Modeling Clone Evolution

Jan Harder, Nils Göde
University of Bremen, Germany
`http://www.informatik.uni-bremen.de/st/`
{harder, nils}@informatik.uni-bremen.de

Abstract

During the maintenance of a program, not only the source code but also the code clones contained in it evolve. Some recent studies focused on detecting patterns in the history of code clones to evaluate the harmfulness of clones.

Since clone evolution is a rather novel field of study, there is still need for more comprehensive models and improved methodologies. Current approaches are limited to detecting only a small specialized set of patterns in a clone's evolution, they generally lack scalability and the way they collect data has to be discussed.

We provide an overview of existing methods to model clone evolution by describing the patterns that can be observed and the respective detection procedure. We discuss their shortcomings and point out open questions towards more detailed models of clone evolution.

1 Introduction

Passages of duplicated source code — *clones* — have an effect on the maintainability of a software system. Empirical research suggests that some clones require higher attention than others [4, 5, 6, 7]. To evaluate the harmfulness of a clone, it is not sufficient to observe its occurrence in a single version of a program. Possible negative effects on the quality of the code, like increased maintenance costs or a higher likelihood of bugs, can only be observed when clones are traced over time and correlated to other parameters from the project's history (e.g. faults).

Several recent studies investigated clones with regard to their evolution. They build models of clone evolution that allow for the investigation of specific properties of a clone. While giving important insights, their results do not add up to a concluding view on the effects of clones, yet. This might be due to the studies set up and the way they obtain and evaluate their data. So far, their scope has been limited to smaller systems or few versions of the software and no model has been proposed that aims to capture the entire version history of a clone.

Up to now, clone evolution research was mainly directed to retrospective studies that investigated the past evolution of software systems. Another application that could greatly benefit from clone evolution data is clone management. When a clone's evolution is tracked as it happens, developers could be notified of inconsistent changes to clones and predictions of a clone's future severity could be made. This means that clone management could greatly benefit by models of clone evolution that capture all properties and events in the history of clones that could indicate future problems before they arise. This calls for more detailed models of clone evolution but at the same time more scalable techniques to compute them.

Contributions. In this paper we will outline the relevant studies on clone evolution in Section 2. We discuss their shortcomings and state open issues towards more comprehensive models of clone evolution in Section 3. Section 4 concludes.

2 Modeling Clone Evolution

Several approaches to model a clone's evolution were proposed in the past. In this section we will give an overview of the models they define and their methodology to compute them. We focus on those studies that actually track clones across versions. Methods that quantify properties of cloned code but do not follow individual clones are not in the scope of this paper.

Note, that the field of code clone research has not employed a consistent terminology, yet. We use the following terms throughout this paper: A *clone fragment* is a passage of source code, that is sufficiently similar to at least one other passage of source code. The measure of similarity depends on the specific scenario and clone detection method used. Each fragment belongs to a *clone class*. Such a class contains all fragments that are sufficiently similar to each other. We will also refer to *versions* of source code, that we define as a snapshot of a program's source code at a given point of time.

All approaches that model the evolution of clones need to deal with two major challenges: (1) The clone fragments from one version v_i must be mapped to the fragments of the following version v_{i+1}; and (2) clone evolution patterns must be defined and calculated upon the cross-version correlation of the clone fragments or clone classes.

The first study of this kind was carried out by Kim et al. [5]. They implement the fragment mapping in two functions: One calculates the *location overlapping* of two fragments. Therefore the positions of the fragments in v_{i+1} are realigned to their former positions in v_i using *diff*. Then the relative proportion of the overlapping region is computed as a score. The other metric rates the *textual similarity* of both fragments as the ratio of duplicated code between them. For the evolution of clone classes they define evolution patterns as rules in first order logic. Predicates are the membership of fragments in clone classes (which are sets of fragments) as well as the two metrics mentioned before. That way they formulate five essential clone class evolution patterns that we will explain briefly. The *Same* pattern denotes that all fragments of a class remained the same and no change took place. *Add* describes the case where at least one fragment appears in a clone class that it was not part of in the previous version. Its counterpart *Subtract* models the case that at least one fragment vanished from a clone class. Two further patterns *Consistent Change* and *Inconsistent Change* are introduced to model changes to the fragments' lexical structure. A change is *consistent* if all fragments of the clone class have changed in the same way. It is *inconsistent* if fragments of the clone class have changed differently. This includes the case where only some fragments were changed while others where not. Each detected pattern connects a clone class from v_i and v_{i+1}. While a clone class is allowed to have more than one successor, it only has one predecessor in general, being the one with the best location overlapping (there can be two successors in the case of an ambiguity). Together the clone classes and their mappings form a graph called *clone genealogy* that describes the evolution of clone classes. Kim et al. extracted such genealogies for two small Java systems (*dnsjava* and *carol*) to investigate the lifetime of clone classes. It turned out that many clone classes are short-lived and therefore aggressive removal is not practical as a universal strategy for clone management. At the same time many of the long-living clone classes seem to remain in the source code because removal is not possible due to limitations of the programming language or other constraints.

Aversano et al. picked up Kim's idea of evolution patterns, but chose a semi-automated approach to detect them [1]. They first detect clone classes in a single version and then extract all *modification transactions (MT)*, that changed the cloned code, from the projects version control system. The MTs are calculated upon CVS commit operations using a sliding window technique. Each of them is regarded as a version step. Furthermore they only consider clone classes that are changed inconsistently in at least one MT and whose fragments scatter at least across two different files. Upon these data a refined subset of Kim's patterns is detected. Only consistent and inconsistent changes are taken into account. The inconsistent change pattern is further specialized into *Independent Evolution* and *Late Propagation*. Evolving independently means that fragments that belonged to the same class are changed inconsistently and further develop independently. In a *Late Propagation* two fragments belong to the same class, are then changed inconsistently so that they appear in different classes (or not at all) for some time and then appear again together in the same class. This phenomenon is believed to be a typical risk related to cloning. A programmer might forget to change one or more fragments of a class and performs the missing changes belatedly after an error occurred. It is also possible that changes on some fragments were not forgotten but performed differently by accident. The pattern detection was left to human reviewers who inspected all classes and the corresponding MTs. In two case studies on *ArgoUML* and *dnsjava* they conclude that developers carry out changes consistently in most cases. A considerable amount of changes falls into the *Independent Evolution* category, while *Late Propagations* are much more infrequent. They also conclude that clones at a higher granularity (class clones vs. method clones) more often change consistently.

An approach, that uses information from an abstract syntax tree (AST) was proposed by Bakota et al. [2]. Before matching fragments across versions, they eliminate all possible matches of fragments whose AST subtree representation has a different type of root node. For each remaining pair of fragments, they calculate a similarity metric that is an aggregation of five weighted metric values. These are (in descending priority): (1) the lexical similarity of the fragments' source code, (2) textual similarity of the first named node found in the fragments' AST, (3) the order in which the fragments were reported, (4) the relative position within the enclosing syntactic unit and (5) the similarity of the names of the files containing the fragments. All textual similarity measures base on the Levenshtein distance. Metric (3) is the numerical difference of the position in the clone fragment list of the class the fragments belong to, (4) calculates a distance between the two AST subtrees. For all remaining fragment pairs the similarity metric is calculated and for each fragment in v_i the best match in v_j $(i < j)$ is chosen as descendant. This leads to a partial injective mapping between the fragments of the examined versions. That is, each fragment has at most one descendant and predecessor. As fragments can appear or vanish from one version to another, not all of them are mapped. Based upon the mapping, the authors define four evolution patterns which they call *clone smells*. Bakota et al. do not aim to classify all changes to clone classes. Instead they try to identify changes of cloned fragments that suggest a negative impact on code quality. Such a smell is reported whenever a clone fragment vanishes or occurs for the first time. *Moving* and *migrating* clone fragments are more complex smells. These model modifications to the partitioning of fragments into clone classes across versions. Fragments are said to *move* when they appear in a class together with different fragments compared to the previous version. A *migration* can be compared to Aversano's *Late Propagation*. It is modeled as the case that two fragments appear in the same clone class in one version, then they do not appear in the same class for at least one later version, just to appear again together in one class. Bakota et al. applied their methodology on twelve versions of *Firefox* with a monthly stepping and extracted 60 smells out of which 28 have been rated as false positives. Manual inspection showed that 5 smells could be related to possible bugs and 6 to confirmed bugs. Further 8 smells resulted from bugfixes that corrected inconsistent changes made in earlier versions than the ones inspected by the study.

Krinke conducted a study on consistent and inconsistent changes to clone classes [6]. Like Aversano et al. he studied changes to the clones of one representative version. First he employs a pretty printer on the source code of all versions and detects the clone classes in v_i using *Simian*. He then collects the changes between a pair of versions v_i and v_j $(i < j)$ where *diff* serves him as a tool. From all changes, those that do not overlap clone fragments are discarded. On this basis, consistent and inconsistent changes can be identified by comparing the changes applied to the fragments of one class. When these equal, a *consistent* change took place between the two versions. If they differ or if some fragments were changed but others were not, the change was *inconsistent*. Krinke uses this methodology to evaluate how often consistent and inconsistent changes happen. He further investigated whether cases exist where inconsistent changes turn to consistent ones, when the interval between the versions is enlarged (this corresponds to the *late propagation* pattern). He concludes that the number of consistent and inconsistent changes is about the same for any version interval. Late propagations rarely were observed in the systems he investigated.

3 Discussion

The studies we presented pioneered the field of models for clone evolution. Many open questions on how evolution should be captured, how further conclusions on clone evolution can be made an how future requirements to such models can be met remain. In this section we will discuss the shortcomings of existing approaches and outline the open questions that are most important in our opinion.

Patterns of clone evolution. There seems to be consensus on a basic set of patterns for clone evolution. Although named differently, most studies define similar evolutionary events as being relevant. These patterns can be categorized into three levels by the elements they match and the number of versions they span. On the lowest level there are fragment patterns that model the *occurrence* and the *vanishing* of fragments as well as *changes* to them between two versions. The next higher level models changes to clone classes between two versions. Here the patterns by Kim et al. *Same*, *Add*, *Subtract*, *Inconsistent Change* and *Consistent Change* constitute the common basis. The highest level form clone class patterns that span across more than one version step. The most prominent representative is *Late Propagation*, but also Aversano's *Independent Evolution* belongs here. Yet, no model comprises all of these patterns. The existing approaches either lack the high-level ones [5] or choose a narrow focus and use shortcuts to come to a subset of the patterns directly [1, 2, 6].

We believe that a model that computes all kinds of patterns by aggregating the higher from the lower ones would be beneficial for two reasons: First, starting on a low level leaves enough flexibility to identify and model additional patterns that might not have been considered yet. These cannot be found by studies that explicitly focus on a predefined set of patterns. Secondly, a complete trace of a clone's evolution could support clone management. As stated before this could be a basis for metrics that rate a clones severity by means of its bygone evolution.

Question 1 *How can we build more comprehensive models of the evolution of clones?*

Question 2 *Are there more patterns, that are relevant to the impact of clones on code quality?*

Mapping clones. Despite of using different techniques, the proposed models have in common that they start off mapping fragments of the detected clone classes across versions. However, the way this mapping is further processed differs a lot. It can be embedded in rules that map clone classes from one version to another [5], used to define patterns upon fragments and their membership in clone classes [2] or to compare changes of fragments without considering clone classes of later versions at all [6].

These choices have a vast impact on which evolutionary events can be observed and which cannot. One difficulty here is that cloned fragments are defined in relation to each other. They do not have a meaning outside this context and cannot be considered as single units. This becomes obvious when regarding the *Late Propagation* pattern with classes consisting of only two fragments. After the initial inconsistent change, the class disappears because it now has only one fragment, which is not a clone anymore as the other fragment has changed. When the change is also made to the other fragment in a later version, the class re-appears, because both fragments are again clones of each other. The same happens with late propagations on larger clone classes, where only a single fragment is 'forgotten'. The single fragment will be invisible to clone detectors while it is in an inconsistent state. These problems indicate, that it does not suffice to only map fragments of one version to the next. Instead, it is necessary to keep track of them across multiple versions. Some studies try to circumvent this problem by mapping fragments across every pair of versions and not only consecutive ones [2, 6]. Unfortunately, these approaches do not scale well. Furthermore they are not capable of mapping clone classes across versions.

Question 3 *How can single fragments be tracked efficiently even when they are invisible to clone detection temporarily?*

Question 4 *Can clone class mappings be combined with such a fragment tracking?*

Choice of interval. The interval at which versions of a program are analyzed has a considerable impact on the results. Intervals chosen too long might blur the results and hide important detailed patterns. For example, every late propagation is reported as consistent change if the interval is chosen long enough. Short intervals on the other hand can lead to a high number of inconsistent changes reported. Note that there is no consistent change at all if the interval is chosen short enough, because changes to different fragments cannot be performed simultaneously.

For very fine grained stepping, this means that a consistent change pattern should be allowed to span more that just one step. However this raises the problem that some boundary between consistent changes and late propagations must be defined. In contrast, a wider stepping leaves the categorization of the change to chance, because it depends on the point in time where the version snapshot was taken.

Although the choice of interval often depends on the availability of the source code, we believe that more thought needs to be spent on choosing a sensible interval and investigating the influence on the results.

Question 5 *How can we discriminate between Consistent Changes and Late Propagations?*

Evaluation of mappings. Despite the increasing number of models for clone evolution, their evaluation has not gained much attention, yet. So far, only Bakota et al. defined a quality criterion. It states to what extend two mappings between the fragments of two versions v_n and v_m, that vary in their version stepping, differ. One of the two maps the fragments of v_n and v_m directly, the other one incorporates additional versions between v_n and v_m. The second mapping bases on more detailed data and should therefore be more accurate. The goodness of the direct mapping in respect to the detailed one is then measured as the number of differences between the mappings from v_n and v_m (for the detailed mapping the composition of all mappings from v_n to v_m is used).

While this is a reasonable criterion to compare mappings with different stepping, it does not make any statement on the mapping's precision and recall compared to a human oracle. A bypass to this problem is inspecting all results manually [1, 5]. Not all studies spend this effort and confine themselves to inspecting only patterns of interest [2] or do not state how the computed evolution model was evaluated [6].

Evaluating the mappings should be regarded as an open issue, especially for studies on larger code bases where manual inspection of all results is out of question.

Question 6 *How do we define a mapping's correctness and how do we measure it?*

Clone detection approach. The quality of the results obtained from clone evolution analyses depends to a large extent on the underlying clone detection approach. Available detection methods are known to lack precision and recall. If the clone detector reports many false positives that are taken into account for quantitative measures, the validity of the results is doubtful. To improve precision, an AST-based detection algorithm is used [2] or only type-1 clones are taken into consideration [6]. Another way is manually inspecting clone candidates and discarding false positives [1, 5].

The influence of these countermeasures and whether the results are still valid has not yet been investigated. Furthermore, the scalability of discarding false positives by manual inspection is not given.

While AST-based clone detection alleviates some common problems like false positives and fragments partially overlapping different syntactic regions, the downside is that it cannot deal with most code that contains preprocessor directives which is the case for most large-scale C and C++ systems. Many of these tools are also limited to the analysis of Java code and indeed Java is the predominant language among the systems investigated regarding clone evolution.

Question 7 *How do we deal with false positive clones?*

Scalability. Like with any other algorithm, it is desirable to obtain results in appropriate time also for large systems. Concerning the papers we have considered, the amount of time required to produce results is relatively high. On the one hand, this is due to manually inspecting intermediate results [1, 5]. On the other hand, matching clones of one version to the next has quadratic complexity. The effort can be reduced by presorting clones before matching them [2].

There is reason to believe that clones show different properties in software of different scale and therefore modeling techniques should be able to deal with large systems. Our opinion is the larger the code base gets, the more issues arise that could lead to problematic changes to clones. Plenty of code makes it difficult, if not impossible to keep the whole source code in mind, whereby the chance that clones are overlooked, when a change is made, raises. It also seems plausible that a larger number of developers meets more complex interrelations that might cause negative effects by clones. A study performed by Balint et al. suggests that clones that were introduced by one person could be changed inconsistently by another one because the cloning relationship is not documented [3].

Many clone evolution studies stick with smaller systems because their approaches lack scalability. Only Bakota et al. analyze a program exceeding one million lines of code, but at the same time they merely analyze twelve versions. Scalable and automated methods are needed to broaden clone evolution studies on such systems.

Question 8 *How can techniques to compute evolution models be made more scalable?*

4 Conclusion

Researchers agree that the history of code clones needs to be considered for evaluating their impact. We described different models of clone evolution that have been presented in previous works. We also outlined the respective methods to extract specific evolution patterns from a program's history.

Based on the shortcomings of the individual approaches, we identified some general questions that have not been answered yet. We do, however, believe that answering these questions is essential for drawing conclusions from the results of clone evolution analysis.

References

[1] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *European Conference on Software Maintenance and Reengineering*. IEEE CS Press, 2007.

[2] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *International Conference on Software Maintenance*, pages 24–33. IEEE CS Press, Oct. 2007.

[3] M. Balint, R. Marinescu, and T. Girba. How developers copy. In *International Conference on Program Comprehension*, pages 56–68, Washington, DC, USA, 2006. IEEE Computer Society.

[4] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Working Conference on Reverse Engineering*, pages 19–28, 2006.

[5] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE*, pages 187–196, 2005.

[6] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Working Conference on Reverse Engineering*. IEEE CS Press, 2007.

[7] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Mining Software Repositories*, ICSE Workshop. ACM Press, May 2007.