Late propagation of Type-3 Clones

Saman Bazrafshan

Universität Bremen
saman.bazrafshan@informatik.uni-bremen.de

Abstract

Type-3 clones are duplicated source code fragments that span two or more identical sequences of tokens (whitespace and comments are ignored) that form a contiguous source code fragment interrupted by non-identical token sequences. Several studies on the evolution of code clones have been conducted to detect patterns that can help to manage clones [3, 6]. One of those patterns that is assumed to be of special interest is late propagation [1, 2, 4]. In this paper, ways of detecting late propagation in the evolution of type-3 clones are proposed and discussed.

1 Introduction

During the last years, different studies focused on detecting clone patterns that are considered to have a negative impact on code quality and therefore on maintainability of software. Missing or inconsistent propagation of changes to clones is identified as one pattern that may introduce new defects or prevent the removal of existing ones. To find these clone patterns and enable clone management, a series of tools have been introduced—including clone detectors and clone genealogy extractors. Clones reported by a clone detector are generally distinguished according to their level of similarity. Clones that are identical except for comments and whitespaces are called type-1 clones. Type-2 clones extend type-1 clones by tolerating differences in parameters (e.g., variables, identifiers and literals). Type-3, moreover, allow the insertion and deletion of statements. For a general overview of clone research, please refer to [5]. In this paper, we will name type-1 clones as identical clones and type-2 and type-3 clones as near-miss clones.

An important aspect of clone management is that not all detected clones are equally relevant. To filter relevant clones out of the large number of clones reported by a clone detector, it is of advantage to extract and analyze the evolution of clones—also called clone genealogy [3, 6]. One evolution pattern that has been studied in recent studies is the late propagation. The late-propagation pattern denotes a change to one or more fragments of a clone class that is not propagated to all fragments of the clone class at the same time. Considering defect-correcting changes, late propagation pattern is an indicator that code clones were not intentionally changed inconsistently [1, 2, 4].

2 Late Propagation of Near-Miss Clones

The definition of a late propagation regarding identical clones is straightforward: an inconsistent modification of an identical clone causing the fragments to be non-identical until another inconsistent change to the fragments makes them identical again. However, the definition is not suitable for near-miss clones because they are not completely identical–changes between the identical and the non-identical parts have to be differentiated. The challenging question that arises from this fact is:

> *What are the essential characteristics of a change that makes an inconsistent change to a near-miss clone consistent at a later point of time?*

One way to define the late propagation pattern for near-miss clones is to focus exclusively on the identical parts of a clone disregarding the gaps as the gaps are already not common between the cloned fragments. In this case, we would regard a near-miss clone to be changed consistently if the identical parts undergo the same modifications and continue to be identical–analogously to the definition of a late propagation of identical clones. Hence, to recognize an inconsistent change to a near-miss clone that makes a preceding inconsistent change to the same clone consistent at a later time, all deltas of the clone fragments have to be remembered and compared to every new inconsistent change. Considering a clone class with more than two clone fragments, a gap between two fragments might exist at a different position, in a different form (e.g., different in size), or even not present at all compared to the other fragments of the same clone class. For this reason, it has to be taken into account that a change might hit a gap regarding one or more fragments but an identical block with respect to the other fragments of the clone class. In addition, it is possible that an inconsistent change makes more than one preceding inconsistent change to various fragments consistent at once. Thus, different combinations of deltas have to be compared against an inconsistent change for a sufficient analysis of the consistency of all fragments

belonging to the same clone class. These considerations suggest that detecting late propagations in large near-miss clone genealogies with a high number of inconsistent changes is not feasible using an approach based on the comparison of fragment changes because of the costly comparison strategy.

Defining the late propagation pattern for near-miss clones based on a clone class rather than on its fragments helps to overcome the problem of the cost-ineffective comparison of fragment changes. Instead of focusing on consistent or inconsistent changes regarding clone fragments, a clone class is extended by a state attribute that provides information about the clone class being consistent or inconsistent. A consistent state is defined as follows: At the time of its creation, a clone class is in a consistent state. In the course of the evolution of its clone fragments, a clone class is in a consistent state if it contains at least all clone fragments that have been part of the same clone class the last time it was known to be in a consistent state. The only exception to this rule are fragments that were deleted by the deletion of a whole file. This exception is made because the deletion of source code by a file deletion is not considered to be related to an intended removal of a clone fragment. To make sure we do not handle a moved or renamed file as a deleted one we use the log information of software repository tools that enable file mappings between moved and renamed files between two versions. This way, the deleted clone fragment does not prevent its former clone class to get consistent again. In contrast to a removal of a clone fragment by a file deletion, a deletion of a fragment itself is considered to be a conscious decision to remove the clone fragment. Hence, a clone class that once contained such a clone fragment can not get in a consistent state again. Note, that the information about deleted clone fragments is based on the applied clone detector. Figure 1 shows the evolution of two clone classes throughout three versions of a software system. The left clone class containing the clone fragments a, b, c is in a consistent state in version V_i. In version V_{i+1}, the clone fragment c is not part of the clone class anymore—c was not deleted by a file deletion. Because of the missing fragment c that was part of the clone class the last time it was in a consistent state, namely, in version V_i, the clone class is in an inconsistent state in version V_{i+1}. After the fragment c returns to be part of the clone class in version V_{i+2} it is back in a consistent state. The switch from an inconsistent state back to a consistent one indicates a late propagation considering our definition of a late propagation for near-miss clone genealogies. The right clone class illustrates the above mentioned exception regarding a clone fragment being deleted by a file deletion. Although the clone fragment f, that was part of the clone class in version V_i, is not part of it in the following two versions, the clone class remains in a consistet state. In addition, it can been

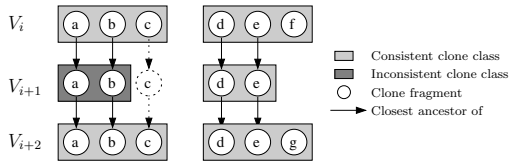

Figure 1: Evolution of two clone classes throughout three versions of a software.

seen that a newly introduced fragment g in version V_{i+2} does not change the state of the clone class from consistent to inconsistent because it is only required that at least all fragments are contained by the clone class that have been contained in the same clone class the last time it was in a consistent state.

One might argue that regarding the changes to clone fragments as black boxes instead of doing a more precise analysis of the changes might not be sufficient enough to be considered an indication for a possible late propagation.

3 Conclusion

Currently there is no study in clone research–to the best of our knowledge–that concentrates on adapting the late propagation pattern of identical clones to near-miss clones. In this paper, we propose two different approaches that might be adequate to detect late propagations of near-miss clones and discuss their possible shortcomings. This work is in progress and right now we are working on the implementation of the approaches. Afterwards an evaluation of both approaches using different software systems is planned. Besides a quantitative study of each approach, we intend to compare the results to existing studies of the late propagation pattern in genealogies of identical clones. By comparing our results to existing ones that are generally assumed to be valid, we expect to be able to draw some conclusions regarding differences of late propagations in idetical and near-miss clone genealogies.

References

[1] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proc. of 11th European CSMR*, pages 81–90. IEEEPress, 2007.

[2] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. of the 27th ICSM*, pages 273–282. IEEEPress, 2011.

[3] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proc. of the 10th ESEC/FSE*, pages 187–196. ACM, 2005.

[4] H. H. Mui. Studying late propagations using software repository mining. Masters thesis, Delft University of Technology, 2010.

[5] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queens University at Kingston, Ontario, Canada, 2007.

[6] R. K. Saha, C. K. Roy, and K. A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proc. of the 27th ICSM*, pages 293–302. ACM, 2011.