *API* cloning pattern, which describes the cloning of a series of program steps, pre-determined by the usage of a specific interface. For ArgoUML, however, the second largest pattern of cloning we found is *boiler-plating*, which reuses trusted source code to perform consistent or similar behaviour by replacing data types (instead of using generics or polymorphism). The importance of this kind of these cloning patterns confirms the results of other studies [47].

Cloning due to *language idioms* forms the third largest class of code clones in Apache Mina and ArgoUML. Mina has a network API that makes heavy use of Java exception handling, iterators and data structures, which involves tedious, yet frequently needed code fragments. ArgoUML has similar needs.

The remaining clone patterns observed in the three projects are *verbatim* copy and pasting, cloning to implement *cross-cutting 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. In ArgoUML, we found one case of cloning due to *experimentation* at the release level: a class named ModuleLoader is replicated and renamed to ModuleLoader2 to perform experimental changes. Since this is the only experimental clone in the release level data, this confirms our belief that studying clones at the release level largely filters out temporary clones, which are done for experimentation.

Of the eleven inconsistent changes at the release level that introduced software defects, we found four defect introducing inconsistent changes to clone groups of pattern *replicate and specialize*, two defect introducing inconsistent changes to clone groups of pattern *API*, one defect introducing inconsistent change to a clone group of pattern *experimentation*, one defect introducing inconsistent change to a clone group of pattern *boiler-plating* and three defects introducing inconsistent changes to a clone group of pattern *verbatim*. These numbers reflect the relative importance of each pattern as shown in Fig. 8.

To conclude, our observations for Q4 confirm that most clones in the studied subject systems are meant to diverge, i.e., inconsistent changes typically will not introduce defects. Of course, repeated inconsistent changes can change the clones of a clone group to such a degree that sudden consistent changes become harder to propagate to the full clone group, as shown by the existence of the defects found in Q2 and Q3. Yet, as the fraction of defects introduced through inconsistent

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

changes is quite limited, we believe that the developers of both projects are somehow aware of these long-lived clones in their software systems and are able to effectively manage the independent evolution of these clones.

6. Threats to validity

We identified the following threats to the validity of our research.

Coarser-grained clone tracking has lower precision. Too long intervals between analyzed code versions make it much harder for clone tracking tools to connect clone groups in subsequent snapshots to each other [17]. To address this threat, we adopt the CRD technique for clone tracking, which has shown to have high accuracy [10].
Hidden impact of intermediate clones on end users. Even though the impact of clones at the release level is limited in our subject systems, clones in intermediate (e.g., weekly) 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 caused by intermediate clones could have slowed down development or required a huge rewrite of important components, leading to defects (seemingly unrelated 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 in Section 4.2. We settled on using the `SimScan` tool, which was used in previous studies in this research area [2,11]. These studies report a good performance and accuracy for the `SimScan` tool. However, we encountered scalability issues when using `SimScan` to detect clones in the large code bases of `jEdit` and `ArgoUML`. Recent research in the area of clone detection has recognized these problems and developed more scalable solutions in the form of incremental clone detection methods [5,15,16].
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 such a way 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 clone transitivity, as described in Section 4.3.
Lack of domain-specific knowledge. Although the authors are familiar with network API programming, text and UML editors, they are not experts in the `Apache Mina`, `jEdit` and `ArgoUML` projects. Hence, our manual inspections might miss important insights, due to a lack of required domain-specific knowledge. We address this threat by consulting readily available additional data sources, such as the source code and bug repositories of these projects, archived mailing list discussions and project documentation.
Generalizability. As case study subjects, we picked three open source Java projects, some of which have been used before in clone detection research. Although we chose these 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 project, 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 or system [15,43,47], our findings might not generalize beyond the studied Java projects.

7. Conclusions

This paper presents an empirical study on inconsistent changes to code clones at the release level, in order to evaluate the impact of these changes on the quality of software releases. Whereas previous work on software cloning is essential for understanding the immediate effects that 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 all three studied projects (`Apache Mina`, `jEdit` and `ArgoUML`), we observe the presence of relatively long-lived, yet small, clone genealogies. We discover that only a fraction of 1.02%–4.00% of all genealogies encounter defects caused by inconsistent changes at the release level. For one particular system, i.e., `ArgoUML`, we found that the percentage of genealogies with defects at weekly level is 4 times higher than at the release level. Out of 19 bugs at the weekly level for `ArgoUML`, only 4 managed to occur in a release. This is even more remarkable when taking into account that most of the clone groups with inconsistent changes are of patterns that are meant to evolve independently, which increases the risk of losing track of the links between cloned parts in a clone group.

These numbers mean that for the three studied systems, clones in general do not have a large impact on post-release defects (quality), since (1) there are not that many defects related to clones to start with (which confirms the recent findings of Rahman et al. [39] on three additional systems), and (2) the number of clone-related defects in released code is much smaller than the total number of clone-related defects. Since the number of related defects is relatively low, it becomes very unlikely that weekly level clone-related defects slow down development. This supports our assumption that especially release level clones matter.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

Despite this good news for the developers of the subject systems, the fact that there are still clone-related defects popping up in releases means that developers cannot handle clones perfectly. Hence, our findings still emphasize the value of tools and IDE support to manage and track clone groups during development [9,10,38]. To validate the generalizability of our findings, more studies on other software systems are needed.

Acknowledgements

We would like to thank the WCRE 2009 reviewers for their valuable feedback.

References

[1] G. Antoniol, U. Villano, E. Merlo, M. Di Penta, Analyzing cloning evolution in the linux kernel, Information and Software Technology 44 (13) (2002) 755–765.
[2] L. Aversano, L. Cerulo, M. Di Penta, How clones are maintained: an empirical study, in: CSMR'07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2007, pp. 81–90.
[3] B.S. Baker, On finding duplication and near-duplication in large software systems, in: WCRE'95: Proceedings of the 2nd Working Conference on Reverse Engineering, IEEE Computer Society, 1995, p. 86.
[4] T. Bakota, R. Ferenc, T. Gyimothy, Clone smells in software evolution, in: ICSM'07: Proceedings of the 23rd IEEE International Conference on Software Maintenance, 2007, pp. 24–33.
[5] L. Barbour, H. Yuan, Y. Zou, A technique for just-in-time clone detection in large scale systems, in: ICPC'10: Proceedings of the 18th IEEE International Conference on Program Comprehension, IEEE Computer Society, 2010, pp. 76–79.
[6] I.D. Baxter, A. Yahin, L.M. de Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: ICSM'98: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1998, pp. 368–377.
[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on Software Engineering 33 (9) (2007) 577–591.
[8] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, A.E. Hassan, An empirical study on inconsistent changes to code clones at release level, in: WCRE'09: Proceedings of the 16th Working Conference on Reverse Engineering, IEEE Computer Society, 2009, pp. 85–94.
[9] M. de Wit, A. Zaidman, A. van Deursen, Managing code clones using dynamic change tracking and resolution, in: ICSM'09: Proceedings of the 25th IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 169–178.
[10] E. Duala-Ekoko, M.P. Robillard, Tracking code clones in evolving software, in: ICSE'07: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 158–167.
[11] E. Duala-Ekoko, M.P. Robillard, Clonetracker: tool support for code clone management, in: ICSE'08: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 843–846.
[12] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: ICSM'99: Proceedings of the 15th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1999, p. 109.
[13] M. Gabel, L. Jiang, Z. Su, Scalable detection of semantic clones, in: ICSE'08: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 321–330.
[14] R. Geiger, B. Fluri, H.C. Gall, M. Pinzger, Relation of code clones and change couplings, in: FASE'06: Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering, Springer, 2006, pp. 411–425.
[15] N. Göde, Evolution of type-1 clones, in: SCAM'09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE Computer Society, 2009, pp. 77–86.
[16] N. Göde, R. Koschke, Incremental clone detection, in: CSMR'09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2009, pp. 219–228.
[17] J. Harder, N. Göde, Modeling clone evolution, in: IWSC'09: Proceedings of the 4rd International Workshop on Software Clones, Kaiserlautern, Germamy, 2009, pp. 17–21.
[18] R.C. Inc., http://www.redhillconsulting.com.au/products/simian/, last accessed February 2010.
[19] L. Jiang, G. Misherghi, Z. Su, S. Glondu, Deckard: scalable and accurate tree-based detection of code clones, in: ICSE'07: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, 2007, pp. 96–105.
[20] J.H. Johnson, Substring matching for clone detection and change tracking, in: ICSM'94: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, 1994, pp. 120–126.
[21] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter? in: ICSE'09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 485–495.
[22] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670.
[23] C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, P. Weissgerber, Subjectivity in clone judgment: can we ever agree? in: Duplication, Redundancy, and Similarity in Software, in: Dagstuhl Seminar Proceedings, vol. 06301, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2007, Schloss Dagstuhl, Germany.
[24] C. Kapser, M.W. Godfrey, Cloning considered harmful considered harmful, in: WCRE'06: Proceedings of the 13th Working Conference on Reverse Engineering, IEEE Computer Society, 2006, pp. 19–28.
[25] C. Kapser, M.W. Godfrey, Supporting the analysis of clones in software systems: research articles, Journal of Software Maintenance and Evolution 18 (2) (2006) 61–82.
[26] M. Kim, V. Sazawal, D. Notkin, 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, ACM, 2005, pp. 187–196.
[27] K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, M. Bernstein, Pattern matching for clone and concept detection, Automated Software Engineering 3 (1–2) (1996) 77–108.
[28] R. Koschke, Survey of research on software clones, in: Duplication, Redundancy, and Similarity in Software, in: R. Koschke, E. Merlo, A. Walenstein (Eds.), Dagstuhl Seminar Proceedings, vol. 06301, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2007, Schloss Dagstuhl, Germany.
[29] R. Koschke, Identifying and removing software clones, in: Software Evolution, Springer, 2008, pp. 15–36.
[30] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: WCRE'06: 13th Working Conference on Reverse Engineering, IEEE Computer Society, 2006, pp. 253–262.
[31] J. Krinke, Identifying similar code with program dependence graphs, in: WCRE'01: Proceedings of the 8th Working Conference on Reverse Engineering, IEEE Computer Society, 2001, p. 301.
[32] J. Krinke, A study of consistent and inconsistent changes to code clones, in: WCRE'07: Proceedings of the 14th Working Conference on Reverse Engineering, IEEE Computer Society, 2007, pp. 170–178.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010

[33] J. Krinke, Is cloned code more stable than non-cloned code? in: SCAM'08: Proceedings of the 8th IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society, 2008, pp. 57–66.
[34] B. Lague, D. Proulx, J. Mayrand, E.M. Merlo, J. Hudepohl, Assessing the benefits of incorporating function clone detection in a development process, in: ICSM'97: Proceedings of the 13th International Conference on Software Maintenance, IEEE Computer Society, 1997, p. 314.
[35] C. Liu, C. Chen, J. Han, P.S. Yu, Gplag: detection of software plagiarism by program dependence graph analysis, in: KDD'06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2006, pp. 872–881.
[36] A. Lozano, M. Wermelinger, Assessing the effect of clones on changeability, in: ICSM'08: Proceedings of the 24th IEEE International Conference on Software Maintenance, IEEE, 2008, pp. 227–236.
[37] A. Marcus, J.I. Maletic, Identification of high-level concept clones in source code, in: ASE'01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, 2001, p. 107.
[38] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, T.N. Nguyen, Clone-aware configuration management, in: Proceedings of the 24th ACM/IEEE International Conference on Automated Software Engineering, IEEE, 2009, pp. 123–134.
[39] F. Rahman, C. Bird, P.T. Devanbu, Clones: what is that smell? in: MSR'10: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, IEEE, 2010, pp. 72–81.
[40] C.K. Roy, J.R. Cordy, A survey on software clone detection research, Queen's University, Kingston, Canada, Tech. Rep. 541, 2007.
[41] C.K. Roy, 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, IEEE Computer Society, 2008, pp. 153–162.
[42] C.K. Roy, J.R. Cordy, A mutation/injection-based automatic framework for evaluating code clone detection tools, in: ICSTW'09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, IEEE Computer Society, 2009, pp. 157–166.
[43] C.K. Roy, J.R. Cordy, Near-miss function clones in open source software: an empirical study, Journal of Software Maintenance and Evolution: Research and Practice 22 (3) (2010) 165–189.
[44] C.K. Roy, J.R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: a qualitative approach, Science of Computer Programming 74 (7) (2009) 470–495.
[45] W. Shang, B. Adams, A.E. Hassan, An experience report on scaling tools for mining software repositories using mapreduce, in: ASE'10: Proceedings of The 25th IEEE/ACM International Conference on Automated Software Engineering, IEEE/ACM, 2010, pp. 275–284.
[46] J. Sim, C.C. Wright, The kappa statistic in reliability studies: use, interpretation, and sample size requirements, Physical Therapy 85 (3) (2005) 257–268.
[47] S. Thummalapenta, L. Cerulo, L. Aversano, M.D. Penta, An empirical study on the maintenance of source code clone, Empirical Software Engineering 15 (1) (2010) 1–34.

Please cite this article in press as: N. Bettenburg, et al., An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.11.010