Empirical Software Engineering (EMSE) manuscript No.
(will be inserted by the editor)

An Empirical Study on the Maintenance of Source Code Clones

Suresh Thummalapenta · Luigi Cerulo · Lerina Aversano · Massimiliano Di Penta

the date of receipt and acceptance should be inserted later

Abstract Code cloning has been very often indicated as a bad software development practice. However, many studies appearing in the literature indicate that this is not always the case. In fact, either changes occurring in cloned code are consistently propagated, or cloning is used as a sort of templating strategy, where cloned source code fragments evolve independently.

This paper (i) proposes an automatic approach to classify the evolution of source code clone fragments, and (ii) reports a fine-grained analysis of clone evolution in four different Java and C software systems, aimed at investigating to what extent clones are consistently propagated or they evolve independently. Also, the paper investigates the relationship between the presence of clone evolution patterns and other characteristics such as clone ra-

Suresh Thummalapenta
North Carolina State University, Raleigh, USA
E-mail: sthumma@ncsu.edu

Luigi Cerulo, Lerina Aversano, Massimiliano Di Penta
Department of Engineering – University of Sannio, Benevento, Italy
E-mail: lcerulo@unisannio.it, aversano@unisannio.it, dipenta@unisannio.it

dius, clone size and the kind of change the clones underwent, i.e., corrective maintenance or enhancement.

1 Introduction

Software maintenance and evolution are crucial activities in the software development lifecycle, impacting up to 80% of the overall cost and effort [2]. In the past and in recent years, several researchers have indicated a number of factors that affect source code maintainability. A few common factors are: the lack of traceability between high-level and low-level artifacts [3, 37]; the presence of bad smells [42]; the use of inconsistent coding style [40]; and finally the presence of duplicated or similar source code fragments, known as *code clones*.

Code clones have been considered as a bad software development practice, since they can potentially cause maintainability problems: when a cloned code fragment needs to be changed, for example because of a bug fix, it might be necessary to propagate such a change across all clones. This triggered the development of different kinds of clone detection approaches and tools [6, 9, 18, 24, 25, 31, 34, 38]. To mitigate problems a clone might cause, Merlo *et al.* [8] proposed an approach for clone refactoring. However, despite the availability of promising clone refactoring approaches, developers tend not to refactor clones [15]: refactoring is a costly and above all risky and error-prone activity. Developers need to be aware of the presence of clones, without actually refactoring them.

Although there is a common understanding that cloning is a bad practice, recent studies have shown that clones are not necessarily a bad thing. As shown by Kapser and Godfrey [29], in many cases cloning has been used as a development practice, and developers are often able to handle "harmful" situations. However, to avoid problems clones can cause due

to change mis-alignment, it is necessary to provide the developers with tools able to support clone tracking (e.g., [16]). Also, empirical studies are needed to understand how developers change clones. In particular, it would be interesting to investigate whether they always propagate the changes, whether they clone the source code and then change differently each clone to realize different features, or, instead, whether developers propagate changes across clones at different times, e.g., because they are not aware of the presence of clones.

This paper proposes an approach to analyze the evolution and maintenance of source code clones, and reports results from an empirical study aimed at analyzing how clones detected in a release of four C and Java open source systems—namely ArgoUML, JBoss, OpenSSH, and PostgreSQL—undergo maintenance in following file revisions extracted from Concurrent Versioning System (CVS) or Subversion (SVN) repositories.

The analysis of clone maintenance is performed automatically using a language independent approach that, starting from a line tracing algorithm [13], identifies how a cloned code fragment evolves over time by tracing changes occurring in code snippets (referred in the paper as clone sections) composing clone fragments that belong to the clone class, i.e., to the set of (near) cloned fragments identified by a clone detection tool. In particular, the approach can distinguish cases where (i) changes are consistently and immediately propagated to all cloned fragments belonging to the same clone class; (ii) changes are consistently propagated, however, there is some delay between changes performed on different clone fragments; (iii) finally, cases where clones are not consistently changed; instead, they evolved independently, e.g., to implement different features. While previously proposed clone tracking approaches [7, 30] focus on reconstructing the clone genealogy, our approach takes into account the changes clones underwent during their lifetime, and is able to identify different evolution patterns, such as cases where an inconsistent change is only temporary, e.g., because developers forgot to correctly propagate the change, or cases where fragments be-

longing to the same clone class evolve independently. The empirical study aims at answering the following research questions:

1. how can clones be classified into the above mentioned evolution patterns;
2. whether there is any relationship between the clone granularity/size and the evolution pattern followed by the clone;
3. to the same extent, whether there is any relationship between the clone radius—i.e., the distance between clones in the code directory structure—and the clone evolution pattern; and
4. whether there is a relationship between clone evolution patterns and the occurrence of bug fixings.

Overall, results indicate that the detected clones are, in most cases, consistently changed, and when this consistent change does not happen, most of the clones underwent an independent evolution, where inconsistent changes are intentional. Only a small percentage of clones, always below 16%, underwent late change propagations. The way clones evolve does not depend on their granularity, although in some cases independent evolution—e.g., code templating—tends to occur in larger code artifacts, such as entire files. The distance between clone fragments in the code directory structure does not appear to be a cause of inconsistent changes, thus developers are able to track clones even when these clones are distributed across several directories in the source code. Finally, clone classes in which a late clone propagation was found exhibit a higher proportions of bugs than other clone classes, confirming that such a behavior—although occurring in a few cases—is potentially dangerous since that behavior can cause a bug to appear multiple times.

The paper is organized as follows. Section 2 describes the proposed clone tracing approach. Section 3 provides the definition and planning of the empirical study, while results

are reported in Section 4. Then, lessons learned are summarized in Section 5, while Section 6 discusses the empirical study's threats to validity. Section 7 relates the present work with the existing literature. Section 8 concludes the paper and outlines directions for future work.

2 Automatic clone tracing approach

This section describes our approach for automatically tracing clone changes, used to perform the empirical study presented in this paper. Given the clones detected in a given snapshot or release of a software system, the approach is able to identify whether, in subsequent file revisions, clones undergo different evolution patterns. The approach consists of three steps. First (Step 1), we extract from the CVS or SVN repository the sequence of all change sets that occurred in the time interval we want to analyze. Each change set produces a new snapshot of the system from the previous one. Then (Step 2), we use a clone detection tool to identify clones in the first snapshot of the time period of interest, and then we analyze their changes in future snapshots. This does not prevent us from applying the approach for tracing clones detected in all snapshots. In the last step (Step 3), the core of our approach, we analyze the evolution of detected clones and classify them according to the different evolution patterns.

2.1 Step 1: Extracting change sets from CVS/SVN

The analysis of clone evolution can be performed by considering the software evolution history at different levels. It is possible to consider (i) at a coarse-level, only major releases, as done by Antoniol *et al.* [4], or (ii) at a finer level—as proposed by Fischer *et al.* [17]—one can cluster together changes [19], performed by developers working on a bug fix or an

enhancement feature, into sets known as "change sets". Techniques to extract change sets consider the evolution of a software system as a sequence of *Snapshots* $(S_0, S_1, \ldots, S_m)$ generated by sequences of source code changes, also known as *Change Sets* (Δ_1, Δ_2, ..., Δ_m), representing the changes performed by a developer, for example, in terms of added, deleted, and modified source code lines. Suppose the software system is viewed as a set of n files $\{f_1, \ldots, f_n\}$, and suppose that a CVS or SVN system tracks all revisions of such files (we indicate with $f_{i,j}$ the revision j of file i). A snapshot S_k is composed of a set of file revisions, i.e., $S_k \equiv \{f_{1,j^1}, \ldots f_{n,j^n}\}$, where $j^1, \ldots, j^n$ indicate revisions of files $f_1, \ldots f_n$ in snapshot S_k; revisions can be different because each file could have been subject to a different number of changes before S_k.

In file-based versioning systems, such as CVS, a commit is the change performed on one file, and a change set—which groups changes performed on one or more files—is a sequence of file commits. Instead, in a repository-based versioning system, such as SVN, a commit could include the changes of more than one file, and similarly a change set could include more than one commit. These change sets can be detected from versioning systems using several existing approaches [12, 19, 45]. We used a time-windowing approach which considers a change set as a sequence of commits that share the same author, branch, and commit notes, and such that the difference between the time-stamps of two subsequent commits is less than or equal to 200 seconds [45].

2.2 Step 2: Detection of clone elements

This section describes how to detect clones we are interested to trace. To this aim, we identify the elements composing each clone class, i.e., clone fragments and clone section pairs. Since the purpose of our study is to analyze how clones—existing in a given release of a

software system—will be maintained in the future, we need to detect, using a clone detection tool, the clones in the source code snapshot corresponding to that release. Most of the existing clone detection tools return a set of clone classes (*CC*) and each clone class consists of a set of (near) duplicated clone fragments (*CF*), often specified by the clone detection tool in terms of file name and cloned source code line ranges.

More precisely, we define the zth clone class in a snapshot S_k as below:

$$CC_{z,k} \equiv \{CF_1^k, \ldots, CF_h^k\} \tag{1}$$

i.e., as the set of h (near)duplicated clone fragments identified in the snapshot k. Each fragment is defined as a set of source code lines of file revision $f_{i,j} \in S_k$ in the interval $[l_{start}, l_{end}]_k$. Using the *Change Sets* (Δ) information extracted in Step 1, we identify the succeeding snapshots in which at least one of the clone fragments belonging to the clone class $CC_{z,k}$ is modified. For example, consider that the clone fragment $CF_1^k \in CC_{z,k}$ belongs to the line interval $[l_{start}, l_{end}]_k$, of a source file in snapshot S_k. Suppose that the change set Δ_{k+1}, computed from S_k to S_{k+1}, indicates that some lines belonging to the interval $[l_{start},\ l_{end}]_k$ in the source file of CF_1^k have been changed. As the intersection of Δ_{k+1} and the clone fragment interval is non-empty, we identify that the clone fragment is modified in snapshot S_{k+1} and compute the new clone fragment interval $[l_{start}$, $l_{end}]_{k+1}$based on changes indicated by Δ_{k+1}. Such a process is repeated for all subsequent snapshots until the last snapshot in the chosen time period. Based on the changes indicated by Δ_{k+1}, a source code file containing a clone fragment can undergo different changes:

– Changes within the clone fragment, that can also increase/decrease its size with the addition/removal of lines;

– Changes outside the clone fragment, that, if performed above it, can move the clone fragment upward (line removal) or downward (line addition).

Often, clone fragments belonging to the same clone class may not match perfectly; in particular, they can be gapped clones, i.e., they contain lines that differ between clone fragments. In addition, as mentioned above, a clone fragment may undergo changes aimed at inserting/removing lines within it. Thus, commonly used differencing algorithms such as the Unix *diff* are not sufficient to trace clone fragment changes. To this aim, we introduce the notion of a clone section (*CS*) pair, that represents the mapping between similar elements of two clone fragments in a clone class. We denote the set of all clone section pairs between two clone fragments CF_x, CF_y—where $x, y \in [1 \dots h]$ (where h is the number of clone fragments within a clone class), see equation (1)—as $CS_{x,y} = \{CS^1_{x,y}, CS^2_{x,y}, \dots, CS^l_{x,y}\}$. Since $CS_{x,y} \equiv CS_{y,x}$, the number of possible clone section sets is $\frac{h(h-1)}{2}$. Fig. 1 shows an example taken from the ArgoUML source code with two clone fragments and four clone section pairs. Clone sections are detected by computing the difference between two clone fragments using an improved *diff* algorithm [13]. Given two file revisions, such an algorithm is able to identify added, deleted, changed, and unchanged lines, overcoming the Unix *diff* limitations. In fact, the Unix *diff* only classifies as changes cases where a set of adjacent lines undergo additions and removals in the same (row) position. Briefly, the differencing approach [13] combines the use of the Unix *diff* with text similarity measures (vector space models cosine similarity) and string similarity (Levenshtein distance) to better identify changed lines.

2.3 Step 3: Identification of Clone Evolution Patterns

This section defines the evolution patterns we consider in our work, and describes how we automatically classify clones into evolution patterns. A clone evolution pattern describes

CF_1 CrNoIncomingTransitions.java (ver. 1.1)

```
 1: package org.argouml.uml.cognitive.critics;
...: ...
12:
13: public class CrNoOutgoingTransitions extends CrUML {
14:
...: ...
30: public boolean predicate2(Object dm, Designer dsgr) {
31: if (!(dm instanceof MStateVertex)) return NO_PROBLEM;
32: MStateVertex sv = (MStateVertex) dm;
33: if (sv instanceof MState) {
34: MStateMachine sm = ((MState)sv).getStateMachine();
35: if (sm != null && sm.getTop() == sv) return NO_PROBLEM;
36: }
37: Collection outgoing = sv.getOutgoings();
38: boolean needsOutgoing = outgoing == null || outgoing.size() == 0;
39: if (sv instanceof MFinalState) {
40: needsOutgoing = false;
41: }
42: if (needsOutgoing) return PROBLEM_FOUND;
43: return NO_PROBLEM;
44: }
45:
46: } /* end class CrNoOutgoingTransitions */
```

$CS^1_{1,2}$ $CS^2_{1,2}$ $CS^3_{1,2}$ $CS^4_{1,2}$

CF_2 CrNoOutgoingTransitions.java (ver. 1.1)

```
 1: package org.argouml.uml.cognitive.critics;
...: ...
12:
13: public class CrNoIncomingTransitions extends CrUML {
14:
...: ...
30: public boolean predicate2(Object dm, Designer dsgr) {
31: if (!(dm instanceof MStateVertex)) return NO_PROBLEM;
32: MStateVertex sv = (MStateVertex) dm;
33: if (sv instanceof MState) {
34: MStateMachine sm = ((MState)sv).getStateMachine();
35: if (sm != null && sm.getTop() == sv) return NO_PROBLEM;
36: }
37: //Vector outgoing = sv.getOutgoing();
38: Collection incoming = sv.getIncomings();
39: //boolean needsOutgoing = outgoing == null || outgoing.size() == 0;
40: boolean needsIncoming = incoming == null || incoming.size() == 0;
41: if (sv instanceof MPseudostate) {
42: MPseudostateKind k = ((MPseudostate)sv).getKind();
43: if (k.equals(MPseudostateKind.INITIAL)) needsIncoming = false;
44: //if (k.equals(MPseudostateKind.FINAL)) needsOutgoing = false;
45: }
46: // if (needsIncoming && !needsOutgoing) return PROBLEM_FOUND;
47: if (needsIncoming) return PROBLEM_FOUND;
48: return NO_PROBLEM;
49: }
50:
51: } /* end class CrNoIncomingTransitions */
```

Fig. 1 Clone sections as a way to map lines between two fragments. An example coming from ArgoUML.

how a clone class, i.e., a set of source code artifacts classified as clones, is affected by maintenance activities. The patterns are defined for clone classes instead of individual clone fragments, since one of the aims of this work is to analyze how/whether maintainers update—or not—clone fragments belonging to a given class. Let us consider a clone class $CC_{z,k}$ identified in the snapshot S_k. When a clone fragment $CF_x \in CC_{z,k}$ changes differently from other fragments of the same clone class (or has been removed), it cannot be considered anymore as a clone belonging to that clone class $CF_x \notin CC_{z,k+1}$. When, between S_k and S_{k+1}, at least one clone fragment belonging to $CC_{z,k}$ changes, the entire clone class can evolve according to one of the following evolution patterns:

– *COnsistent Evolution (CO):* all clone fragments in $CC_{z,k}$ are either consistently changed—i.e., they do not differ more than a fixed threshold (see Fig. 2-a)—or some of them disappear because, according to the differencing algorithm [13], all clone fragment lines are removed. This evolution pattern typically occurs when cloning is used to implement similar features; when one feature is changed, the same change has to be propagated to all other similar features.

– *Late Propagation (LP):* it occurs within a clone class, if a clone fragment undergoes a change at snapshot S_{k+1}, and at least one another clone fragment undergoes a similar

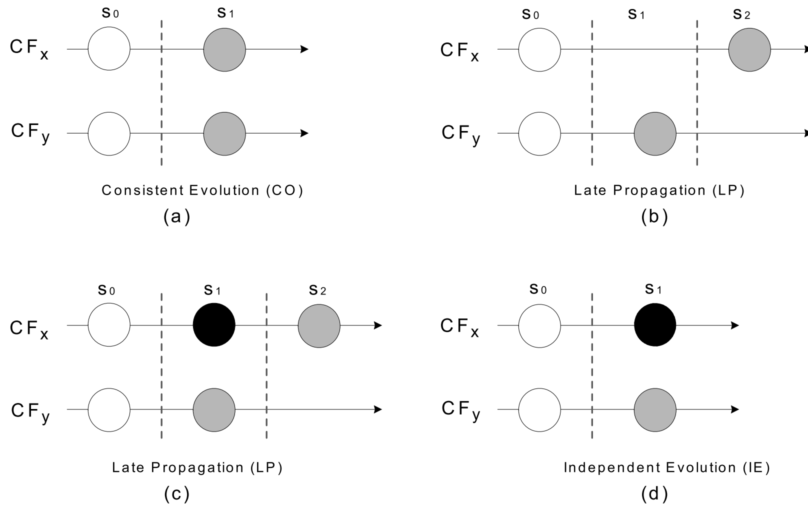


Fig. 2 Clone evolution patterns for two clone fragments CF_x and CF_y.

(i.e., with a difference below a threshold) change at snapshot $S_{k+1+\delta}$ ($\delta > 0$), within the time interval T we are observing (see Fig. 2-b). Or, clone fragments undergo different changes at S_{k+1}, and then, at snapshot $S_{k+1+\delta}$ ($\delta > 0$), are realigned by other changes (see Fig. 2-c). The realignment is detected by means of the differencing tool, and considering the same similarity threshold used to consider clone fragments as different.

– *Delayed Propagation* (hereby indicated with *L2*): is a particular case of Late Propagation where changes between two clone fragments, although not propagated within the same snapshot, are propagated within 24 hours (the time interval computed between the last commit of S_k and the first commit of $S_{k+\delta+1}$). This is representative of situations where developers are not able to commit all files within the same snapshot, however they are able to do it within the same day.
– *Independent Evolution (IE):* happens when two or more clone fragments in $CC_{z,k}$ undergo different changes—i.e., changes that differ more than a fixed threshold—between snapshots S_k and S_{k+1} (see Fig. 2-d) and makes such clone fragments no more belong-

ing to $CC_{z,k+1}$ in the observed time period. IE can occur when programmers clone code fragments and then adapt them to implement new pieces of functionality. However, IE can also occur when, for some reason, a clone change is propagated outside our interval of observation. For this reason, although in the interval of observation considered IE and LP are mutually exclusive, it cannot be guaranteed that, outside such an interval, all clone fragments belonging to the clone class become consistent again, thus resulting in a late propagation. As shown later in the case study we experienced that the interval of observation considered is always much greater than the time for late propagations, then the latter case occurs very rarely.

Concrete examples of the above evolution patterns, detected on the four systems considered in our empirical study, are discussed in Section 4.5. The identification of clone evolution patterns is performed through a sequence of three steps, aimed at analyzing the evolution of clone elements at increasing levels of granularity, i.e., (i) clone section pair evolution, (ii) clone fragment pairs evolution, and (iii) clone class evolution. With the proposed approach, one can study the evolution at the desired level or granularity, although one is often interested—as we do in our empirical study—to investigate how a whole clone class evolves. Fig. 3 shows these steps in the case of a clone class composed of three clone fragments CF_1, CF_2, and CF_3. Our differencing approach [13] identifies clone section pairs by matching similar code elements between two clone fragments. First, as it will be discussed in Section 2.3.1, the approach classifies the evolution of each clone section pair. Then, considering the evolution patterns for all the clone sections belonging to the same pair of clone fragments, the approach identifies—as explained in Section 2.3.2—the evolution pattern for each possible pair of clone fragments belonging to a clone class. Finally, this information is

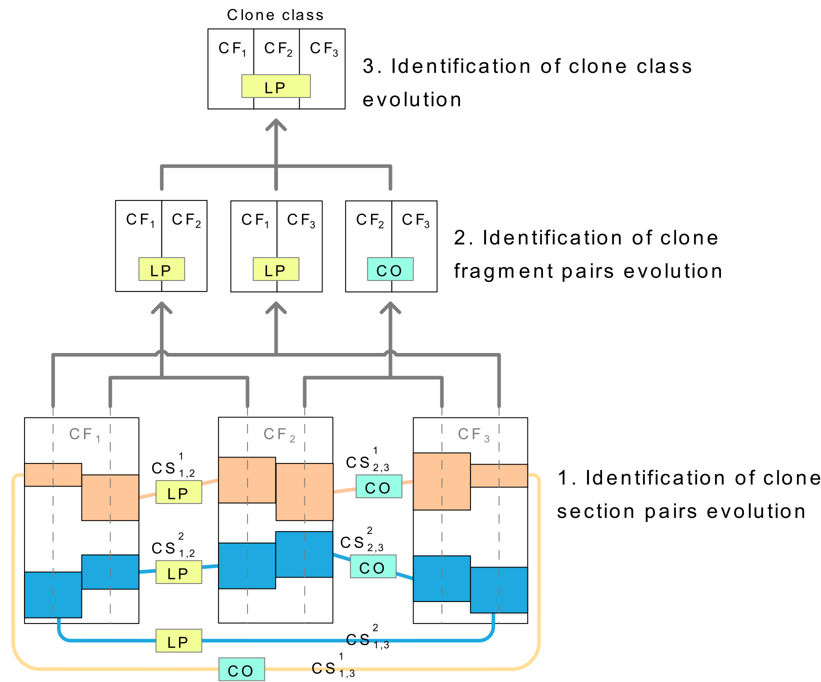


Fig. 3 Clone evolution patterns considering clone elements at different levels of granularity.

used—as explained in Section 2.3.3—to determine the evolution pattern of the whole clone class.

2.3.1 Identification of Clone Section Pairs Evolution

To detect the evolution pattern of a clone section, $CS^{\eta}_{x,y}$ in a snapshot S_k, we consider the succeeding snapshots where at least one clone fragment pair, CF_x and CF_y, has been changed. In such snapshots we compute the similarity between both elements of the clone section pair, one belonging to the first clone fragment, CF_x, and the other belonging to the second clone fragment, CF_y. The similarity is computed by using a Levenshtein edit distance [33] (LD) applied to the sequence of tokens—belonging to the clone section elements—extracted from the source files by means of a lexer (thus ignoring changes on comments,

identifiers and literals). The Levenshtein edit distance between two strings s_1 and s_2 measures the number of editing operations needed to transform s_1 into s_2. The higher the LD is, the more the two strings differ. To allow for comparisons (as the LD is a distance and not a similarity measure) we used the *Normalized Levenshtein Distance* (*NLD*), which ranges in the interval $[0, 1]$, where 1 means that lines match, while 0 means that lines are strictly different. For two non-empty strings, s_1 and s_2, *NLD* is defined as:

$$NLD(s_1, s_2) = 1 - \frac{LD(s_1, s_2)}{max(s_1, s_2)} \quad (2)$$

where $LD(s_1, s_2)$ is the *Levenshtein Distance*, and $max(s_1, s_2)$ is the length of the longest string.

To classify whether a clone section pair underwent a consistent evolution or an independent evolution, we compute the NLD between its two elements represented as a sequence of tokens. When a clone section pair underwent a change, two thresholds for *NLD*, namely $NLDT_{low}$ and $NLDT_{high}$ ($NLDT_{low} < NLDT_{high}$), are used to distinguish among different states:

- *Consistent (C)* if $NLD(s_1, s_2) \geq NLDT_{high}$;
- *Inconsistent (I)* if $NLD(s_1, s_2) \leq NLDT_{low}$;
- *“Unknown” (UN)*, i.e., no classification is possible if $NLDT_{low} < NLD(s_1, s_2) < NLDT_{high}$. We introduced the “Unknown” category as we observed—by means of a manual inspection when calibrating the thresholds—that the clone section pairs with *NLD* between $NLDT_{low}$ and $NLDT_{high}$ are more prone to be classified into an incorrect evolution category;
- *Removal(R)*, if one of the two clone section elements disappeared.

After having determined—for all snapshots where the clone section $CS^{\eta}_{x,y}$ changed—whether the clone section elements are in state C or I, we analyze the sequence of C and I and classify them according to the following rules:

- The sequence is classified as UN (Unknown) if it contains at least one UN state.
- If the sequence only contains C, then we can say that, during the period we observed, the clone section pair underwent a consistent evolution (CO). For example, the sequence of states C, C, C, C is classified as CO.
- If the sequence starts with a sequence of C and ends with a sequence of I, then the clone section underwent an independent evolution (IE). For example, the sequence of states C, C, I, I is classified as IE.
- If the sequence contains at least one transition from C to I, and then from I to C again, terminating with C, then the clone section pair underwent a late propagation, which is L2 if the time interval between the IC and the new CC is less than or equal to 24 hours, otherwise it is LP. For example, the sequence of states C, C, I, C is classified as LP (or L2).
- If the sequence terminates with the removal of one clone section element R, then the clone section pair evolution pattern is the one considered before its removal. For example, the sequence of states C, C, C, R is classified as CO.

2.3.2 Identification of Clone Fragment Pairs Evolution

The evolution pattern of a clone fragment pair is obtained by considering the evolution patterns of its clone section pairs by using the following rules, applied in the given order:

1. UN, if at least the evolution of one clone section pair is classified as UN;
2. LP, if at least one clone section pair evolves as LP;

3. L2, if at least one clone section pair evolves as L2;
4. If a number of clone section pairs—representing at least 40% of the source code lines composing the clone section pairs belonging to the fragment—evolves as CO, we can say that the clone fragment pair evolves as CO, otherwise it evolves as IE. The value of 40%, hereby indicated as the Consistency Threshold (*CT*), has been empirically obtained by comparing the precision obtained with different threshold values (see Section 3.4). With this threshold we classify as consistent a change where at most 60% of the fragment lines change differently. In the manual classification described in Section 3.4, we found that higher threshold values tend to lower both precision and recall of consistent and inconsistent change classification, consequently decreasing the performance of evolution pattern classification as well. On the other hand, values lover than 40% would lead to classify as consistent changes that differ for a large portion ($> 60\%$) of the fragment lines, thus lowering the precision of consistent change classification.

The *LP* pattern has the highest priority over all other patterns, i.e., a late propagation occurring in a single clone section pair causes the whole clone fragment pair to be classified as *LP*. This does not mean, however, that the fragment is completely realigned. On the other hand, this allows for capturing cases of partial late propagations, i.e., occurring in some clone sections only. We made this choice to highlight all cases where there was a delayed re-alignment of a clone section pair. For our study, *LP* constitutes the most interesting (and potentially harmful) pattern we are interested in observing, and we want to detect *LP* even when it is partial. Different is the case where we would like to observe whether the whole clone class becomes consistent again. In such a case the clone class, after the change propagation, shall not contain any inconsistent clone section pairs.

Let us consider a clone fragment pair, CF_x and CF_y, composed of three clone section pairs, $CS^1_{x,y}$ (60% of the lines), $CS^2_{x,y}$ (10% of the lines), and $CS^3_{x,y}$ (30% of the lines) with the following sequences of clone section states in the interval of observation ranging from snapshot S_1 to snapshot S_5:

	S_1	S_2	S_3	S_4	S_5	
$CS^1_{x,y}$	*C*	*C*	*I*	*C*	*R*	$\rightarrow$ *LP*
$CS^2_{x,y}$	*C*	*C*	*C*	*C*	*C*	$\rightarrow$ *CO*
$CS^3_{x,y}$	*C*	*I*	*I*	*I*	*I*	$\rightarrow$ *IE*

The evolution pattern for the clone fragment pair CF_x and CF_y in the same interval of observation (S_1, S_5) is *LP* as it predominates over all.

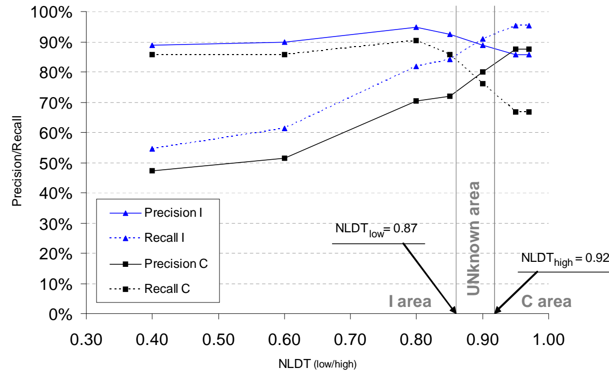
Instead, considering different intervals of observation the classification could be different, the following is what happens:

1. Interval S_1, S_2: CS^1 and CS^2 (70% of the lines) evolve as CO and CS^3 evolves as IE, then CF_x and CF_y evolve as *CO*;
2. Interval S_1, S_3: CS^1 and CS^3 (90% of the lines) evolve as IE, then CF_x and CF_y evolve as *IE*;
3. Interval S_1, S_4: CS^1 evolves as LP, then CF_x and CF_y evolve as *LP*.

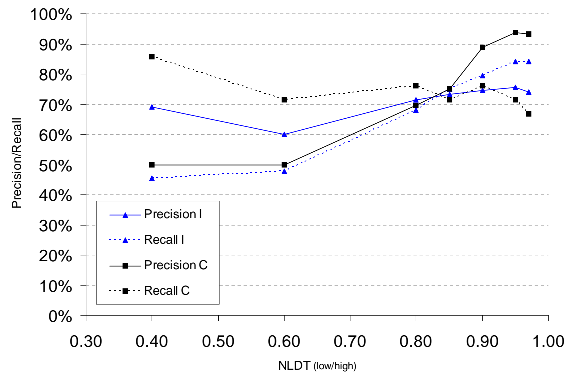
2.3.3 Identification of Clone Class Evolution

The evolution pattern of a clone class is obtained by analyzing the evolution patterns of all of its clone fragment pairs and considering the most dominant pattern, according to the following priority chain, for which the motivation is the same as for clone fragments (the first one in the chain has the highest priority):

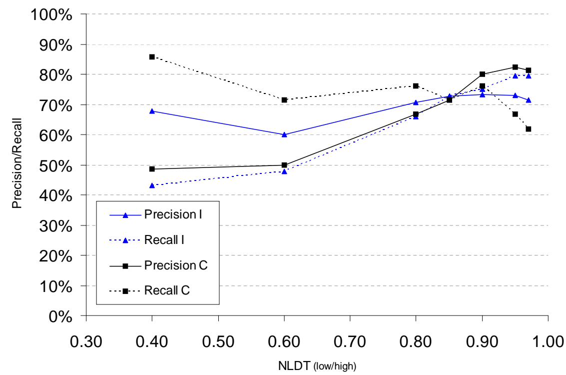
LP $\rightarrow$ *L2* $\rightarrow$ *IE* $\rightarrow$ *CO*



(a) CT=0.4



(b) CT=0.6



(c) CT=0.8

Fig. 4 Precision and recall curves for different CT values.

2.3.4 Algorithm parameter calibration

To calibrate the $NLDT_{low}$, $NLDT_{high}$, and CT thresholds, we perform an empirical assessment of precision and recall with the results of an earlier manual analysis performed on ArgoUML. Such a manual analysis [5] was performed on all clone fragments belonging to each clone class (detected with Simscan), and for each snapshot where at least a fragment changed. The inspectors used utilities, such as *diff* and the *Emacs* editor, to see whether in subsequent snapshots clone fragments evolved independently or if a late propagation occurred. Change log messages were also inspected to analyze the change rationale. The analysis was supported by a checklist asking the inspector to indicate:

1. whether the clone is a false positive or a true clone;
2. whether the clone fragment had a consistent (C) or inconsistent (I) change. Then, the inspector had to manually analyze the sequence of such states to identify the evolution pattern (*CO*, *IE*, or *LP*);
3. whether the maintenance was due to a bug fixing;
4. whether a new bug appeared because a bug fixing was not propagated to all cloned code;
5. in case the change is not propagated within a single change set, whether the same change was propagated in later change sets; and
6. a brief description of the change rationale.

The threshold calibration is performed by computing precision and recall for C and I, considering different values of the thresholds $NLDT_{low}$ and $NLDT_{high}$, and considering three different values for CT: 0.4, 0.6, and 0.8. Fig. 4 shows, for different values of CT, how the classification precision and recall vary with $NLDT_{low}$ and $NLDT_{high}$. The figure shows how precision and recall for I vary with different $NLDT_{low}$ values, and how precision and recall for CO vary with different $NLDT_{high}$ values. The best tradeoff between precision and

recall is achieved by choosing $CT = 0.40$ and for $NLDT \sim 0.89$. We choose $NLDT_{low}$ = 0.87 and the lower $NLDT_{high}$ = 0.92 as a tradeoff to achieve a good precision and a limited number of clone classes classified as *UN*. Increasing the range $[NLDT_{low}, NLDT_{high}]$ increases the precision of the *C* and *I* classification; however, it also increases the number of clone sections classified as *UN*, and vice versa when decreasing the range. It can be noticed that, when increasing *CT*, the intersection between precision and recall curves tend to occur in correspondence of lower values. No improvements were found for $CT < 0.40$.

3 Empirical Study Definition

The *object* of this empirical study is *to analyze* the evolution of software clones with the *purpose* of investigating how changes that occurred in cloned code fragments are propagated through all clones. The *quality focus* is the consistency of change propagation on source code clones, where a lack of consistency could, in some cases, be the cause of an increased fault-proneness. The *perspective* is of researchers, who want to investigate the effects of clone maintenance, and of project managers that need to monitor how changes on cloned code are propagated. The *context* consists of four open source projects of different sizes and developed with different programming languages (Java and C). The remainder of this section reports detailed characteristics of context, research questions, empirical study settings and analysis method.

3.1 Context

We select a set of four open source projects which differ in size, programming language, and application purpose. For each system, we choose a time period and detect clones in the first snapshot of the chosen time period. Then, changes occurring on such clones are

Table 1 Case study history characteristics.

System	Start	Language	Releases	Snapshots analyzed	CVS Users	KNLOC (min–max)	Files (min–max)
ArgoUML	Sep, 2000	Java	58	5,524	32	99.5–159.5	446–2,381
JBoss	Apr, 2000	Java	27	28,474	267	363.1–2601.8	3410–25,143
OpenSSH	Jan, 2000	Ansi C	33	1,314	49	15.5–47.3	75–170
PostgreSQL	Jul, 1996	Ansi C	119	9,323	27	121.2–497.9	630–2,530

analyzed with the approach described in Section 2 until the last snapshot of the chosen time period. The snapshot on which clones are detected is selected earlier in the system history in order to have enough snapshots on which to observe the evolution, but, at the same time, corresponding to a stable release where the system reached a stable size and level of functionality provided (i.e., after the earlier alpha and beta releases). Since we choose an early snapshot in the software system lifetime (as recorded in the CVS), we limited the variability in clone ages. It was also checked, from commit notes, that the snapshot did not occur right after a major clone refactoring activity.

Although the choice of considering clones from a single snapshot limits the number of clones considered in the study, and does not consider new clone fragments appearing in the same clone classes, it allows for considering the same period of observation for all clones detected in the same system. Different studies aimed at analyzing clone genealogy (e.g., Kim *et al.* [30]) occurring in any system snapshot. The above choice does not constitute a limitation for our approach, which can be used to build clone genealogies as well. However, building genealogies is not the purpose of our study. Instead, we are primarily interested in identifying the evolution patterns of a set of clones detected in the software system (in a release) and relating the evolution pattern with other variables (clone radius, clone granularity, cloned code fault-proneness). By excluding new clone classes (or just new clone fragments)

occurring in future snapshots, we limit the age difference of clones we are considering in the study.

Table 1 shows the history characteristics for each project, in particular: starting date, programming language, number of released versions, number of analyzed snapshots, number of CVS committers, KNLOCs (non-commented KLOCs), and number of source code files in the repository. ArgoUML[1] is an open source UML modeling tool with advanced software design features, such as reverse/refactoring engineering and code generation. The project started in September 2000. A total of $5,525$ snapshots have been extracted, and clone detection has been performed on the first snapshot that corresponds to release $0.9.0$. Clone evolution was studied on the remaining $5,524$ snapshots. JBoss[2] is a J2EE compliant application server. The project started in April 2000. The number of snapshots extracted is $37,577$, and clone detection has been performed on snapshot $9,083$ which corresponds to release $3.2.2$. Clone evolution was studied on the remaining $28,474$ snapshots. OpenSSH[3] is a free version of the SSH connectivity tools implemented in ANSI C and used to provide a secure channel between network connections. The project started in January 2000 and quickly reached a good level of maturity. The total number of snapshots extracted from the CVS repository is $2,548$, and clone detection has been performed on snapshot $1,234$ which corresponds to release $2.9.9$. Clone evolution was studied on the remaining $1,314$ snapshots. PostgreSQL[4] is an open source relational database system implemented in ANSI C and running on a wide number of operating systems. The project started in July 1996. The number of snapshots extracted is $12,577$, and clone detection has been performed on snap-

[1] *http://argouml.tigris.org*
[2] *http://www.jboss.org*
[3] *http://www.openssh.com*
[4] *http://www.postgresql.org*

shot $3,254$, which corresponds to release 6.5. Clone evolution was studied on the remaining $9,323$ snapshots.

3.2 Research Questions

In this section we formulate four research questions—concerning different aspects of clone evolution—this study aims to answer:

- **RQ1:** *What is the percentage of clones following different evolution patterns?* This question aims at investigating how the evolution of different clone instances fall into evolution patterns defined in Section 2.3. This research question is useful to understand what percentage of clones evolve independently, likely because developers used cloning "for templating and evolving" purposes? What percentage evolve consistently, i.e., a change is needed on all clone fragments belonging to the same clone class and developers take care of promptly propagating the change? Finally, what percentage undergo late propagation probably because developers were not able (or neglected) to promptly perform the change on all clone fragments of the clone class. Such a classification is useful to understand where cloning represents a common development practice and is thus properly handled by developers, and where—in the case of late propagation—cloning can trigger potentially dangerous situations. Since we are interested in the clone class maintenance rather than in its genealogy, for the clone classes that disappear, we consider the evolution pattern in the period the clone class existed, i.e., before it disappeared.
- **RQ2:** *Is there any relationship between the clone granularity/size and the evolution pattern followed by the clone?* This research question investigates whether clones having a different granularity (cloned classes, methods or code fragments) undergo different evolution patterns. This research question has the purpose of understanding whether, for

example, small clones (e.g., blocks) are more prone to late propagations, since they are more difficult to be monitored, or source code templating and independent evolution occur at the method level or for entire classes. It should be noted that the granularity of clones (class/file, method, block) does not have to be confounded with the granularity of clone elements (clone class, clone fragments and clone section) introduced in Section 2.3.

- **RQ3:** *Is there a relationship between the clone radius and the clone evolution pattern?* This question investigates whether late propagations happen more frequently when the distance between clone fragments, in terms of the clone radius defined by Kamiya *et al.* [41], is high. They define the clone radius as follows *"For a given clone class C, let F be a set of files which include each code fragment of C. Define RAD(C) as the maximum length of the paths from each file F to the lowest common ancestor directory of all files in F."*. As Kamiya *et al.* claim, if the clone radius is very high, it will be more difficult to modify faults in the clone class, e.g., because the developer is not able to find all clone fragments, thus leading to potential inconsistent changes.
- **RQ4:** *Is there any relationship between clone evolution patterns and the occurrence of bug fixings?* This question investigates whether the proportion of bug fixing activity changes across evolution patterns. One could expect late propagations to increase the cloned code fault-proneness, because when developers forget to propagate bug fixing changes to all clone fragments of a class, the same bug will occur again in the future.

3.3 Analysis Method

To analyze the evolution of software clones, we choose a stable release of each selected project and detect clones using two different approaches: token-based and AST-based. The

Table 2 Clone statistics for each system.

System Release	Snapshot	Token clone classes	Token clone fragments (min–max, avg)	Token % of cloned lines	AST clone classes	AST clone fragments (min–max, avg)	AST % of cloned lines
ArgoUML 0.9	1	242	2–114, 5.40	21.02%	280	2–30, 4.02	20.95%
JBoss 3.2.2	9,083	601	2–177, 5.63	10.70%	1,095	2–45, 3.16	20.01%
OpenSSH 2.9.9	1,234	22	2–26, 4.45	4.54%	100	2–80, 6.75	23.95%
PostgreSQL 6.5	3,254	54	2–75, 4.80	2.74%	289	2–67, 4.24	3.00%

token-based clone detection approach transforms input source text into tokens and detects clones through a token-by-token comparison. In our study, we use CCFinder developed by Kamiya *et al.* [25], a very well-known token-based clone detection tool. It supports multiple languages like Java, C, and C++. The AST-based approach [9] builds an AST for the input source files and compares the hash value of subtrees for detecting clones. To apply this approach, we rely on the two clone detectors SimScan[5] and Bauhaus *ccdiml*[6] for Java and C source code files, respectively.

Table 2 summarizes the results of our clone detection process. In particular, the table shows the number of clones detected by the clone detector as well as the minimum, average, and maximum number of clone fragments in all detected clone classes, and the overall percentage of cloned source code lines.

To address **RQ4**, it is necessary to analyze the changes made on the source code of each clone class, and determine whether the changes were due to bug fixing or not. Each change corresponds to a commit in the CVS/SVN repository. To classify the changes, the literature reports heuristics that classify as bugs all changes for which the log message either contains some specific keywords (e.g., ‘Fixed’, ‘Bug’, or ‘Issue’) or it contains a bug report ID (e.g.,

[5] *http://www.blue-edge.bg/simscan/*

[6] *http://www.bauhaus-stuttgart.de/clones/*

'#12345'). Further details can be found in [20,39]. We assume that if any clone fragment belonging to a clone class undergoes a change, then the change, which can be a bug fixing, affects the clone class.

3.4 Study Settings

An important setting for our study is the calibration of the clone-detection tools. While token-based clone detection approaches ensure a high recall with a low precision, AST-based approaches ensure a high precision with a low recall [10]. In this context, we are interested in achieving, in both cases, a high precision. Although we are aware that by lowering the recall, many data points will be missed in our study, we need to build our claims on clone evolution on a set of samples having the smallest error (in terms of false positives) possible, without the need for performing a manual inspection, which would have lead to a subjective evaluation [26]. The following settings are used for each clone detection tool:

- *CCFinder* filters out clones smaller than 60 tokens. The *ccreformer* tool is used with the options to remove non-longest and overlapping clone pairs. For replication purposes, it is used with the following command line parameters: *-b 60 -cg -cf*.
- *Simscan* is set up to consider a *low volume* threshold, a *medium* similarity and an *average* execution speed. For replication purposes, it is used with the following command line parameters: *-volume 1 -similarity 2 -speed 3*.
- *Bauhaus ccdiml* is set up to consider all statements in the source code and at least 10 source code lines for comparison. For replication purposes, it is used with the following command line parameters: *-all_statements -minlines 10*.

As explained in Sections 2.3.1 and 2.3.2, the automated classification approach relies on the thresholds $NLDT_{low} = 0.87$ and $NLDT_{high} = 0.92$, and $CT = 0.40$.

3.4.1 Statistical Analyses

Different kinds of statistical analyses are used to address the research questions formulated in Section 3.2. To test the presence of a significant difference among proportions of different clone evolution patterns identified on clones detected by different clone detection tools (**RQ1**), the Pearson Chi-Square test is used. It tests differences in proportions for dichotomic data in contingency tables, with number of rows or columns greater than two. Much in the same way, the Chi-Square test is used to test the different proportions of evolution patterns for different granularity levels (**RQ2**). To test whether clone classes with different evolution patterns have different average clone radius (**RQ3**), we use the Kruskal-Wallis test. Such a non-parametric test can be used on ordinal data—like the clone radius—to test the difference between more than two medians. To test whether or not the proportion of bugs change across clone evolution patterns (**RQ4**), we perform a proportion test. All tests were performed assuming a significance level of 95%.

4 Study Results

This section reports results of the empirical study defined in Section 3. For replication purposes, raw data used for our analyses is available for download[7].

4.1 RQ1: How can clones be classified into evolution patterns?

Table 3 and Fig. 5 show, for each system and for different clone detection approaches (Token-based and AST-based), the number of clone classes that follow different evolution patterns. UN indicates the number of clone classes the approach was not able to classify automatically, as explained in Section 2.3.1.

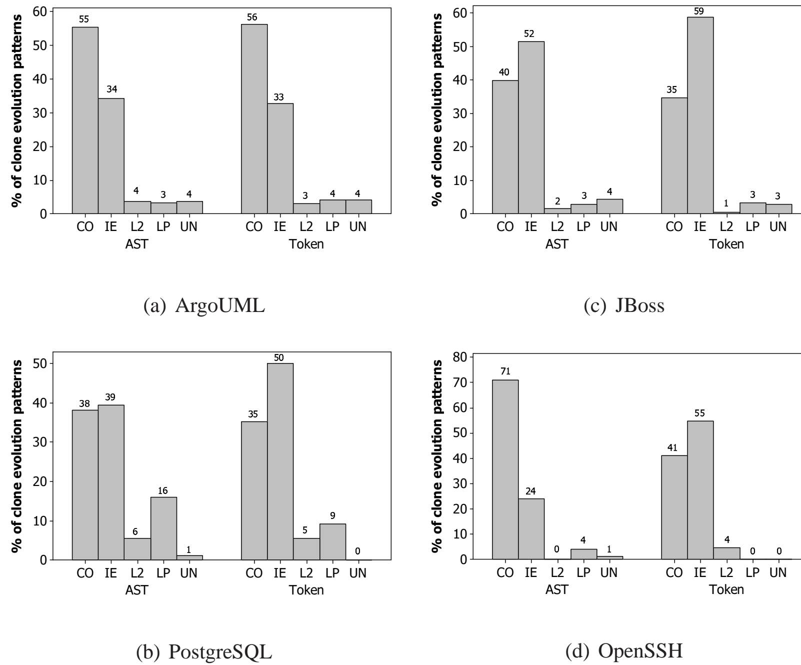
[7] *http://www.rcost.unisannio.it/mdipenta/clone-evol-rawdata.zip*

Table 3 Number of evolution patterns detected on different systems with different clone detection approaches.

System	AST-based					
	CO	**IE**	**L2**	**LP**	**UN**	**All**
ArgoUML 0.9	155	96	10	9	10	280
JBoss 3.2.2	436	564	17	31	53	1101
OpenSSH 2.9.9	71	24	0	4	1	100
PostgreSQL 6.5	110	114	16	46	3	289
System	**Token-based**					
	CO	**IE**	**L2**	**LP**	**UN**	**All**
ArgoUML 0.9	136	79	7	10	10	242
JBoss 3.2.2	209	353	3	19	17	601
OpenSSH 2.9.9	9	12	1	0	0	22
PostgreSQL 6.5	19	27	3	5	0	54

For all systems, over 80% of the clones undergo IE or CO evolution patterns, but for PostgreSQL the cumulative percentage is slightly below 80%. In two cases (ArgoUML and OpenSSH) CO is the most frequent, while in the other two cases (JBoss and PostgreSQL) IE is the most frequent one. This could indicate that either clones tend to be consistently updated, or they underwent independent evolutions, because developers adopted cloning as a development strategy, i.e., they templated code fragments and then evolved them independently to build different pieces of functionality. It is possible to say that, in most cases, clones were consistently updated or evolved independently. Late propagations are not very frequent, almost always below 5% (see Fig. 5) except for PostgreSQL where they reach the 16% for AST-based clone detection.

We also analyze whether, for different systems, the proportion of clone evolution patterns varies between different clone detection approaches. To this aim, we perform a Pearson Chi-Square test, building contingency tables where columns represent evolution patterns



(a) ArgoUML

(c) JBoss

(b) PostgreSQL

(d) OpenSSH

Fig. 5 Clone evolution patterns among different clone detectors (figures on bars indicate percentages).

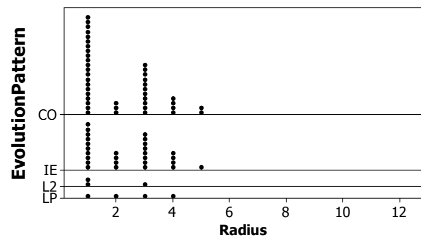
(IE, CO, L2, LP) and rows represent the clone detection approaches (H_0: the proportion of clone classes for different evolution patterns does not differ for the two clone detection approaches). We ignore L2 and LP patterns in all cases where the number of instances for these classes is smaller than 5, a number not sufficient to allow for applying the Chi-Square test. For ArgoUML, the difference of proportions is not significant (p-value = 0.950). For JBoss, the test indicates a significant difference for all the evolution patterns (p-value = 0.015). For OpenSSH, we obtain a significant difference between CO and IE evolution patterns (p-value = 0.004). No significant difference is found for PostgreSQL, considering CO, IE, and LP evolution patterns (p-value = 0.264). Overall, we cannot reject the above stated null hypothesis.

Table 4 Evolution patterns for different levels of granularity.

System	AST-based														
	Class/File					Method/Function					Block				
	CO	**IE**	**L2**	**LP**	**All**	**CO**	**IE**	**L2**	**LP**	**All**	**CO**	**IE**	**L2**	**LP**	**All**
ArgoUML 0.9	47	8	3	1	59	86	68	7	6	167	22	20	0	2	44
JBoss 3.2.2	117	150	1	1	269	220	243	7	20	490	99	171	9	10	289
OpenSSH 2.9.9	0	0	0	0	0	1	1	0	0	2	70	23	0	4	97
PostgreSQL 6.5	0	0	0	0	0	4	9	1	3	17	106	105	15	43	269
System	Token-based														
	Class/File					Method/Function					Block				
	CO	**IE**	**L2**	**LP**	**All**	**CO**	**IE**	**L2**	**LP**	**All**	**CO**	**IE**	**L2**	**LP**	**All**
ArgoUML 0.9	6	9	0	2	17	28	13	1	1	43	102	57	6	7	172
JBoss 3.2.2	5	22	0	1	28	59	171	0	5	235	145	160	3	13	321
OpenSSH 2.9.9	0	0	0	0	0	0	0	0	0	0	9	12	1	0	22
PostgreSQL 6.5	1	1	0	0	2	5	5	1	2	13	13	21	2	3	39

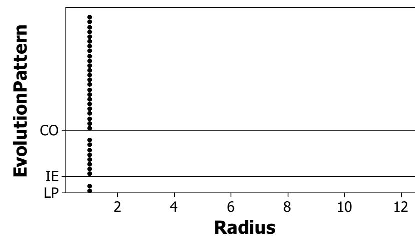
4.2 RQ2: Is there any relationship between the clone granularity and the evolution pattern followed by the clone?

Table 4 shows, for each system, the number of evolution patterns detected for different levels of granularity, i.e., (i) block, (ii) method (function for the C language), and (iii) class (file for the C language). We perform a Pearson Chi-Square test on a contingency table, where columns represent evolution patterns and rows granularity levels (H_0: the proportion of clones having different granularity levels does not change among evolution patterns). For clones detected with the token-based approach, there is a significant difference among granularity levels only for JBoss (p-value < 0.001). In this case, we only consider CO and IE evolution patterns, as all the others have a number of instances less than 5 (thus the Chi-Square test is not applicable). It can be noted that IE is more frequent than CO for class/file



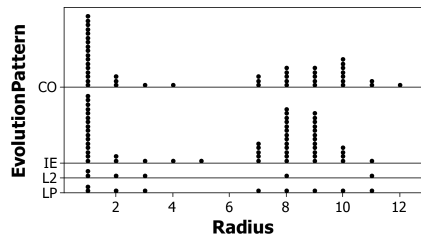
Each symbol represents up to 4 observations.

(a) ArgoUML



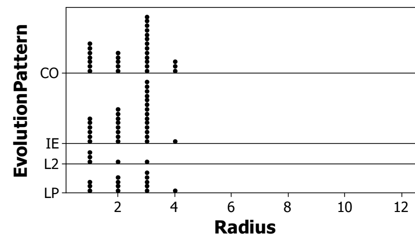
Each symbol represents up to 3 observations.

(c) OpenSSH



Each symbol represents up to 11 observations.

(b) JBoss



Each symbol represents up to 4 observations.

(d) PostgreSQL

Fig. 6 Dot plots of clone classes for different average clone radius (Note: dots indicate different number of classes for different systems).

and method/function clones, while for block-sized clones such a difference is smaller. For ArgoUML, no significant relationship is detected (p-value = 0.14), while for OpenSSH and PostgreSQL the Chi-Square test is not applicable. Similarly, for clones detected with the AST-based approach, we can observe a significant difference among granularity levels for both ArgoUML and JBoss (p-value = 0.003 and 0.001 respectively), considering all the evolution patterns. The test is also repeated excluding the L2 and LP evolution patterns, which have a number of instances smaller than 5. Also in this case, the differences are significant (p-value < 0.001 and 0.017 respectively). For AST-based clones, it is not possible to apply the Chi-Square test for OpenSSH and PostgreSQL. Overall, for *RQ2* we cannot reject the null hypothesis.

4.3 RQ3: Is there a relationship between the clone radius and the clone evolution pattern?

To investigate the presence of a relationship between the clone evolution patterns and the clone radius, we perform a Kruskal-Wallis test (H_0: the median radius does not differ among evolution patterns). In the following, we report p-values adjusted for ties. For ArgoUML, the test indicates that the median radius for different patterns is not significantly different (p-value=0.08 for AST clones and 0.19 for token-based clones). For JBoss, different kinds of clones exhibit different behaviors. In fact, while the difference is significant for AST-based clones (p-value=0.001), it is not significant for token-based clones (p-value=0.87). Differences are not significant for OpenSSH (p-value=1 in both cases). This may be because for OpenSSH only clones within the same source file (i.e., radius=1) are found. Finally, results for PostgreSQL indicate significant differences for both AST-based clones and token-based clones (p-value=0.004 and 0.034 respectively).

Fig. 6 provides an overview, using a dot plot notation, of the number of clone classes for different radii. Although such a number tends to vary with the radius and, as indicated with the test, is often different among patterns, it is not possible to observe any increasing or decreasing trend. In other words, it is not possible, for example, to identify clone evolution patterns which tend to be more frequent for a high clone radius. One could have expected, for example, that LP tends to be more frequent for clones with a high radius, since the developer might not be able to track all of them. However, results indicate this is not the case for the systems we have analyzed (the null hypothesis cannot be rejected).

4.4 RQ4: Is there any relationship between clone evolution patterns and the occurrence of bugs fixings?

Table 5 reports the number and percentage of change sets due to bug fixing, classified by evolution pattern. The percentage is computed across all the change sets of the observation period.

To test whether the proportion of bug fixing related changes is different across evolution patterns, we performed a proportion test (H_0: there is no significant difference among proportions of bug fixing changes for different evolution patterns). For AST-based clones, results indicate that proportions are significantly different for ArgoUML (p-value=$2.6 \cdot 10^{-6}$), JBoss (p-value=0.0003), and PostgreSQL (p-value=0.0009), and marginally different for OpenSSH (p-value=0.06). Except for ArgoUML, where the highest proportion of bug fixing changes is found for the IE pattern, in all the other cases the highest proportions are found for the L2 and LP patterns.

For token-based clones, proportions of bug fixing related changes among clone evolution patterns are significantly different for ArgoUML (p-value=0.002) and PostgreSQL (p-value=0.02). In the first case, proportions are significantly higher for LP (0.53), in the second case for L2 (0.33). No significant difference in proportions is found for JBoss (p-value=0.34) or for OpenSSH (p-value=0.14), although in the first case proportions for L2 (0.19) and LP (0.19) are higher than for CO (0.14) and IE (0.16).

It should be noted that the above analysis only aimed at analyzing whether the occurrence of bugs fixings among evolution patterns changed in proportion. It is not possible with the information available to clearly establish a cause-effect relationship between the clone evolution pattern and the occurrence of bug fixing changes.

Table 5 Number and percentage of bug fixing changes among different evolution patterns.

System	AST-based											
	CO			**IE**			**L2**			**LP**		
	all	bug	%	all	bug	%	all	bug	%	all	bug	%
ArgoUML 0.9	6518	2869	44%	4698	2307	49%	574	270	47%	394	179	45%
JBoss 3.2.2	4313	757	18%	2186	393	18%	69	20	29%	160	42	26%
OpenSSH 2.9.9	7376	1127	15%	1180	151	13%	0	0	NA	132	23	17%
PostgreSQL 6.5	19020	4346	23%	11826	2820	24%	713	176	25%	3135	814	26%
System	**Token-based**											
	CO			**IE**			**L2**			**LP**		
	all	bug	%	all	bug	%	all	bug	%	all	bug	%
ArgoUML 0.9	4590	2045	45%	3249	1561	48%	113	50	44%	268	142	53%
JBoss 3.2.2	2971	420	14%	1830	289	16%	16	3	19%	59	11	19%
OpenSSH 2.9.9	771	120	16%	536	64	12%	48	5	10%	0	0	NA
PostgreSQL 6.5	2412	553	23%	1654	374	23%	164	54	33%	202	44	22%

4.5 Examples of Evolution Patterns

This section discusses some examples of evolution patterns detected in the four software systems we analyzed, with the aim of explaining the rationale/cause of each evolution.

4.5.1 Consistent evolution

The Consistent (CO) evolution pattern can be observed in all systems, especially where clones belong to a critical part of the system and a lack of change propagation causes an earlier unwanted side effect. For example, in ArgoUML, the class *GenAncestorClasses* has been cloned from *GenDescendantClasses*. Both are utility classes used to navigate class hierarchies in UML diagrams. Such classes underwent different refactoring changes during their life, always consistently performed within the same change set and propagated across

clone fragments. The consistent evolution was needed since these classes implement *dual* features, i.e., one allows for navigating hierarchies upward and the other for navigating downward. Without such a propagation, the system would have exhibited a failure. In JBoss, classes *CmdAUTH* and *CmdRCPT* have a similar *SMTPResponse handleRequest* method, which underwent a number of signature changes consistently propagated to both methods in the same change sets.

4.5.2 Independent evolution

Clone independent evolution mainly happens because developers are prone to start coding with a common template and then adapt it to the required functionality. In ArgoUML, for example, the classes *GeneratorJava* and *GeneratorDisplay* contain some common cloned methods. In their first versions, the methods *generateAttribute* of the two classes were detected as clones. These two classes generate Java code and UML diagrams from a model, respectively. The second class becomes more complex in newer ArgoUML versions to account for enhanced visualization features. Thus, starting from revision 1.8 of *GeneratorJava* and 1.4 of *GeneratorDisplay*, the two classes evolved independently.

4.5.3 Late propagation

The most obvious reason for late propagation is when a developer forgets to propagate the change to other fragments of a clone class. We observed such a scenario in a two block-sized clones of PostgreSQL detected in modules *parse_oper.c* and *parse_func.c*. The first underwent a bug fix (August, 26 1999) related to an expression returning an erroneous value. The same bug was discovered six months later in the other clone fragment (February, 20 2000). In the CVS commit note, the developer mentioned the missed change, *"...I had previously fixed the identical bug in oper_select_candidate, but didn't realize that the same error was*

repeated over here...". Similarly, in ArgoUML a bug fix was performed at different times by the same developer on two block-sized clones in *FigMNodeInstance* and *FigMNode*.

Another interesting case of late propagation is due to refactoring. For example, in ArgoUML the classes *CrNoIncomingTransitions* and *CrNoOutgoingTransitions* are two class-sized clones. ArgoUML underwent a refactoring that introduced a Facade pattern that encapsulates a number of features to test if an object is of a certain type. This lead to many changes in the source code where statements such as `if (sv instanceof MPseudostate)` were changed to `if (ModelFacade.isAPseudostate(sv))`. Such changes were not propagated in one change set. Since a refactoring does not affect the behavior, such a late propagation did not cause any problem/inconsistency. For this reason, developers did not pay much attention to propagate it immediately.

5 Lessons Learned

This section reports lessons learned organized as Pieces of Evidence (PoE) that we collected from our quantitative results, and provides examples of different clone evolution patterns occurring in the systems we analyzed.

PoE 1: Clones are often consistently propagated immediately. Developers tend to consistently propagate clone changes immediately where this is needed. This result is confirmed in all systems analyzed, where we found that the proportion of cases where at least one clone fragment was lately propagated to be below 16%. This suggests that developers "know" when and how to propagate clone changes; provided that they have time and effort available and realize the propagation is needed, they do it. Also, this suggests that clone refactoring—which would limit the need for clone change propagation—is not necessary at all for developers, as earlier suggested by Cordy [15]. Moreover, it can generate

unwanted side effects, e.g., refactoring can, on its own, introduce errors. Of course, further investigation is needed to better study the effect of refactoring on clone maintenance and fault-proneness. Section 4.5.1 reported some examples of consistent clone evolution.

In only a few cases, clones were lately propagated, i.e., there was the need for propagating the clone change on at least one another clone fragment. However, this was not done immediately for different reasons, that can be the lack of resources/effort, development strategies, or just because developers were not able (or forgot) to keep track of clone changes (see Section 4.5.3). Better support for clone tracing would probably further reduce this phenomenon.

PoE 2: Using clones for templating is a common phenomena in software systems

Other than consistent evolution, we observed that (again in all systems) a large percentage of clone classes underwent independent evolutions. Such a percentage was, in some cases, higher and, in other cases, lower than the one of consistent evolution. As shown in Section 4.5.2, independent evolutions often correspond to scenarios where developers template the source code and then adapt it to realize different specific features. However, this result must be carefully considered: although the propagation was not observed in the analyzed time interval, there is no guarantee that it would not happen in the future.

PoE 3: clone characteristics do not influence the evolution pattern

Intrinsic characteristics of the clone, such as clone granularity, clone radius, programming language used to implement the system analyzed, and the clone detector used might potentially influence the evolution pattern. However, we could not find, at least for the systems we analyzed, any empirical evidence of that.

In particular, there is no evident relationship between clone granularity and evolution patterns. This seems to depend rather on the particular system analyzed. For example, JBoss and PostgreSQL exhibited a higher number of independent evolutions, while consis-

tent changes were predominant in ArgoUML. This suggests that (i) when consistent clone changes need to be performed, developers are able to cope with them even when clones are small code fragments (e.g., block level clones) and (ii) code templating is performed at every granularity level. For example, code blocks are cloned to produce similar control logic, data structure access, synchronization blocks, etc. Also, entire methods are cloned from one class to another, and then properly adapted. As shown in Section 4.5.1 we find code templating practice adopted at different levels of clone granularity.

When evolution patterns significantly differed, in proportions, among levels of granularity—i.e., in the case of JBoss—we noticed that clones having a higher granularity—i.e., classes/files and then methods/blocks—were more subject to independent evolution than smaller clones. This clearly depends on the different nature of these clones. Smaller clones are code fragments realizing a similar piece of logic that developers replicate where needed, or maybe just always implement in the same way. On the other hand, developers do not duplicate entire classes or methods just for replicating a piece of functionality; they rather do it for templating and evolving purposes. Section 4.5.2 provided some examples on independent evolutions coming from code templating.

One could expect that the clone radius could have impact on the clone evolution pattern, in particular clone fragments that are far from each other in the code directory structure could suffer from late propagations, because it is more difficult to locate them. However, our empirical results indicate this is not the case: developers are able to keep track of clones also when they are spread across the software system code structure.

Finally, results obtained on the four systems we analyzed do not indicate any particular influence of the clone detection approach and tool used (token-based vs. AST-based), nor of the programming language (C, Java) or paradigm (procedural vs. object-oriented) used to implement the system analyzed. The first result was quite expected: although AST-based

clone detection often identifies a smaller set of clone classes, in most cases proportions of different evolution patterns are not significantly different. More than the clone structure—sequence of tokens vs. similar ASTs—evolution patterns depend on the clone usage intent and on how maintenance activities were performed. In fact, some differences, e.g., in the percentage of independent evolution, were found between different systems.

PoE 4: high proportions of bug fixing changes occur for clones exhibiting late propagations

The last research question concerns the relationship between the evolution pattern and the occurrence of bug fixing changes. Results showed that, in all cases but for ArgoUML, there is a significantly high proportion of bug fixing changes for late propagations (L2 and LP). Clearly, we cannot claim the presence of any cause-effect relationship. However, since late propagations require the same change to be performed multiple times on different fragments, and considering some examples we found by inspecting CVS/SVN log messages, results suggest that, at least, late propagations deserve more attention and should be limited as much as possible. In Section 4.5.3 we reported some late propagation examples due to bug fixing.

6 Threats to Validity

This section discusses threats to validity that can affect the results reported in Section 4, following a well-known template for case studies [44].

Regarding *construct validity*, threats can be due to the measurement performed, in particular: (i) to the precision and recall of the clone detection tool and (ii) to the errors that can be introduced by the clone pattern classification approach. Errors due to lack of precision were limited by the chosen setting for both the token-based and AST-based approaches. We

are aware that such a setting would reduce the recall. Indeed, as indicated in Section 3.4, this choice at least permits the detection of a reliable sample of clones that can be used to answer our research questions.

Our approach is sensitive to the movements of source code from one code section to another. This kind of movement could cause the incorrect classification of clones into the *IE* evolution pattern, although the clone fragments are still consistent. We explain this limitation through the example shown in Fig. 7. Let us consider that the clone class $CC_{z,k}$ has two clone fragments CF_1 and CF_2 with two clone sections. Let us consider that both clone sections in the snapshot S_k are consistent, and a small code fragment is moved from the first clone section to the second one, because of the changes that occurred between S_k and S_{k+1}. Although the overall clone fragment is not changed, the similarity measure (*NLD*, see equation 2) computed for clone sections in the snapshot S_{k+1} can be below the threshold, resulting in the *IE* evolution pattern. If an entire fragment is moved, such a problem does not happen, since the fragment remains unaltered.

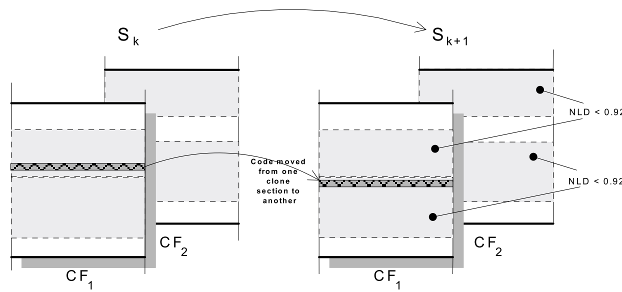


Fig. 7 Movement of source code from one clone section to another.

Another sensitivity problem can be due to code refactorings that happen in the clone section borders. They can be wrongly classified as *LP* clone evolution patterns if such refactorings do not happen in the same snapshot.

The relationship $CF_x \in CC_{z,k}$ between a clone fragment and a clone class is computed using a clone detection tool in the first analyzed snapshot, and then using the differencing approach described in Section 2.3. In other words, if in snapshot $k+1$ the differencing approach considers a fragment different from the other one, then $CF_x \notin CC_{z,k+1}$. Although this has the advantage of allowing to trace specific clones without re-running the clone detection tools on each snapshot, clone detectors could still produce different results.

Finally, results of **RQ4** can be affected by the performance of the heuristic adopted to check whether a change is a bug fix or another kind of change. Such a heuristic has been successfully adopted to classify bugs in several empirical studies, e.g., [20, 39].

Threats to *internal validity* concern internal factors that may affect the outcome of the study. This is mainly an exploratory study [44], thus not particularly affected by this kind of threat. Indeed, one possible threat can occur when considering the relationship between the clone evolution pattern and the occurrence of bug fixings. In fact, the bug may or may not be due to the clone evolution pattern. However, the intent of this work is to study the percentage of bugs that occurred for different clone evolution patterns, rather than to find a cause-effect relationship between the two phenomena. Evolution histories of different length could also have affected the results, influencing the resulting evolution patterns. We limited this threat for clones found in the same system by restricting our analysis to clones detected in a single snapshot such that the period of observation was the same for all clones. We believe that a comparison between different systems is not possible since different systems underwent different evolution strategies. Another possible threat to internal validity is the dependence of the clone evolution on the clone age, due to the fact that we detected clones in a precise snapshot: they could have been introduced before at different times. However, we limited this threat by choosing a relatively early (for ArgoUML the first one) snapshot in the software system lifetime. In some other cases (e.g., OpenSSH) we could not choose the first

snapshot because it corresponded to a preliminary (e.g., alpha) release, where the system size and set of functionality was still limited with respect to subsequent (stable) releases.

Threats to *external validity* are related to what extent we can generalize our findings. We considered four software systems, differing for their domain, size, and programming language. Results indicated both common findings, but also findings specific to some systems, in some cases due to the system size or programming language. Replication on further systems can be useful. Also, our study focuses on the evolution of clones detected on a single snapshot. Of course, clones detected on different snapshots of the same system could have produced different results, although we, at least, observed pretty consistent results for four different software systems. Regarding the clone detection approaches, we adopted the most widely used ones. Other approaches can produce different results although (i) the metric-based approach is often suited for coarse-grained clones and it might not be useful to study the evolution of smaller code fragment clones and (ii) the graph-based clone detection approach could suffer from scalability problems [10].

Threats to *reliability validity* concern the possibility to replicate the study. The source code and the CVS repositories of the three systems are publicly available, as well as the clone detection tools. The clone pattern classification technique and its settings are thoroughly described in the paper. Finally, as indicated in Section 4, we put raw data available on-line to ensure the replicability of analyses.

Conclusions made in the paper were supported by proper statistical tests as discussed in Section 3.4.1, in all cases we made sure that the preconditions for the applicability of these tests hold.

7 Related Work

This section relates the present work with the existing literature, mainly related to (i) empirical studies aimed at analyzing the presence and the evolution of source code clones; (ii) approaches to study clone genealogies and to trace clone evolution.

7.1 Empirical studies on the presence and evolution of clones

The literature reports several empirical studies concerning the use of clones in large software systems and the evolution of clones across releases. Mayrand *et al.* [38] proposed a metric-based clone detection approach and studied the presence of clones in a telecommunication system. In their study, they found that between 5% and 20% of source code was cloned code. Clone detection case studies on the Linux Kernel have been performed by Godfrey *et al.* [22, 23], who performed a preliminary investigation of cloning among Linux SCSI drivers. Antoniol *et al.* studied the presence [14] and then the evolution [4] of code clones in the Linux Kernel using a metric-based approach. They found that the percentage of cloned code did not change during software evolution and that, while new clones were added, some were factored out. Kasper and Godfrey [27] proposed a classification of clones based on their distance, i.e., within the same function, or file, or directory, and based on their granularity, i.e., block, function, or file. They used such a classification on clones detected in the Linux Kernel. They also proposed a tool support for the comprehension of software clones [28]. Kapser and Godfrey [29] also presented a taxonomy of cloning patterns, describing for each pattern the motivation, the pros and cons, the way clones manifest in the code structure, and clone management issues. Each pattern is described using examples detected in large software systems such as the Linux Kernel or the Apache Web server. We share with them the result that in many cases the source code is first cloned to realize “tem-

plates" of new pieces of functionality that then evolve independently. We also agree with Kapser and Godfrey that, in other cases, changes on clones are almost always consistently propagated. Other than clones, redundancies—e.g., redundant assignments, conditionals, or dead code—can be the cause of errors, as shown by Xie and Engler [43]. As reported by Al-Ekram *et al.* [1] it also happens that, in some cases, the similarity between source code fragments occurs "by accident", since different developers solve similar problems in the same way. Our study shares with them the evidence that, in most cases, a large portion of the "inconsistent" clone changes is due to an independent evolution. Nevertheless, there are a few cases in which, as shown in Section 4.5, inconsistent clone changes caused a fixed bug to appear again. Aversano *et al.* [5] performed a fine-grained analysis of clone evolution pattern on ArgoUML and DNSJava. The present paper goes beyond [5] in that:

1. It proposes and applies an automatic approach to classify clone evolution patterns, rather than relying on a manual analysis
2. It reports four case studies and uses two kinds of clone detection approaches (AST-based and token-based) instead of just using the AST-based approach.

Another reason for bugs introduced by clones is when developers copy-and-paste source code fragments and do not make changes (such as identifier renaming) consistently [34].

7.2 Clone genealogy and tracing

Although many studies on clone evolution have been performed in the past, such studies were performed at a coarse-grained level, i.e., considering the evolution of clones across releases. Such studies, however, neither considered phenomena such as the propagation of changes to clones belonging to the same clone class nor did they investigate the relationship between the presence of clones and the source code fault-proneness. The availability of

techniques to analyze fine-grained changes by mining CVS/SVN repositories opened new analysis scenarios to better understand how clones evolved. An empirical study of code clone genealogies has been presented by Kim *et al.* [30]. They proposed a formal definition of clone evolution and built a clone genealogy tool to automatically extract the history of code clones from a source code repository. According to the empirical study they performed, refactoring does not always constitute an improvement, and in many cases it is not worth doing it, due to the fact that many code clones exist in the system for only a short time (although this could happen because of regular refactoring activities). Moreover, even for long-lived clones, where refactoring could be desirable, it is often hard to perform due to programming language limitations. As anticipated in Section 3, we share with Kim *et al.* part of their clone classification (shown in Figure 8), whilst the objective of our study is different:

1. While Kim *et al.* aim at reconstructing a clone genealogy, we aim at analyzing how a clone class is maintained over time, keeping track of previous changes made. In particular, what Kim *et al.* classified as inconsistent change can be caused by (i) a different (independent) evolution of clone fragments during the analyzed timeline, i.e., each clone fragment undergoes different changes or (ii) a temporary situation, since after a while the clone fragments become consistent again. This is the case of late propagation (LP or L2).
2. We use a fine-grained model of clones that allows us to deal with gapped clones also, by analyzing the evolution pairs of clone sections—i.e., gapped portions of clone fragments.

Krinke [32] also presented a study on changes to code clones with an aim to understand if these change are consistent to all code clones of a clone group. His results are consistent

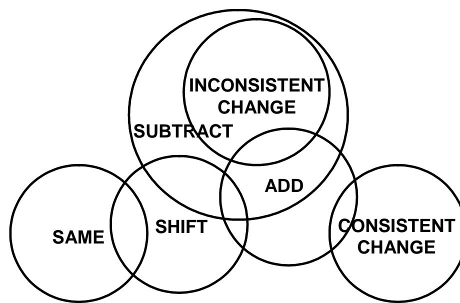


Fig. 8 Clone evolution patterns defined by *Kim et al.* [30].

with ours and show how, during the evolution of a system, code clones of a clone group are consistently changed; moreover, when the changes are inconsistent the missing changes will appear in a later version. There are several differences between Krinke's work and ours, such as:

1. although he is able to distinguish between consistent and inconsistent changes, our approach is also capable of distinguishing cases of independent evolution from cases of late propagation.
2. Krinkes' analysis is performed at version level, while our analysis is a fine-grained analysis on change sets identified according to [45]. This allows us to precisely identify when a clone change is propagated on other clone fragments.
3. While Krinke's tracing approach relies on *diff*, we use a fine-grained change tracing approach [13] that is able to trace changes where *diff* fails (e.g., moving a method).
4. Our work relies on AST-based clone detection tools (Bauhaus and Simscan) and on a token-based clone detection tool (CCFinder), while Krinke uses Simian, which is a text-based clone detection tool.

The relationship between clones and change couplings has been examined by Geiger *et al.* [21]. They proposed a framework to examine code clones and relate them to change couplings taken from release history analysis. The results showed that, although the rela-

tionship is not statistically significant, the analyzed systems exhibit a reasonable number of cases where the relationship holds. Although having different objectives, our analysis shares with Geiger *et al.* the use of change sets to analyze clone evolution.

Bakota *et al.* [7] presented an approach that identifies clone smells from one version to another. The approach is based on a similarity measure which classifies clones into four categories: (i) clones that disappear, (ii) newly introduced clones, (iii) clone fragments that are moved between clone classes, and (iv) late propagations of clones. Their tracing approach is based on tree similarity measures, since their clone detection approach is AST-based. Bakota *et al.* [7] is different (and complementary) to our work for three main reasons:

1. while their focus is to map clones detected in different releases of a software system, i.e., to build a clone genealogy, our approach aims at analyzing how clone fragments belonging to the same clone class are maintained. Basically, we add to the clone genealogy the capability of detecting consistent and inconsistent changes.
2. We trace clones by starting from clone section pairs using a variant of a code tracing approach previously proposed [13] and adapted to analyze clones. Instead, they map clones based on a similarity measure defined as a weighted sum of six features such as file name, clone position in the file, AST, node, etc.
3. Although the approach of Bakota *et al.* [7] can be applied at a finer-grained level, their empirical study was performed by tracing clones between software system releases, rather than file revisions as extracted from the CVS.

Duala-Ekoko and Robillard [16] proposed a technique for tracking clones in evolving software, where their technique notifies developers of modifications to clone regions and supports the simultaneous editing of clone regions. They assessed, by means of an experiment, the usefulness of the proposed approach. This confirms the conjecture that tool sup-

port for tracing clone evolution is useful. Lozano *et al.* [35] highlighted the usefulness of historical information to assess the impact of bad smells, which included the harmfulness of clones. They also performed an empirical study [36] and found that methods containing cloned code tend to change more frequently than other methods. Their preliminary results seem to indicate that clones could be the cause of repeated changes since developers are not aware of their presence.

8 Conclusions and future work

This paper reported an empirical study aimed at investigating, in four different software systems, the evolution of source code clones. The study is based on an automatic process for the tracing and classification of clone evolution patterns. The availability of such an automatic process allowed us to perform a large-scale analysis of clone evolution that could not have been feasible by means of a manual inspection of clones detected in several revisions of the same file.

It was found that most of the clone classes—a percentage above 70%—were either consistently changed or they underwent an independent evolution (i.e., according to the evolution patterns we defined, they underwent different changes). In only a small percentage of cases, usually below 16%, they actually underwent a late propagation, which is a phenomenon that could produce undesired effects such as causing an already fixed bug to appear again in the future. In most cases, we did not find any significant difference in the evolution patterns for clones having different granularities or radii.

When analyzing the relationship between the evolution patterns and the occurrence of bug fixing changes, we found that the highest proportion of such changes are found in clone classes that underwent a late propagation. This represents an important result: although what

we found does not demonstrate a cause-effect relationship between the late propagation and the occurrence of bug fixing changes, at least it is possible to say that, when such a clone evolution pattern occurs, the code is more bug-prone than in other cases and thus worthy of more attention.

By inspecting the source code, we were able to find, as shown in Section 4.5, examples supporting our conjecture about how maintainers changed clones following the different evolution patterns. For example, we found a clear explanation for a late propagation: *"...I had previously fixed the identical bug in oper_select_candidate, but didn't realize that the same error was repeated over here..."*.

Although developers consistently change clones, the way they keep track of these clones is mainly based on their personal knowledge, on ad-hoc mechanisms, or on the use of common code browsing tools and integrated development environments. This suggests that, although developers are already able to successfully cope with clone change propagation, recommender systems able to support clone tracing, such as the one proposed by Duala-Ekoko and Robillard [16], or tools automatically tracing clones, e.g., based on the approach presented in this paper, can be beneficial.

There are still a number of factors to be investigated. Among others, it would be useful to analyze what is the effort of keeping clones aligned, and whether clone detection tools serve as useful support for that. This can be done, for example, by monitoring the evolution of open source projects (as Bouktif *et al.* [11] did). Then, factors such as the kinds of changes clones undergo, or the relationship between the clone evolution patterns and factors such as the programming language adopted are worthy of more specific studies. Finally, the study can be replicated on further software systems belonging to different domains and developed using different programming languages.

9 Acknowledgments

We would like to thank the anonymous reviewers for their very constructive comments on early versions of this manuscript. We also thank William Harris for his review comments that helped us to improve the draft. Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta are partially supported by the project METAMORPHOS (MEthods and Tools for migrAting software systeMs towards web and service Oriented aRchitectures: exPerimental evaluation, usability, and tecHnOlogy tranSfer), funded by MiUR (Ministero dell'Università e della Ricerca) under grant PRIN2006-2006098097. Suresh Thummalapenta is partially supported by NSF grant CCF-0725190 and ARO grant W911NF-08-1-0443.

References

1. R. Al-Ekram, C. Kasper, R. Holt, and M. Godfrey. Cloning by accident: An empirical study of source code cloning across software systems. In *International Symposium on Empirical Software Engineering (ISESE 2005)*, pages 376–385, 2005.
2. G. Alkhatib. The maintenance problem of application software: an empirical analysis. *Journal of Software Maintenance*, 4(2):83–104, 1992.
3. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.*, 28(10):970–983, 2002.
4. G. Antoniol, E. Merlo, U. Villano, and M. Di Penta. Analyzing cloning evolution in the Linux Kernel. *Information and Software Technology*, 44:755–765, Oct 2002.
5. L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems, CSMR 2007, 21-23 March 2007, Amsterdam, The Netherlands*, pages 81–90. IEEE Computer Society, 2007.
6. B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE '95)*, pages 86–95, July 1995, IEEE Computer Society.

7. T. Bakota, R. Ferenc, and T. Gyimóthy. Clone smells in software evolution. In *Proceedings of the International Conference on Software Maintenance (ICSM '07)*, pages 24–33, Paris, France, 2007. IEEE Computer Society.
8. M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the Working Conference on Reverse Engineering*, pages 98–107. IEEE Computer Society, 2000.
9. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE Computer Society, 1998.
10. S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9):577–591, 2007.
11. S. Bouktif, G. Antoniol, and E. Merlo. A feedback based quality assessment to support open source software evolution: the GRASS case study. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 155–165. IEEE Computer Society, 2006.
12. S. Bouktif, Y.-G. Gueheneuc, and G. Antoniol. Extracting change-patterns from cvs repositories. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 221–230. IEEE Computer Society, 2006.
13. G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*, page 14. IEEE Computer Society, 2007.
14. G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Identifying clones in the Linux Kernel. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 90–97. IEEE Computer Society, 2001.
15. J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 196–206. IEEE Computer Society, 2003.
16. 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*, pages 158–167, Minneapolis, MN, USA, 2007. IEEE Computer Society.
17. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of 19th IEEE International Conference on Software Maintenance*, pages 23–32, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society.

18. M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 321–330. ACM, 2008.
19. H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 190–197, 1998.
20. H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13. IEEE Computer Society, 2003.
21. R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, number 3922 in Lecture Notes in Computer Science, pages 411–425, Vienna, Austria, March 2006. Springer.
22. M. W. Godfrey, D. Svetinovic, and Q. Tu. Evolution, growth, and cloning in Linux: A case study. In *CASCON workshop on 'Detecting duplicated and near duplicated structures in largs software systems: Methods and applications*, 2000.
23. M. W. Godfrey and Q. Tu. Evolution in open source software:a case study. In *Proceedings of the 2000 International Conference on Software Maintenance*, pages 131–142, 2000.
24. L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 96–105. IEEE Computer Society, 2007.
25. 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*, 28(7):654–670, July 2002.
26. C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, and P. Weiβgerber. Subjectivity in clone judgment: Can we ever agree? In *Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings*. Eds.: Internationales Begegnungs-und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007., 2007.
27. C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6-7 September 2004, Kyoto, Japan*, pages 85–94. IEEE Computer Society, 2004.
28. C. Kapser and M. W. Godfrey. Improved tool support for the investigation of duplication in software. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 305–314. IEEE Computer Society, 2005.

29. C. Kapser and M. W. Godfrey. 'cloning considered harmful' considered harmful. In *Proceedings of the 2006 Working Conference on Reverse Engineering*, pages 19–28, Benevento, Italy, October 2006. IEEE Computer Society.
30. M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, pages 187–196, Lisbon, Portogal, September 2005. ACM Press.
31. J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Working Conference on Reverse Engineering*, pages 301–309, Stuttgart, Germany, Oct 2001.
32. J. Krinke. A study of consistent and inconsistent changes to code clones. In *14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada*, pages 170–178, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
33. V. I. Levenshtein. Binary codes capable of correcting deletions,insertions, and reversals. *Cybernetics and Control Theory*, (10):707–710, 1966.
34. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, Mar. 2006.
35. A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 31–34, New York, NY, USA, 2007. ACM.
36. A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*, page 18. IEEE Computer Society, 2007.
37. A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 124–135, Portland, OR, USA, May 2003. IEEE Computer Society.
38. J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253, Monterey, CA, Nov 1996. IEEE Computer Society.
39. A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2000.
40. S. P. Reiss. Automatic code stylizing. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 74–83. ACM, 2007.

41. Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *8th IEEE International Software Metrics Symposium (METRICS 2002), 4-7 June 2002, Ottawa, Canada*, pages 67–76. IEEE Computer Society, 2002.
42. E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*, pages 97–107. IEEE Computer Society, 2002.
43. Y. Xie and D. R. Engler. Using redundancies to find errors. In *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 51–60, 2002.
44. R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.
45. T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.