An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies

Ripon K. Saha   Chanchal K. Roy   Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{ripon.saha, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—Extracting code clone genealogies across multiple versions of a program and classifying them according to their change patterns underlies the study of code clone evolution. While there are a few studies in the area, the approaches do not handle near-miss clones well and the associated tools are often computationally expensive. To address these limitations, we present a framework for automatically extracting both exact and near-miss clone genealogies across multiple versions of a program and for identifying their change patterns using a few key similarity factors. We have developed a prototype clone genealogy extractor, applied it to three open source projects including the Linux Kernel, and evaluated its accuracy in terms of precision and recall. Our experience shows that the prototype is scalable, adaptable to different clone detection tools, and can automatically identify evolution patterns of both exact and near-miss clones by constructing their genealogies.**

***Index Terms*—clone genealogy extractor; mapping; clone evolution.**

I. Introduction

The investigation and analysis of code clones has attracted considerable attention from the software engineering research community in recent years. Researchers have presented evidence that code clones have both positive [10], [22] and negative [16] consequences for maintenance activities and thus, in general, code clones are neither good nor bad. It is also not possible or practical to eliminate certain clone classes from a software system [10]. Consequently, the identification and management of software clones, and the evaluation of their impact has become an essential part of software maintenance. Knowing the evolution of clones throughout a system's history is important for properly comprehending and managing the system's clones [9].

There has been quite a bit of research on studying code clone evolution. Most of these studies investigate retrospectively how clones are modified by constructing a clone genealogy. A clone genealogy describes how the code fragments of a clone class propagate through versions during the evolution of the subject system. Therefore, accurately mapping clones between versions of a program, and classifying their change patterns are the fundamental tasks for studying clone evolution.

Researchers have used three different approaches to map clones across multiple versions of a program. In the first approach [2], [10], [20], clones are detected in all the versions of interest and then clones are mapped between consecutive versions based on heuristics. In the second approach [1], clones are detected in the first version of interest, and then they are mapped to consecutive versions based on change logs provided by source code repositories such as *svn*. In the third approach [15], [6], clones are mapped during clone detection based on source code changes between revisions. A combination of the first and second approaches has also been used in some studies [3].

Although intuitive, each of these approaches has some limitations. In the first approach, a number of the similarity metrics used to map clones have quadratic time complexities [9]. In addition, if a clone fragment changes significantly in the next version and goes beyond the given similarity threshold of the clone genealogy extractor, a mapping may not be identified. In the second approach, only clones identified in the first version are mapped. Therefore, we do not know what happens to clones introduced in later versions. The third approach ("incremental approach") avoids some of the limitations of the previous two approaches by combining detection and mapping, and works well for mapping clones in many versions. By integrating clone detection and clone mapping this approach can be faster than the approaches that require clone detection to be conducted separately for each version. Although this incremental approach is fast enough both for detection and mapping for a given set of revisions, it might not be as beneficial at the release level [6] because there might be a significant difference between the releases. Furthermore, in the sole available incremental tool, *iClones* [6] (available for academic purpose), when a new revision or release is added for mapping, the whole detection and mapping process should be repeated since clones are both detected and mapped simultaneously. Clone management is likely being conducted on a changing system, and it is a disadvantage for an approach to require detecting clones for all revision/versions each time a new revision/version is added. Another issue with the incremental mapping is that it cannot utilize the results obtained with a classical non-incremental clone detection tool as the detection of clones and their mapping is done simultaneously. Most of the existing clone detection tools are non-incremental. There is also no representative tool available. Depending on the task at hand and the availability of tools, one might want to study cloning evolution with several clone detection tools. It is thus important to have a clone evolution analysis tool in place independent of the clone detection tools. Scalability of the incremental approaches is another great challenge because of huge memory requirements.

Again, while most of these approaches [1], [2], [3], [10], [20] are based on the state of the art detection and mapping techniques, they only focused on Type-1 and Type-2 clones.

Literature [19] shows that there are a significant number of Type-3 clones (where statements might be added, deleted and modified in the copied fragments) in software systems and thus extracting the genealogies of such clones and understanding their change patterns is equally important.

In this study, we propose a framework for extracting both exact (Type-1) and near-miss (Type-2 and Type-3) clone genealogies across multiple versions of a program, and identifying their change patterns automatically. The framework works with any existing clone detection tool that represents a clone fragment by its file name and begin-end line numbers. Genealogies are constructed incrementally by merging current mapping results with previously stored genealogies to give a complete result. To validate the effectiveness of our proposed framework, we developed a prototype clone genealogy extractor (*CGE*), extracted both exact and near-miss clone genealogies across many releases of three open source systems including the Linux Kernel, adapted the *CGE* with other clone detection tools, and evaluated the correctness of the mappings reported by the prototype in terms of precision and recall. We also compared our result qualitatively and quantitatively with a result of an incremental clone detection tool, *iClones* [6]. The experimental results suggest that the proposed framework is scalable and can identify the evolutionary patterns automatically by constructing genealogies for both exact and near-miss clones. We name our prototype as gCad.

The rest of this paper is structured as follows: Section II briefly describes the model of clone genealogy. In Section III we describe the proposed framework, whereas Section IV outlines the details of our evaluation procedure. In Section V we compare our method with others. Section VI discusses the threats to the validity of our work. In Section VII we discuss the related work to ours, and finally, Section VIII concludes the paper with our directions for future research.

II. Model of Clone Genealogy

Kim et al. [10] defined a model of clone genealogy that describes how each fragment in a clone class changes from version to version with respect to other fragments in the same clone class. Each clone genealogy consists of a set of clone lineages that originate from the same clone class. A clone lineage is a directed acyclic graph that describes the evolution history of a clone class from the final release of the software system to the version in which it originated. Based on the change information and the number of fragments in the same clone class in two consecutive versions, Kim et al. identified six change patterns for evolving clones. We followed their model of clone genealogy to design our framework. However, we have adapted the definitions for some of the change patterns since our primary objective is to deal with Type-3 clones.

Let cc^i be a clone class in version v_i. If cc^i is mapped to cc^{i+1} in v_{i+1} by a clone genealogy extractor, then the evolution is characterized as follows (with Figure 1):

Same: each clone fragment in cc^i is present in cc^{i+1}, and no additional clone fragment has been introduced in cc^{i+1}.

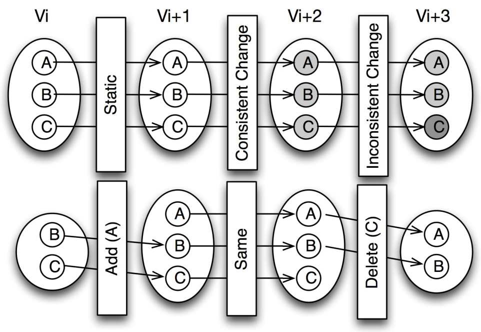

Fig. 1. Various evolutionary patterns of code clones

Add: at least one clone fragment has been introduced in cc^{i+1} that was not part of cc^i.

Delete: at least one of the clone fragments of cc^i does not appear in cc^{i+1}.

Static: the clone fragments in cc^{i+1} that were also part of cc^i have not been changed.

Consistent Change: all clone fragments in cc^i have been changed consistently, and thus all of them are again part of cc^{i+1}. However, a clone class may disappear even though it has been changed consistently. This could happen when the resulting fragments of that clone class go beyond the minimum line/token numbers set by a subject clone detection tool.

Inconsistent Change: all clone fragments in cc^i have not been changed consistently. Here we should note that for Type-3 clones where the addition and deletion of lines are permitted, all the clone fragments of a particular clone class could still form the same clone class in the next version even if one or more fragments of that class have been changed inconsistently. The dissimilarity between the fragments of a clone class usually depends on the heuristics/similarity threshold of the associated clone detection tools.

III. The Proposed Framework

In this section we discuss our proposed framework to build a near-miss clone genealogy extractor (*CGE*). Usually a *CGE* accepts n versions of a program, maps clone classes between the consecutive versions, and extracts how each clone class changes throughout an observation period. A version may be a release or a revision. The approach has four phases: (1) Preprocessing (2) Function Mapping (3) Clone Mapping, and (4) Automatic Identification of Change Patterns. Figure 2 shows how these steps work together to construct clone genealogies of a software system. At first we describe a naïve version of our *CGE* that maps clones between two versions of a program. Then we will describe a complex *CGE* that can deal with n versions of a program, and can work with various clone detection tools.

A. Preprocessing

For two given consecutive versions, v_i and v_{i+1} of a software system, first we extract all the function signatures from both the versions. For extracting functions we use TXL [4], a special-purpose programming language that supports lightweight agile parsing techniques. We exploit TXL's extract function, denoted by [^], to enumerate all the functions. For

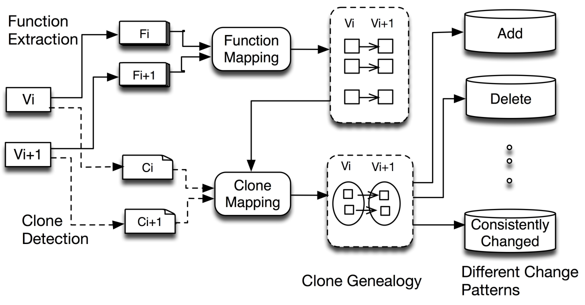

Fig. 2. The proposed framework for two versions of a program

TABLE I
CHANGE SCENARIOS OF FUNCTIONS

Scenarios	Signature	Body	Location
1	not changed	not changed	not changed
2	not changed	not changed	changed
3	not changed	changed	not changed
4	not changed	changed	changed
5	changed	not changed	not changed
6	changed	not changed	changed
7	changed	changed	not changed
8	changed	changed	changed

each function we store the function signature, class name, file name, the start and end line number of the function in the file, and its complete directory location in an XML file. Certainly, one can use a tool other than TXL for extracting this required information. Once the preprocessing is completed for a particular version V_i, there is no need to preprocess it again.

B. Function Mapping

Once the preprocessing is done, we map the functions of v_i to those of v_{i+1} instead of mapping their clones. A function is the smallest unique element of a software project if we consider the signature of a function along with its class name and complete file path. Therefore, we can use these attributes as a composite key to map functions between two versions. There are already a few great studies [23], [24] where program elements are mapped based on the function and file names. However, in practice some functions could be renamed, or could move to different files or directories during the evolution of the system. Table I summarizes all possible change-scenarios of a function during evolution. We have two advantages of mapping the functions before mapping the clones. First, a very few functions are renamed in the next version compared to the total number of functions in a version [7], [11]. Therefore, most of the functions could be mapped by only using their signatures which is computationally very fast. Second, if a function name has not been changed in the next version we can accurately map them even if the body of the function has been changed significantly. This overcomes the disadvantage of threshold based mappings that are dependent on the text similarity of two functions. In the following subsections, we discuss how we adapted and optimized some established function mapping techniques [23], [24] to match functions that are common between two versions, v_i and v_{i+1} of a program.

1) When function names remain the same: This part of the algorithm maps those functions for which the function names have not been changed in the next version (change scenarios 1-4 in Table I). For each function, if its name occurs once in v_i and once in v_{i+1}, it is considered the same function without considering any further information. Thus even if the function has been moved to another file or directory, our algorithm would map it correctly. One might argue that a function *foo* might be deleted from v_i and a new function having the same name could be inserted in the next version. However, our experience suggests that such circumstances seem to occur rarely. On the other hand, if two or more functions exist having the same name in either one or both versions of the program, our algorithm maps them applying the following rules. For each of the functions we check the locations of the functions. If they are from the same location, then their signatures must be different, and they will be matched by their signatures. In contrast, if they are from different locations, then their locations will be used to resolve the mapping. For example, sometimes it might happen that the method signatures and file names both are the same. It indicates that they are from different packages/directories. In this situation we will consider the hierarchy level of their directory until we find different names to map them properly.

2) When function names have changed: This part maps those functions, which have been renamed in the next version (change scenarios 5-8 in Table I). From the previous step, the functions (D) of v_i that have not been mapped into any function in v_{i+1} are the possible candidates for being renamed. That is, if O is the set of function names of v_i, and N is the set of function names of v_{i+1}, $D = O - N$. Similarly, the set of either new or renamed functions (A) in v_{i+1} is $N - O$. Therefore, we need to find an appropriate mapping between D and A. To find an appropriate mapping between D and A, we use Kim et al.s' [11] automatic detection of origin analysis method. Kim et al. introduced eight similarity factors to determine whether a function has been renamed or not. They used all possible combinations of those factors to find the renamed functions, and report that adding more factors does not necessarily improve the accuracy of the origin detection. They also noted that 90.2% of renamed functions could be accurately mapped using only function name and body, whereas the best accuracy was only 91% using more factors. By further investigating their result we noticed that the average number of renamed functions per version is very few; less than 2%. Therefore, we use only the function name and body to find the origin of functions in the set A because we can achieve a considerable performance gain by sacrificing only 0.8% accuracy for the renamed functions, which is negligible (0.016%) compared to the total number of functions.

We use the longest common subsequence count (*LCSC*) similarity metrics to find the origin of a function because only checking common parts of two fragments (function body) or two names (function names) is not sufficient. We also have to check that the common part also maintains the same sequence as its origin. We use equation 1 to calculate *LCSC* similarity metrics of two fragments, A and B where $|A|$ and $|B|$ are the number of elements in A and B respectively. It returns a value between 0 and 1 where 0 means no similarity whereas 1 means exactly the same. LCSC calculates name similarity at the character level and text similarity at the line level. We use comment-free pretty-printed lines to ignore formatting or commenting differences. Now for two functions $A \in v_i$ and $B \in v_{i+1}$, we identify A as the origin of B if the average of $(LCSC\ Similarity(name_A, name_B)$ and $LCSC\ Similarity(body_A, body_B))$ is greater than 0.6 which was found as one of the best thresholds by Kim et al [11]. If we find more than one possible origins, we take the one that gives the highest average. However, we did not consider any split or merging of functions during function mapping.

$$LCSC\ Similarity = \{\frac{LCSC_{AB}}{|A|} + \frac{LCSC_{AB}}{|B|}\}/2 \tag{1}$$

C. Clone Mapping

At this point we have the function mappings between versions v_i and v_{i+1}. We also assume that the clones have already been detected from these two versions using a clone detection tool. Typically a clone detection tool reports results as a collection of clone classes where each clone class has two or more clone fragments. A clone fragment could be of any granularities such as function, structural block, arbitrary block and so on. Let $CC^i = \{cc_1^i, cc_2^i, ..., cc_n^i\}$ be the reported clone classes in v_i where $cc_j^i = \{CF_{j1}^i, CF_{j2}^i, ..., CF_{jm}^i\}$. Here CF_{jk}^i refers to the clone fragments of the clone class cc_j^i where $1 \leq k \leq m$. Now the algorithm attempts to map each clone fragment CF^i to its contained (parent) function, F^i in v_i. We say that a clone fragment (CF) is contained by a function (F) if both are in the same file, and the range of line numbers of (CF) is within the range of line numbers of F. In algorithmic form,

```
boolean isContained(Block CF, Function F) {
    return ((CF.FileName == F.FileName)
            AND (CF.BeginLine >= F.BeginLine)
            AND (CF.EndLine <= F.EndLine))}
```

We should note that we already have the required information (i.e., file names and begin-end line numbers) for functions from the *Preprocessing* phase (Section III-A) and for the clone fragments from the subject clone detection tool.

Now let us assume a clone fragment CF^i is mapped with a function F^i in version v_i, and another clone fragment CF^{i+1} is mapped with a function F^{i+1} in version v_{i+1}. Let us also assume that F^i is in fact mapped with F^{i+1} obtained from the Function Mapping phase (Section III-B). If F^i contains only one clone fragment, CF^i we can easily map that clone fragment in v_{i+1} since we already know the corresponding mapped function F^{i+1} in v_{i+1}, and the mapped clone fragment CF^{i+i} should be found in v_{i+1} if it has not been disappeared. However, if F^i has more than one clone fragments, we attempt to map each of the clone fragments between functions F^i and F^{i+1}. For example, if F^i has m clone fragments and F^{i+1} is the corresponding mapped function, we map the clone fragments in two ways. First, we calculate the *LCSC Similarity* (Section III-B2) for each of the clone fragments of function F^i with the clone fragments of function F^{i+1}. Second, based on the similarity scores and the relative locations of the clone fragments in the corresponding functions, we take the final decision for mapping. There might be still some clones that have not been mapped yet. They might be file clones, clones that span more than one functions, clones in declarations, or clones in C preprocessor code. We map such clones by following the method discussed in Section III-B2 with the exception that instead of using function names as one of the attributes, we use file names of the clone fragments. Therefore, we now have the mapping for each of the clone fragments CF_{jk}^i of version v_i to a clone fragment in version v_{i+1} if it has not been deleted in version v_{i+1}. Because the functions of versions v_i and v_{i+1} are already mapped in the *Function Mapping* phase (Section III-B), the mapping of function clones is straightforward, and of course there is no question of mapping blocks to their corresponding functions.

The next step is to map the clone classes of the two versions. For each of the clone fragments CF_{jk}^i of the clone class cc_j^i in version v_i we find the corresponding clone fragment $CF_{j'k'}^{i+1}$ in version v_{i+1}. In principle, if cc_j^i of version v_i remains the same or changed consistently in version v_{i+1}, all the mapped fragments of cc_j^i in v_i should be in the same clone class in v_{i+1}. However, if the consistent changes are made to the extent that the size of the fragments is under the minimum clone size threshold of the subject clone detector, the detector will not detect this clone class. When we deal with Type-3 clones, all these mapped clone fragments of CF_{jk}^i could be found in the same clone class in v_{i+i} even though they have been changed inconsistently but still their dissimilarity is under the given threshold set by the subject clone detection tool. Of course, a clone class may split due to the extensive inconsistent changes between its clone fragments. Therefore, in order to find out the mapping of the clone class cc_j^i in v_{i+1}, we find all the clone class(es) $\{cc_x^{i+1}, cc_y^{i+1}, ...\}$ from v_{i+1} that form clone classes with any of the clone fragments of cc_j^i of v_i. If all the clone fragments are mapped to the same class, cc_x^{i+1}, we map $cc_j^i \rightarrow cc_x^{i+1}$. On the other hand if they are mapped to multiple classes, $\{cc_x^{i+1}, cc_y^{i+1}, ...\}$ we map $cc_j^i \rightarrow \{cc_x^{i+1}, cc_y^{i+1}, ...\}$.

D. Automatic Identification of Change Patterns

Automatic and accurate identification of change patterns is one of the important features of a clone genealogy extractor. There is a marked lack of in-depth study in the literature on this issue [9], especially for near-miss clones. In this paper, we also attempt to deal with this important issue. Among the change patterns (discussed in Section II, the identification of *Same*, *Add*, and *Delete* change patterns are fairly straightforward because these change patterns are classified based on the appearances or disappearances of the clone fragments of a clone class of version v_i in version v_{i+1}. The clone classes of v_i those are not spilt in v_{i+1} are the candidate set for these sort of change patterns. Let us assume that from the *Clone Mapping* phase we obtain a mapping, $cc_j^i \rightarrow cc_{j'}^{i+1}$, where cc_j^i and $cc_{j'}^{i+1}$ have p and q number of clone fragments respectively. Now if $cc_{j'}^{i+1}$ has m clone fragments that have their origins in cc_j^i, we can identify the aforementioned patterns automatically as follows:

- if $m = p = q$ then it is the *Same* change pattern.
- if $q > m$ then it is the *Add* change pattern.
- if $p > m$ then it is the *Delete* change pattern.

Here it should be noted that the *Add* and *Delete* patterns are not mutually exclusive since additions and deletions of cloned fragments in a clone class could take place simultaneously. Unlike the above patterns, the identification of consistent and inconsistent change patterns is challenging. It becomes more complicated when we deal with Type-3 clones since for Type-3 clone classes, all the clone fragments of a particular clone class could form the same clone class in the next version even if one or more clone fragments of that class have been changed inconsistently. Therefore, we cannot conclude whether a particular change pattern is consistent or inconsistent even if all the clone fragments of a clone class in v_i form a clone class in v_{i+1}.

In order to deal with this issue, we use a multi-pass approach that makes decisions in each pass, and identifies consistent and inconsistent change patterns gradually. First, the program identifies the clone classes that have not been changed in the next version at all (*Static*), and those clone classes that have been split. The program identifies the split clone classes as inconsistent change because certainly their fragments have been changed inconsistently, and thus they are part of two or more clone classes in the next version. Identification of these change patterns is straightforward and accurate because there is no ambiguity in selecting them. Fortunately, our program can make a decision for most of the change patterns during the first pass because a large number of clone classes do not change at all. We have shown this phenomena for Type-1 and Type-2 clones at the release level in our previous study [20]. This seems to hold for Type-3 clones as well (see Table IV).

In the second pass, the program will make a decision for Type-1 and those Type-3 clone classes where modifications of different fragments of the same clone class are only limited to line additions or deletions but do not have any variable renaming. If $cc^i_j \rightarrow cc^{i+1}_{j'}$ is such a mapping, the program computes the differences between each of the clone fragments of cc^i_j with the corresponding clone fragments of $cc^{i+1}_{j'}$ using *diff*. If the differences for each of the fragment pairs (CF^i_{jk}, $CF^{i+1}_{j'k'}$) are the same, then the clone class is classified as consistent change, otherwise as inconsistent change. Inconsistent reordering of statements are also considered as inconsistent changes and thus the weakness of *diff* that it cannot detect reordering of statements does not have any impact on our results. One might argue that the changes in the gap in the Type-3 clones should not be considered because they are already not common between the clone fragments. However, the gaps in Type-3 clones could be arbitrary, and a gap between two clone fragments of a clone class might not be the same compared to the other clone fragments of that same clone class. Thus, a change might actually happen in a gap with respect to one fragment (or more fragments) but in the common parts of the rest of the clone fragments, which is important to consider in classifying the change patterns.

In the third pass, the rest of the clone classes (Type-2 clones, and Type-3 clones with identifiers renaming) are considered. Since the clone fragments of these clone classes have variations in their identifiers, we do not exploit *diff* directly because the differences will not be the same even if the fragments have been changed consistently. In order to deal with this issue we consistently rename the identifiers of the clone fragments using TXL. For example, if the first identifier and all its occurrences in a fragment is replaced by x_1, the second identifier and all of its occurrences will be replaced by x_2 and so on. We then compute the differences. As before if the differences are the same we classify the change pattern as consistent change, otherwise as inconsistent change. All of the identified mappings as well as their change patterns are stored in an XML file for future use.

E. Construction of Genealogies

In order to keep the discussion simple, we have described the algorithm only for two versions of a program . Now we extend it for *n* versions to construct the clone genealogies. Let us assume that $CGE(v_i, v_{i+1})$ is the algorithm for only two versions, whereas $CGE(v_1, v_2, ..., v_n)$ is the algorithm for *n* versions. Practically the *n*-version algorithm is the combined result of $CGE(v_1, v_2)$, $CGE(v_2, v_3)$, ..., $CGE(v_{n-1}, v_n)$. Our mapping is thus incremental in nature where we can easily integrate the mappings of the previous versions with a new version. If a genealogy of a clone class propagates through *(p+1)* versions, we call it a *p*-length genealogy. Now if a *p*-length genealogy has any inconsistent change pattern(s) during the propagation, it will be classified as *Inconsistently Changed Genealogy (ICG)*. Similarly, if a genealogy has any consistent change pattern(s) but does not any inconsistent change pattern(s) during the propagation, it will be classified as *Consistently Changed Genealogy (CCG)*. If a genealogy has been never changed during the observation period, it will be classified as a *Static genealogy*. Similarly, if a genealogy has any *Add* or *Delete* change pattern(s), it will be classified accordingly. We should note that most of the genealogies have two types of change patterns. One pattern is based on how it has been changed, such as *Static, CCG, ICG*, and another is based on the addition or deletion of clone fragments in a clone class such as *Add, Delete, Same*.

F. Time Complexity

In this section we provide the time complexity of the proposed mapping algorithm for two versions v_1 and v_2 of a program. We represent a version v_i by $(l_i, nf_i, ncf_i, ncc_i)$ where l_i, nf_i, ncf_i, ncc_i denote the LOC, number of functions, number of clone fragments, and number of clone classes respectively in version v_i. For function mapping, at first the algorithm matches name signatures of the corresponding functions of the two versions, v_1 and v_2. This step takes only nf_1 units of time because these mappings are done using a hash map (Section III-B1). Let us assume that α number of mappings are found after the execution of the previous step. Certainly, $min(nf_1, nf_2) \geq \alpha$. Now the origin detection of

TABLE II
FEATURES SUPPORTED

Features	Currently Suupported
Types of Clones	Type-1, Type-2 and Type-3
Granularity of clones	Block (Arbitrary + Structural), Function,
Clone Relation	Clone Pairs, Clone Class, RCF [8]
Adaptability to Tools	NiCad, CCFinderX, iClones
Adapt. to Languages	C, C#, Java
Software Versions	Revision level, Release level
Types of Genealogies	Same, Add, Delete, Static, CCG, ICG
Scalability	Large systems (e.g., Linux releases)

the remaining functions (Section III-B2) takes approximately $t = (nf_1 - \alpha) \times (nf_2 - \alpha) \times (l_1/nf_1) \times (l_2/nf_2)$ units of time, where l_i/nf_i is the average length of a function in v_i in terms of lines of code. Therefore, the required time for function mapping is $nf_1 + t$ units of time. Once all the functions are mapped, the time complexity for mapping each clone fragment to its contained function is also linear, which could be achieved by constructing a multi-valued hash map where file name is the key and the functions of the file are values.

For mapping the clone classes (Section III-C), the required time is ncf_1 units because we already have the mappings for each of the clone fragments of the two versions. Therefore, the speed of the whole process is inversely proportional to the number of functions that do not change their names (here the value of α) in the next version, and the number of clone fragments that do not belong to any function. Therefore, for the best case where all of the function names remained the same in the next version ($\alpha = nf_1$) and each clone fragment belongs to a function, the total time complexity is linear. However, it is quadratic in the worst case where all of the function names are changed in the next version ($\alpha = 0$) and all the clone fragments are outside of functions boundaries. Fortunately, on average 98% of the functions do not change their names [11], and from our experiment we found that only a few number of clone fragments are on the outside of functions, which indicates that the time complexity is almost linear.

IV. EVALUATION OF THE FRAMEWORK

In order to validate the efficacy of the proposed framework, we have developed a prototype clone genealogy extractor, gCad, as discussed in Section III. Currently, gCad can automatically extract both exact and near-miss clone genealogies at function or block level across multiple versions of a system, and classifies them automatically into meaningful change patterns as discussed in Section II. Although it can be easily adaptable to other languages for which a TXL grammar is available, we tested it for three languages, Java, C, and C#. gCad can deal with large systems such as the Linux Kernel which ensures its scalability. Currently, gCad can work with three clone detection tools, NiCad [18], CCFinderX[1] and iClones [8]. However, since it relies on relatively little information (e.g. file names and begin-end line numbers of the clone fragments), it is adaptable to any clone detection tools that provides such information. Table II summarizes the features that are currently available in gCad.

[1]http://www.ccfinder.net/

TABLE III
SUBJECT SYSTEMS

Attributes	ArgoUML	Linux Kernel	iTextSharp
Prog. Lang.	Java	C	C#
Start Date	Oct 4, 2008	Dec 14, 2009	Dec 8, 2009
End Date	Jan 24, 2011	Jan 5, 2011	Feb 18, 2011
Start Release	0.27.1	2.6.32	5.0.0
End Release	0.32.beta2	2.6.37	5.0.6
# Releases	47	45	7
# Functions	14798	244311	8162
Cloned Functions	3238	39266	1226
Type-1 clone class	169	659	53
Type-2 clone class	226	4483	53
Type-3 clone class	422	9076	282

A. Experiment

In this section, we discuss the details of our experimentation that we performed to evaluate gCad. We designed our case study in such a way that validates each characteristic of gCad as discussed above. We chose three open source systems, ArgoUML[2], the Linux Kernel[3], and iTextSharp[4]. We have selected these systems because they are developed in three different popular languages (Java, C, and C#), their size varies from medium to very large-scale in terms of lines of code. Furthermore, the interval length between releases of each system varies from a few days to several months. Table III summarizes the key attributes of the last releases of these systems that we considered.

Since currently gCad supports three clone detection tools, we used all of them to extract genealogies and to validate the result provided by gCad from different perspectives. Among them NiCad provides us the facility to detect both exact and near-miss clones at the function or block level. Because one of our primary objectives is to construct and classify near-miss clone genealogies, we used NiCad to detect code clones in all of the subject systems with NiCad setting of minimum clone size 5 LOC, consistent renaming of identifiers and 30% dissimilarity threshold. With this setting we detect a fairly good amount of near-miss clones because we allow consistent renaming of identifiers as well as a dissimilarity threshold of 30% which allows 30% dissimilarity of the clone fragments in their pretty-printed normalized format. We also used CCFinderX to detect code clones in two systems, ArgoUML and iTextSharp, with its default settings to see whether gCad works well with other clone detection tools or not. Finally, we used iClones to compare the accuracy of our mappings as discussed in Section V.

1) Result: We ran gCad on a *Mac Pro* that has a 2.93 GHz Quad-core Intel Xeon processor and 8 GB of memory (though our program uses a single processing unit). Table IV shows the results of our study for the different types of genealogies including the execution time of gCad. Here the execution time is the average time that is taken by gCad for two consecutive versions of a given subject system. From the first row we see that gCad takes less than a second to a couple of minutes for mapping the clones across two versions and

[2]http://argouml-downloads.tigris.org/
[3]http://www.kernel.org/
[4]http://sourceforge.net/projects/itextsharp/

TABLE IV
TEST RESULTS

Change Patterns/Time	ArgoUML			Linux Kernel		iTextSharp		
	Function	Block		Function	Block	Function	Block	
	NiCad	NiCad	CCFinderX	NiCad	NiCad	NiCad	NiCad	CCFinderX
Execution time/version	0.77s	2.02s	25s	1m18s	2m35s	0.50s	1.33s	3m38s
Same	893	1376	2266	16836	61362	479	821	959
Add	267	435	309	2255	9373	50	100	20
Delete	40	72	95	334	1603	17	18	14
Static	1350	2451	2019	18642	83295	500	840	897
CCG	33	41	307	1277	1796	20	52	83
ICG	463	719	510	4509	13535	102	136	20
Total Genealogies	1846	3211	2838	24428	98626	622	1028	1000

for finding different change patterns, depending upon the size of the given system and the level of granularity. However, the number of functions that have not changed their names in the next version plays a key role in reducing execution time. As we assumed, the number of renamed functions is very few for all of the systems. Approximately 2% of total functions have been renamed on average for the Linux Kernel, whereas for ArgoUML and iTextSharp it is less than 0.5%. We also found that functions are not renamed that much between minor releases. Most of the related studies did not report the execution time of their clone genealogy extractor except Bakota et al. [2]. As they reported, their preparation step took approximately 2 hours on an IBM BladeCenter LS20 machine equipped with 10 modules, each of them operating with two AMD Dual Core Opteron 275 processors, running on 2.2 GHz containing 4 GB of memory each, and the other step, construction of the evolution mapping, took approximately three hours to complete on one processor core for Mozilla Firefox having 12 versions. Since gCad is not exactly the same as of theirs, we could not directly compare our execution time with theirs. However, the execution time taken by gCad is far less than theirs and looks reasonable enough to be used for software maintenance purposes.

Identifying the evolving change patterns for Type-3 clones is always challenging and time consuming. From the results we see that most of the genealogies of all the systems at any granularity level (approximately 76%-80% for function and 76%-81% for block) did not change at all during the observation period, which is consistent with our previous results [20] where only Type-1 and Type-2 clones were considered. However, there is a large amount of *ICG* compared to *CCG* which is surprising. Therefore, we evaluated gCad in terms of mapping using different strategies. We will discuss the evaluation procedure in detail in the next subsections.

B. Correctness of Mapping

Evaluating the correctness of clone mapping is important for any clone genealogy extractor because all other calculations (e.g., identifying evolution patterns) depends on it. To evaluate the mapping (M) of clones between two versions of a program reported by a tool, the correct set of mappings (E) for the given versions is needed. The requirement of E makes the whole evaluation process challenging because it is very difficult to determine. Our quantitative evaluation is based on two criteria: precision and recall. While precision expresses the percentage of mapping that are correct, recall gives the percentage of correct mappings that the tool finds. We can compute precision and recall with the following two equations:

$$Precision = |M \cap E|/|M| \tag{2}$$

$$Recall = |M \cap E|/|E| \tag{3}$$

While manual verification of each mapping gives us an idea about the precision, it does not give any hints about recall. Therefore, besides manually validating the genealogies, we also performed a couple of automatic tests to measure both the precision and recall artificially.

1) Identity test: In this step we have assumed that there are two versions of a program that are identical, in other words, there has been no change between these two versions. Practically, the same program has been used as the inputs of gCad. The major advantage of this test is that the oracle mapping set (E) between these two versions of program is known and it is one to one self-mapping of each clone fragment. Therefore, the precision and recall for this step could easily be computed. We conducted the test for each of the versions we analyzed. For each of the tests, the value of both precision and recall were 1.0. Thus this test ensures that the gCad works well for scenario 1 in Table I.

2) Mutation/Injection Framework: We have adapted an existing mutation/injection based framework for evaluating clone detection tools [17] to evaluate the correctness of the mapping detected by gCad. However, instead of completely automating the framework we followed a semi-automated approach and only for function clones to avoid errors. As the original framework we also have two phases as follows:

a) Generation Phase: In the generation phase, we create a number of mutated versions of the original code base. To get a mutated code base, first we keep a copy of the original code base as the starting version of the subject system. We then make a second copy of the original code base, pick a clone class randomly from it, and remove the selected clone class from it. Once a clone class is selected, we change the fragments of that class in such a way that mimics a developers possible change scenarios following Roy and Cordy's editing taxonomy [17]. A TXL-based mutation process was used to apply the changes (single or multiple) to mutate each of the fragments to create a mutated clone class. Then we inject the mutated clone class into copied code base randomly but manually in such a way that the resulting code base is still syntactically correct. Now we can consider this mutated code base as the next evolving version of the original code base where a clone class has been changed. In this way, we created

TABLE V
RECALL AND PRECISION OF THE PROTOTYPE (%)

Subject Systems	Block Clones	Function Clones		
	Manual	Automated		Manual
	Precision	Precision	Recall	Precision
ArgoUML	0.98	1.0	0.97	0.99
Linux Kernel	0.99	0.96	0.94	0.99
iTextSharp	0.99	1.0	0.98	0.99

70 mutated code bases for each of the systems (last version) for scenarios 2-8 of Table I where each of the scenarios applied to 10 mutated code bases. For each of the scenarios where the function body was changed (scenarios 3,4,7, and 8), we applied consistent changes on half of the clone classes and applied inconsistent changes to the rest. All of these changes are stored in a database, which works as the oracle (E) for evaluating the correctness of the mapping. We then detected the mappings (M) and identified the change patterns between the original code base and the mutated code bases for each of the systems. Now that we have the oracle (E) and the detected genealogies (M), we can compute the precision and recall using equations 2 and 3 above.

b) Tool Evaluation Phase: Table V summarizes the results of our test cases. Among the 70 clone classes in the 70 mutants of ArgoUML, 69 of them were detected by NiCad because one of the clone classes was mutated beyond the similarity threshold of NiCad. On the other hand, gCad reported 67 genealogies. When we manually investigated why the two genealogies were missing, we found that the name and body of two clone classes were changed significantly. Therefore gCad could not find the origins of their fragments. However, all the detected genealogies were correct. In the Linux Kernel, among the 70 test cases, gCad reported 69, among which 66 were the correct mappings. We experienced incorrect mappings for three of the test cases because their corresponding clone classes were mapped incorrectly due to extensive change in the function bodies and function names. Similarly we found 69 correct mappings for iTextSharp. Although this approach computes the precision and recall artificially, it provides sufficient hints whether gCad works well for those editing scenarios during clone evolution.

3) Manual verification: Besides the automatic testing discussed above, we also manually validated our result. Since it is almost impossible to check manually all the mappings for all the systems, we systematically analyzed all the mappings across the last five versions for each system. At first, we analyzed the *Static* genealogies. If the clone fragments of such genealogies have not been moved, we did not need to investigate their source code because they have not changed at all. We manually checked all the *Static* genealogies for which some of the clone fragments have been moved. However, we did not find any false mapping for such genealogies. For all the changed genealogies, at first we investigated the correctness of the mapping by investigating their source code, function names, and file paths. After confirming the correctness of the mapping, we investigated the change patterns using a visual *diff*. We were not quite sure about some of the mappings, which were excluded from the calculations. During the manual verification we noticed that gCad detected many genealogies that where clone fragments changed in such a way that it would have been difficult to detect them using any heuristic or text-based similarity approach. For all of the systems we found that at least 98% of both the reported mappings and change patterns were correct. However, we found a few incorrect mappings when a function was renamed or deleted but a similar type of function was in the system. Table V summarizes the result of our manual verification. However, measuring the recall manually was out of scope.

V. COMPARING GCAD WITH OTHER METHODS

Besides the extensive manual verification and automatic testing, we also compared the quality of the mappings and change patterns from gCad with the mappings provided by iClones, and our previous approach [20], which was actually adapted from Kim et al. [10]. We have chosen the GNU Wget[5] as the subject system for this purpose because both the clones and their mappings across seven versions of this system detected by iClones are available online [21]. Furthermore, the size of this system is reasonable enough to manually investigate and compare each of the genealogies reported by the two tools under consideration. Therefore, to compare the mappings and the change patterns with iClones, at first we downloaded the *wget.rcf* file from the Software Clones website [21] that contains both the clones and their mapping information for seven versions of the subject system. CYCLONE [8] is another tool that can interpret the mapping information from iClones and construct genealogies from it. We ran CYCLONE on the *wget.rcf* to extract the clone genealogies. On the other hand, we extracted only the clone classes from *wget.rcf* for all the seven versions of the subject system, and applied gCad on them to extract the clone genealogies. It means that CYCLONE and gCad used the same clone classes to construct genealogies but CYCLONE used the mappings from iClones, whereas, gCad mapped the clone classes using our proposed method. We found that CYCLONE reported 370 genealogies, whereas, gCad reported 374 genealogies along with their change patterns. We then manually investigated each of the genealogies from these tools. We found 364 genealogies which were common between the two results. We investigated the 6 genealogies that gCad missed but was detected by iClones. We noticed that iClones returns some overlapping clone classes (which could be considered as false positives and should not have been detected), making two clone classes using the same clone fragments with only minor differences in the token sequence numbers of the fragments. While iClones maps such clone classes gCad does not in the cases when minor variations in token sequence numbers are within the same line because of its line-based comparison. This is in fact expected as such mappings can be considered false positives.

gCad also detected 4 more valid mappings than iClones. Since iClones exploits the difference information of the modified files in mapping clones, it cannot map those clones accurately in the cases where the functions/methods of the associated clone fragments are re-ordered w.r.t. their source

[5]http://www.gnu.org/software/wget/

TABLE VI
COMPARISON RESULTS

Clone Detector	Clone Mapping and Construction of Genealogies	Number of Genealogies	Common
iClones	(iClones + CYCLONE)/gCad	370/374	364
CCFinderX	Saha et al.[20]/gCad	75/81	75

coordinates within the same files in the next version. Our manual investigation confirms that this is exactly the reason why iClones missed these four mappings. Because gCad first maps the functions/methods and then locates clone fragments within the corresponding functions/methods, we have successfully detected these mappings

Our comparison results with iClones (considering its output as benchmark) confirm that gCad is no less than iClones/CYCLONE, rather is better both in terms of not reporting the mappings of some false positive clones and accurately mapping Type-3 clones. An independent comparison of gCad with iClones by using a third party clone detection tool for the detection of clones might have given a better picture in comparing the accuracy of iClones and gCad. However, such an experiment was not possible since iClones maps the clones during the detection and thus adapting a third party clone detection tool was not applicable.

Since iClones maps the clone classes incrementally during the detection of clones, it is obviously faster than gCad. In order to compare the scalability, we have run iClones with the Linux Kernel versions, and experienced “Out of Memory” error messages. We then attempted to run iClones only with two versions of Linux Kernel and experienced the same error. For this experiment we used a Mac Pro machine that has a 2.93 GHz Quad-core Intel Xeon processor with 8 GB of memory. However, when we talked to the author of iClones, he suggested that he successfully worked with Linux Kernel in a server machine of 64 GB memory. Unfortunately, we do not have such a high configured server within our reach.

In order to compare gCad with our previous method [20] we used CCFinderX for clone detection since it was based on CCFinderX. We ran both gCad and our previous method separately on the detected clone classes to construct genealogies, and found 81 and 75 genealogies respectively. All of the 75 clone genealogies detected by our previous method were also detected by gCad. When we investigated the six additional genealogies reported by gCad, we noticed that they are in fact valid genealogies. Our previous method missed them because the clone fragments were significantly changed to the extent that their text similarities were less than the given threshold. However, gCad was successful in accurately identifying those fragments by first mapping the functions and then locating the clone fragments within the functions. These comparison results confirm that gCad not only can detect all the genealogies reported by our previous approach but also can accurately handle the mapping of Type-3 clones. The comparison results are summarized in the Table VI.

CloneTracker is a great tool for tracking clones in the IDE [5]. However, our work is fundamentally different from that of CloneTracker both in terms of objectives and the representation of clone fragments. We attempted to identify clone genealogies and classify their change patterns. On the other hand they attempted to assist developers in managing clones in evolving software in the IDEs. CloneTracker can neither construct clone genealogies nor can classify them. However, we were still interested to see how well the CRDs of CloneTracker can map clone classes of different versions of a software systems. We experienced that it is not straightforward at all. The authors also noted similar concerns and used several heuristics for checking the robustness of CRDs for tracking clones over versions. Using a dummy project we experienced similar results as of them [5] that it can track about 95% of Type-1 and Type-2 clones. However, in case of Type-3 clones we experienced difficulty with CloneTracker. They also noted that CloneTracker cannot handle Type-3 clones because of the possible changes in the anchors of CRDs [5]. In contrast, our approach is specifically designed for tracking Type-3 clones.

VI. Threats to the Validity

There are two potential threats to the validity of our result. First, in our function mapping algorithm, we did not consider the merging and splitting of functions. Therefore, if a function splits in the next version, our algorithm first attempts to map it to the part in the next version for which the function name remains the same as of previous version. If the name of any of the parts does not match, our algorithm maps it to the part in the next version for which source code was more similar to its origin. However, the effect of this situation should be minimal on the results for clone genealogies and change patterns because if any clone fragments of a clone class splits in the next version then all fragments of that clone class should be split accordingly for maintaining consistent change, and gCad should map similar fragments in the next version and thus will detect the change as consistent change, otherwise will detect as inconsistent change. Second, there might have been some unintentional errors during the manual verification due to the lack of the domain knowledge or human errors.

VII. Related Work

Extracting clone genealogies across multiple versions of a software system is not a new topic. There have been several such studies. While they differ significantly in many aspects, they are also related to this study. Kim et al. [10] were the first who mapped code clones across multiple revisions. They used two metrics, *text similarity* and *location overlapping* to map clones between two revisions. They implemented a *CGE* that is able to extract and classify genealogies automatically. However, they only considered Type-1 and Type-2 clones, detected by CCFinder. Aversano et al. [1] extended Kim et al.’s study, and investigated how clones are maintained when an evolution activity takes place. However, they manually investigated all the genealogies for finding change patterns and mostly focused on Type-1 clones. We overcome the limitations of these two studies by proposing a fast and scalable approach that can extract not only Type-1 and Type-2 but also Type-3 clone genealogies, works with different clone detection tools and can automatically identify the change patterns at both function and block levels. To understand the stability of cloned code Krinke conducted two separate but related studies [12], [13] for Type-1 clones. Like Aversano et al. he also considered the clones of the first revision in the observation period and examined the changes of clones by extracting the changes from the source code repository. Thummalapenta et al. [22] conducted a similar study to understand the maintenance implications of clones.

Bakota et al. [2] proposed an AST based machine learning approach for mapping clones across consecutive versions. They used a number of similarity metrics such as file name, position and lexical structure of the clone instances, and so on, to find the appropriate mapping between two clone instances. However, using a large number of similarity features makes the mapping process computationally expensive. In contrast, we used simple similarity metrics which ensured fast computation while maintaining high accuracy.

As of Duala-Ekoko and Robillard [5] (Section V), Bettenburg et al. [3] also used a *CRD* to map clones between two versions. However, they did not identify the change patterns of the genealogies automatically.

As of Göde and Koschke [6] (Section V), Nguyen et al. [15] also introduced an incremental clone detection tool, ClemanX. The advantages of both of the approaches mostly relies on the tiny changes between revisions. Furthermore, though they mapped clones across multiple versions, they did not classify genealogies based on the change patterns as we did.

Lozano and Wermelinger [14] mapped clones' imprint across multiple versions. Like us they also mapped all the functions/methods before mapping clone classes. However, they did not consider Type-3 clones, whereas we applied a sophisticated technique both for fast and efficient mapping of Type-3 clones and identify the change patterns of such clones with high precision and recall.

VIII. Conclusion

As for Type-1 and Type-2 clones, extracting the genealogies of Type-3 clones and identifying the change patterns are equally important, especially because there are a significant number of such clones in software systems. However, mapping Type-3 clones across multiple versions of a program and automatically identifying their change patterns is challenging. In this paper, we proposed a scalable and adaptable framework that can extract both exact and near-miss (Type-2 and Type-3) clone genealogies and can identify their change patterns automatically. To validate the efficacy of the proposed framework, we developed a prototype and experimented both with multiple versions of three open source systems including the Linux Kernel and a mutation/injection-based framework. We also manually analyzed many detected genealogies including their change patterns, and compared the mapping reported by our prototype with that of an incremental clone detection tool. Our experience suggests that the proposed method is adaptable to other clone detection tools, and reasonably fast while maintaining high precision and recall. Furthermore, it can even tolerate significant changes between two versions and thus could be effectively used to map clones at the release level as well as at the revision level. We believe that this approach would be useful for researchers and developers when studying the evolution of both exact and near-miss clones. In future we would like to conduct large scale empirical studies in the evolution of near-miss clones both at the release and revision levels and investigate the stability of inconsistent changes in Type-3 clones compared to that of Type-1 and Type-2 clones along with the intentionality of such inconsistent changes.

Acknowledgements: We first thank Nils Göde for helping us with iClones and for the fruitful discussion during his recent visit in our lab. We also would like to thank Ekwa Duala-Ekoko for helping us with CloneTracker, and Rainer Koschke for providing us an academic licence of Bauhaus toolset/iClones.

References

[1] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study," Proc. *CSMR*, 2007, pp. 81–90.
[2] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," Proc. *ICSM*, 2007, pp. 24–33.
[3] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan, "An empirical study on inconsistent changes to code clones at release level," Proc. *WCRE*, 2009, pp. 85–94.
[4] J. Cordy, "The TXL source transformation language," *Sci. of Com. Prog.*, 61(3):190–210, 2006.
[5] E. Duala-Ekoko and M. P. Robillard, "Clone Region Descriptors: Representing and Tracking Duplication in Source Code," *TOSEM*, 20(1)3:1–31, 2010.
[6] N. Göde and R. Koschke, "Studying clone evolution using incremental clone detection," *JSME*, 2010.
[7] M. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code," entities. *IEEE TSE*, 31(2):166–181, 2005.
[8] J. Harder and N. Göde, "Efficiently Handling Clone Data: RCF and cyclone," Proc. *IWSC*, 2011, pp. 81–82.
[9] J. Harder and N. Göde, "Modeling clone evolution," Proc. *IWSC*, 2009, pp. 17–21.
[10] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," Proc. *ESEC-FSE*, 2005, pp. 187–196.
[11] S. Kim, K. Pan, and J. E. Whitehead, "When functions change their names: Automatic detection of origin relationships," Proc. *WCRE*, 2005, pp. 143–152.
[12] J. Krinke, "A study of consistent and inconsistent changes to code clones," Proc. *WCRE*, 2007, pp. 170–178.
[13] J. Krinke, "Is cloned code more stable than non-cloned code?," Proc. *SCAM*, 2008, pp. 57–66.
[14] A. Lozano and M. Wermelinger, "Tracking clones imprint," Proc. *IWSC*, 2010, pp. 65–72.
[15] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen, "Scalable and Incremental Clone Detection for Evolving Software," Proc. *ICSM*, 2009, pp. 491–494.
[16] F. Rahman, C. Bird, P. Devanbu, "Clones: What is that smell?," Proc. *MSR*, 2010, pp. 72–81.
[17] C. K. Roy and J. R. Cordy, "A mutation / injection-based automatic framework for evaluating code clone detection tools," Proc. *Mutation*, 2009, pp. 157–166.
[18] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," In *ICPC*, 2008, pp. 172–181.
[19] C. K. Roy and J. R. Cordy, "Near-miss Function Clones in Open Source Software: An Empirical Study," *JSME*, 22(3):165–189, 2010.
[20] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study," Proc. *SCAM*, 2010, pp. 87–96.
[21] Software Clones. http://www.softwareclones.org/ (Feb 2011)
[22] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, "An empirical study on the maintenance of source code clones," *ESE*, 15(1):1–34, 2009.
[23] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE TSE*, 30(9):574–586, 2004.
[24] T. Zimmermann, P. Weigerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE TSE*, 31(6):429–445, 2005.