View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Electronic Communications of the EASST (European Association of Software Science and...

Electronic Communications of the EASST
Volume 63 (2014)

Proceedings of the
Eighth International Workshop on
Software Clones
(IWSC 2014)

Studying Late Propagations in Code Clone Evolution Using Software Repository Mining

Hsiao Hui Mui, Andy Zaidman and Martin Pinzger

11 pages

Guest Editors: Nils Göde, Yoshiki Higo, Rainer Koschke
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Studying Late Propagations in Code Clone Evolution Using Software Repository Mining

Hsiao Hui Mui[1], **Andy Zaidman**[1] **and Martin Pinzger**[1]

[1] hsiaomui@gmail.com, a.e.zaidman@tudelft.nl
Software Engineering Research Group
Delft University of Technology, the Netherlands

[2] martin.pinzger@aau.at
Software Engineering Research Group
University of Klagenfurt, Austria

Abstract: In the code clone evolution community, the *Late Propagation* (LP) has been identified as one of the clone evolution patterns that can potentially lead to software defects. An LP occurs when instances of a clone pair are changed consistently, but not at the same time. The clone instance, which receives the update at a later time, might exhibit unintended behavior if the modification was a bugfix. In this paper, we present an approach to extract LPs from software repositories. Subsequently, we study LPs in four software systems, which allows us to investigate the propagation time, the clone dispersion and the effects of LPs on the software.

Keywords: code clone evolution, late propagation, software repository mining, bugs

1 Introduction

Research in the area of code clones has shown that 7% to 23% of the code in large software systems contains duplicated source code fragments [1, 2]. While these so-called code clones are generally considered harmful [3], other studies indicate the contrary [4]. Research has long focused on techniques for (a) finding and (b) subsequently refactoring code clones [5], however, more recently, the code clone evolution research community has taken interest in *managing* code clones, rather than refactoring them [6, 7]. Code clone management tools, such as *CloneTracker* [6] or *CloneBoard* [7], help developers to understand and remember where code clones are in the system; they can also help to propagate changes from one clone instance, to all instances of the clone relation.

Kim et al. [8] investigated the evolution of code clones and they found patterns of clone evolution. Aversano et al. [9] expanded on this research by adding two new patterns. One of these patterns, the *Late Propagation* is of particular interest to investigate, as it can provide an indication of the usefulness of code clone management tools. Figure 1 shows an example of the late propagation code clone evolution pattern. It shows two duplicated code fragments C1 and C2 in version V_i, both belonging to the same clone relation. In a subsequent version (V_j) C1 is modified while C2 is not, which means that both clones are now inconsistent. In version V_k C2

is also updated and both clones are now consistent again. A late propagation is defined as: *a change which is propagated consistently across clones, although at a later point in time*.

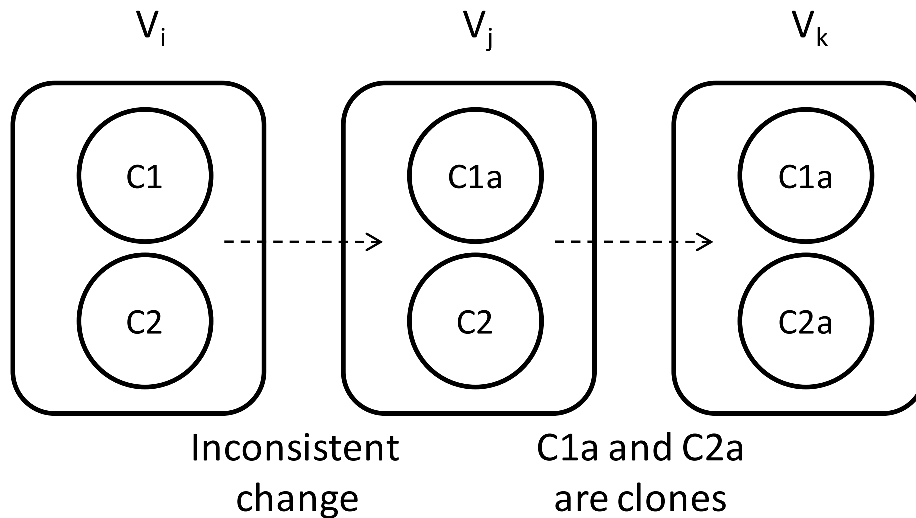

Figure 1: The Late Propagation code clone evolution pattern.

This clone evolution pattern is likely to occur when a developer forgets to update one or more code fragments in a clone relation, and propagates the changes to the other clones in a clone relation at a later point in time. This situation is where code clone management tools can help, as these tools help developers to remember instances of code clone relations.

In this paper, our main research question is: *What is the impact of the late propagation code clone evolution pattern?* Answering this question will provide an initial answer as to the usefulness of code clone management tools [10]. In order to answer our principal research question, we aim to answer the following sub-questions first using a software repository mining approach:

RQ1 Is the late propagation code clone evolution pattern frequently occurring in practice?

RQ2 Do late propagations typically lead to bugs?

RQ3 Is there a connection between the package distance and the propagation time of late propagations?

The structure of our paper is as follows: Section 2 presents our approach for detecting late propagations. In Section 3 we present our findings of studying 4 software systems, while Section 5 presents our conclusions and future work.

2 Approach

Several approaches to track the evolution of code clones have been proposed in literature. For instance, Kim et al. [8] map clone fragments between version i and version $i+1$ by using a

location overlapping function in combination with *unix diff*. To determine how much the source code of a code clone has changed, a textual similarity function is defined that measures the textual similarities between a given version and a previous version of a software system. The approach proposed by Kim et al., however, is not capable of finding LPs. This is because their approach registers an *independent evolution pattern* when the clone instances become inconsistent and they have no mechanism for detecting reappearing clones that might be a member of a previous clone relation several revisions ago, which is essential for finding late propagations.

2.1 Toolchain Structure

This section describes the different components of the late propagation finder tool, of which an overview is depicted in Figure 2. The tool is written in Java and currently only operates on SVN repositories.

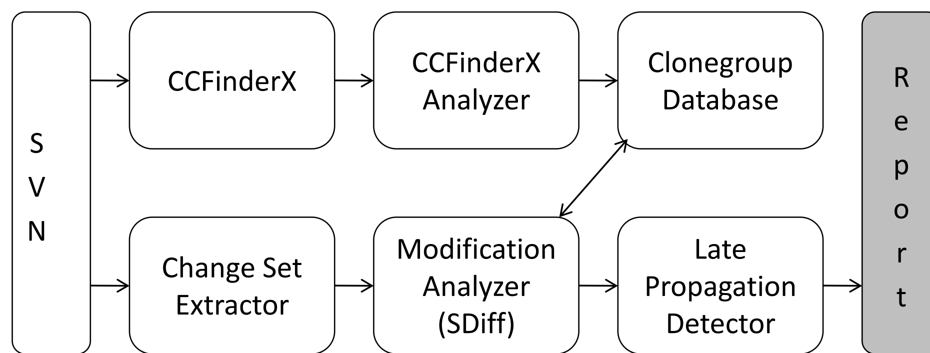

Figure 2: Overview of the late propagation detector tool.

Our approach consists of the following steps. In the initialization phase, we start by extracting the first revision of a software project from a SVN repository and use a clone detection tool to find clones in the codebase; our tool will build clone families, which are saved into a XML database. Then for each subsequent snapshot, we:

1. Determine if any changes are made to the code base, if so, download the sources of that revision.
2. Extract all change sets of that snapshot.
3. Analyze the change sets to see if there are any modifications made to clones in the XML database.
4. Update the changes made to the code duplications into the database.
5. Find out if any LPs have occurred by analyzing the changed clones.
6. Run the clone detector to detect newly introduced clones and update the database accordingly.

The following paragraphs will explain some of the above steps in detail.

Initialization phase. In the first step, the first snapshot is downloaded from a SVN repository and analyzed by a clone detection tool.

Based on existing comparison studies [11], we have selected the CCFinderX [12] tool as the clone detection tool used in this paper. We configured CCFinderX to find clones of at least 30 tokens. The clone families that CCFinderX identified are saved in a XML database.

Extraction and Analysis of Change Sets. Our tool extracts change sets from the SVN log. Each log entry contains information about a commit, such as the author name, revision number, timestamp, log message and also the added, deleted and modified files. The *extractor module* filters out any irrelevant change sets (e.g., modifications to property files) and passes the log entries it has fetched to the *Modification Analyzer* module. This module is responsible for determining if any changes are made to the code clones that are stored in the database. There are three types of modifications that can occur in a clone fragment, which can make it inconsistent with other code duplications in a clone relation:

- A statement is added to a clone fragment, that increases the size of the code duplication.
- A section of the clone is deleted or the entire clone is deleted. In the former case the size of the clone has decreased and in the latter, it means that the code duplication was present in the previous revision but has disappeared in the current one.
- One or more statements in the clone fragment have been modified, but the clone size remains the same.

Additionally, two types of changes can happen outside the clone, which does not affect its clone relationship:

- A code fragment has been added above the clone and as a result the clone has moved downwards.
- One or more statements above the clone have been removed, which means that the clone has moved upwards.

In order to determine what kind of changes have occurred in a method, it is necessary to compare the current version of a source file with its previous revision. Existing studies [8] have used the Unix-Diff tool to find out which parts of a source file have changed. Unix-Diff or Diff is the standard tool for discovering the differences between two versions of a file, but it does have several limitations [13, 14]. In particular, Diff has problems in distinguishing between code modifications and code additions/deletions. Typically, it records a change in a source line as a deletion of the old line and an addition of a new one. It also can not detect refactorings such as method renamings/movements.

While there is a degree of subjectiveness in determining whether a program text has been replaced or changed, recently several approaches have been proposed that can identify edited lines with reasonable accuracy [13–15]. We have selected the Statement Diff or SDiff tool [13]

to track the changes made to code clones in the database. We chose SDiff because the other options are either closed source [15] or are implemented in another programming language [14].

SDiff combines line-based (determining the edit distance between two lines) and structural (using the structure of the source code) approaches in tracking software artifacts. It uses the abstract syntax tree of a source file to break up the code into class declarations, import statements, field declarations and methods [13]. SDiff only reports changes per statement, so we had to modify it to report changes per method instead of statement, so it can report changes inside a method as well as method addition, removal and renamings.

Using SDiff, any changes made to the clone fragments are passed to the Late Propagation Detector module and finally the clones in the database are updated as well.

Detecting Late Propagations. The Late Propagation Detector module is responsible for finding LPs by analyzing the change information from SDiff. If a clone fragment in a clone relation has been changed to such an extent that its size is different (due to statement additions or deletions) from other code duplications in the same group, it is clear that an inconsistent change has happened. If, however, the size of the modified clone fragment is the same as some related code duplications, it is necessary to compare them to decide whether a consistent or inconsistent change has occurred.

The Late Propagation Detector module uses the *Normalized Levenshtein edit distance* (NLD) [16] as a metric to measure the similarity between two clone fragments. The Levenshtein Distance (LD) between two strings *s1* and *s2* is defined as the minimum number of insertions, deletions and substitutions required to transform *s1* into *s2*. For example, for *s1* = ‘qwerty’ and *s2* = ‘azerty’, the edit distance between both strings is 2. The higher the LD, the more different both strings are. In order to conduct comparisons, the *Normalized Levenshtein edit distance* is used, which is ranged in the interval [0, 1], where 1 means that the strings match and 0 indicates that the strings are strictly different. The NLD for two non-empty string *s1* and *s2* is defined as [14, 17]:

$$NLD(s1,s2) = 1 - \frac{LD(s1,s2)}{max(s1,s2)}$$

where LD(s1, s2) is the *Levenshtein Distance* and *max*(s1, s2) is the length of the longest string. We have used the threshold values of a previous study to determine if a clone pair is consistent or inconsistent [17]: a pair of clones is inconsistent if the NLD is smaller or equal to 0.87 and consistent if the NLD is greater or equal to 0.92; between those two values the clone evolution is classified as unknown.

By using the NLD we can determine if the program texts of two clones are similar to each other and discover the clone evolution pattern. A consistent change is found if all clones in a clone relation have received similar updates and this is determined by computing the NLD between each clone fragment. Similarly, an inconsistent change is detected if one of the clone fragments has been modified differently from other clone relation members, as detected by the NLD.

A LP happens, as indicated earlier, when a clone in a clone relation is changed inconsistently from the rest of the clone relation. After a number of revisions the delayed update is propagated to the other related clones and as a result these code duplications are consistent again. This

realignment of clones is detected in the same way as consistent changes are found. Finally, the detector generates a report if a LP is found.

Finding new clones. CCFinderX is also used to detect clones that appear after the initial snapshot. Instead of processing the entire output of the clone detector, the CCFinderX Analyzer module uses the log entries to determine which file has actually changed and only processes the clones that are in the modified files. The module checks if any new clone is related to any code duplication in an existing clone relation and if that is the case, then the clone is added to the clone relation and the clone relation is updated in the database.

3 Experiment

We used four Java software projects from different domains for our experiment, see Table 1. Using the observations from these four systems, we will now try to answer our research questions.

System	# Snapshots	Authors	KLOC	Start date	SLP	LLP	Bugs	% Bugs	# of clone sets	# java files	Final version
Subclipse	1946	15	42-223	6/2003	1	6	2	28	632	762	1.6.17
JEdit	2481	20	102-335	9/2001	30	5	8	22	696	564	4.3.2
FreeCol	4935	35	40-445	4/2004	8	0	0	0	311	707	0.95
Seam	3005	29	2-148	8/2005	0	0	0	0	584	510	2.0

Table 1: Overview of selected open source projects.

3.1 RQ1: Are late propagations frequently occurring?

Table 1 shows that LPs are actually not that common. For example, in the case of JEdit, in which we found most LPs, we see that there are 30 short-term LPs (changes propagated within 24 hours) and 5 long-term LPs ($>$ 1 day) for a development history of approximately 9 years. Furthermore, if we compare these numbers to the total number of clone sets, we see that the total number of late propagations that we could detect is very low. For the case of JEdit we see that in the final version – 4.3.2 – there are 696 clone sets compared to 35 late propagations.

3.2 RQ2: Do late propagations induce bugs?

By again looking at the data from Table 1, we see that in two of the systems that we studied, LPs were related to bugs. For carrying out this analysis we looked at the commit message in which the clone became consistent again, i.e., the commit in which the change was propagated.

Our data shows that in the two systems where LPs were related to bugs, an LP has a 22% (JEdit) to 28% (Subclipse) chance of resulting in a bug. However, when making the distinction between short-term (SLP) and long-term (LLP) late propagations, our analysis shows that most bugs were related to LLPs (70%).

3.3 RQ3: What's the influence of package distance?

Clone radius is defined by Ueda et al. [18] as: *"For a given clone class C, let F be a set of files which includes each code fragment of C. Define RAD(C) as the maximum length of the path from each file F to the lowest common ancestor directory of all files in F"*. Since we are dealing with software systems written in Java, we are actually looking at the package distance between clones. A high package distance implies that the code duplications are scattered over packages, which results in related clones that are more difficult to find and update. A clone pair in the same source file has a distance of zero and if the pair of clones only share the same package, the distance becomes one.

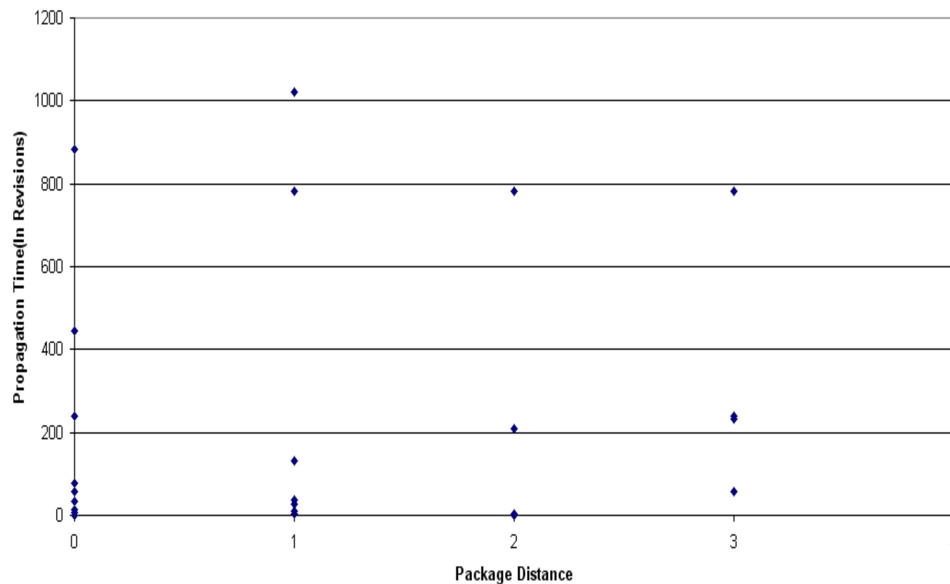

Figure 3: Propagation Time and Package Distance.

Figure 3 shows the overall package distance distribution (X-axis) versus the propagation time (Y-axis) of the 50 LPs (all projects together) that we found during our experiment. We see that most LPs happened within the same class or same package. It seems that the package distance has no real influence on the length of interval for the propagation to happen. This is quite unexpected, as one would estimate that widely spread code clones are more difficult to remember than code clones that are close to each other.

3.4 Threats to validity

External validity. We must point out that studying the development of 4 software systems is not representative for all development practices. We tried to mitigate this concern, by selecting 4 software projects from highly different domains.

Reliability. With regard to the clone detection technique that we are using, i.e., CCFinderX, it may find false negatives (miss clones) or false positives (detect clones that are actually not clones). Similarly, our clone tracking method, which is based on the SDiff tool, may also find false positives (e.g., false statement changes). With regard to CCFinderX, we performed a small analysis, which showed that CCFinderX is very good at finding Type 1 and Type 2 clones, with a recall level of respectively 100% and 89%. CCFinderX however does miss Type 3 clones quite easily (recall of 65%), confirming earlier results from Bellon et al. [11].

4 Related work

Göde and Koschke investigate the frequency and risks of changes to clones [19]. Their research is related to ours in the sense that they too investigate inconsistent changes. The set of inconsistent changes is determined manually for one case study that the authors of the paper have co-developed. In their study they found that 14.8% of all changes to clones are unintentionally inconsistent. Furthermore, only few of these changes have high severity, while the others are more or less cosmetic.

In a follow-up study to [19], Göde and Harder [20] confirm that the number of unintentional inconsistent changes is low. They also found the severity of all unwanted inconsistencies to be low as, for example, they were caused by changes to debugging code or semantic preserving changes. None of them prevents the system from working correctly.

Bettenburg et al. study inconsistent changes to clones at the release level [21]. The authors themselves determine whether a change is inconsistent or not. Based on a case study on two open source software systems, they observe that only 1 to 3% of inconsistent clones introduce software defects. Their study is different from ours in that we look for inconsistencies at a much finer level (revision level versus release level).

Juergens et al performed a study on the effect of inconsistent changes to clones in [22]. They first discussed with the software engineers of the respective systems whether an inconsistent change is intentional or not. Subsequently, they observed that every second unintentionally inconsistent change to a clone leads to a fault, thereby providing a strong indication of the fault-proneness of code cloning behavior.

Different from the aforementioned studies, our study investigates late propagations, which are inconsistent changes to clone groups that are regrouped at a later point. Our tool-chain is able to detect these late propagations automatically, which contrasts the aforementioned studies. Furthermore, for each of these late propagations we used log information left behind in the version control system to determine whether a late propagation can be connected to a reported bug.

Barbour et al. [23] have determined that 8 different late propagation patterns exist in the evolution of clone groups. There first observation is that clone genealogies that contain late propagations are more fault-prone than genealogies that do not contain late propagations. Considering the 8 different types of late propagation patterns, Barbour et al. conclude that while for most types of late propagations the fault-proneness is system dependent, the so-called LP7 and LP8 type of late propagations are the most likely to cause faults. In particular, an LP8 type late propagation involves no propagation at all, and occurs when a clone diverges and then re-synchronizes

itself without changes to the other clone in a clone pair. LP7 occurs when both clones are modied, causing a divergence and then both are modied to re-synchronize the clone pair.

5 Conclusion

In this paper we investigated the the impact of the *late propagation* code clone evolution pattern. From the preliminary evidence shown in this paper, we can conclude that the impact is *moderate*. We came to this conclusion by specifically answering the first two research questions that we set out in Section 1:

RQ1 *Is the late propagation code clone evolution pattern frequently occurring in practice?*
During our study of 4 open source software projects, we found that the late propagation pattern is not very common, with only 50 occurrences detected in total.

RQ2 *Do late propagations typically lead to bugs?*
By further investigating the 50 late propagations that we found by means of log file inspections, we found that around 25% of the detected late propagations gave rise to a bug.

RQ3 *Is there a connection between the package distance and the propagation time of late propagations?*
Another observation that we made is that the package distance has no influence on the speed by which changes are propagated over sets of inconsistent clones.

As can be seen from the study of 4 open source projects, the frequency of occurrence of the late propagation pattern is *low*. Furthermore, when a late propagation does occur, the chance of it becoming a bug also seems low (∼25% chance). These figures raise the question of whether code clone management tools are worthwhile? When the risk of a late propagation is low and the chance that a late propagation becomes a bug is also low, then do we need to invest in clone management tools? Additionally, when we see that the package distance has no real influence on the propagation speed of inconsistent clones (RQ3), it seems that it does not really matter whether an inconcistent clone is nearby or far away from the current editing location for a developer to remember (or to forget) to also change it.

In this paper we have made the following contributions:

- We have built a late propagation detection tool that follows clones through time, even as they (temporarily) become inconsistent.
- We have performed a small scale case study involving 4 Java open-source projects: Subclipse, JEdit, Freecol and Seams.

5.1 Future work

As future work, we aim to expand our study to include more subject software systems. We also intend to improve our toolchain by using an incremental clone detection technique [24], which should improve the efficiency of our approach.

Acknowledgements

Part of this research was sponsored by the Center for Dependable ICT (CeDICT), an initiative of NIRICT, the Netherlands Institute for Research on ICT. Our thanks go out to Cor-Paul Bezemer for helping us with the analyses.

Bibliography

[1] C. K. Roy and J. R. Cordy, “An empirical study of function clones in open source software,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2008, pp. 81–90.

[2] B. S. Baker, “On finding duplication and near-duplication in large software systems,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 1995, pp. 86–95.

[3] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1999, pp. 109–118.

[4] C. J. Kapser and M. W. Godfrey, ““cloning considered harmful” considered harmful: patterns of cloning in software,” *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[5] R. Koschke, “Identifying and removing software clones,” in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, ch. 2, pp. 15–36.

[6] E. Duala-Ekoko and M. P. Robillard, “Clonetracker: tool support for code clone management,” in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 843–846.

[7] M. de Wit, A. Zaidman, and A. van Deursen, “Managing code clones using dynamic change tracking and resolution,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2009, pp. 169–178.

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

[9] L. Aversano, L. Cerulo, and M. Di Penta, “How clones are maintained: An empirical study,” in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2007, pp. 81–90.

[10] J. Harder and N. Göde, “Quo vadis, clone management?” in *Proceedings of the 4th International Workshop on Software Clones (IWSC)*. ACM, 2010, pp. 85–86.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

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

[13] J. Spacco and C. Williams, “Lightweight techniques for tracking unique program statements,” in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2009, pp. 99–108.

[14] G. Canfora, L. Cerulo, and M. Di Penta, “Ldiff: An enhanced line differencing tool,” in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 595–598.

[15] S. P. Reiss, “Tracking source locations,” in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 11–20.

[16] V. I. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[17] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, “An empirical study on the maintenance of source code clones.” *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[18] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “Gemini: Maintenance support environment based on code clone analysis,” in *Symp. on Software Metrics*. IEEE Computer Society, 2002, pp. 67–76.

[19] N. Göde and R. Koschke, “Frequency and risks of changes to clones,” in *Proceeding of the International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 311–320.

[20] N. Göde and J. Harder, “Oops! . . . I changed it again,” in *Proceeding of the International Workshop on Software Clones (IWSC)*. ACM, 2011, pp. 14–20.

[21] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, “An empirical study on inconsistent changes to code clones at release level,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2009, pp. 85–94.

[22] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter?” in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 485–495.

[23] L. Barbour, F. Khomh, and Y. Zou, “An empirical study of faults in late propagation clone genealogies,” *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1139–1165, 2013.

[24] N. Göde and R. Koschke, “Incremental clone detection,” in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2009, pp. 219–228.