gCad: A Near-Miss Clone Genealogy Extractor to Support Clone Evolution Analysis

Ripon K. Saha* Chanchal K. Roy† Kevin A. Schneider†
*The University of Texas at Austin, USA
†University of Saskatchewan, Canada
ripon@utexas.edu, chanchal.roy@usask.ca, kevin.schneider@usask.ca

***Abstract*—Understanding the evolution of code clones is important for both developers and researchers to understand the maintenance implications of clones and to design robust clone management systems. Generally, a study of clone evolution starts with extracting clone genealogies across multiple versions of a program and classifying them according to their change patterns. Although these tasks are straightforward for exact clones, extracting the history of near-miss clones and classifying their change patterns automatically is challenging due to the potential diverse variety of clone fragments even in the same clone class. In this tool demonstration paper we describe the design and implementation of a near-miss clone genealogy extractor, gCad, that can extract and classify both exact and near-miss clone genealogies. Developers and researchers can compute a wide range of popular metrics regarding clone evolution by simply post processing the gCad results. gCad scales well to large subject systems, works for different granularities of clones, and adapts easily to popular clone detection tools.**

***Index Terms*—Type-3 clones; clone genealogy; clone evolution**

I. Introduction

After a decade of active research, it is evident that code clones have both a positive [3] and a negative [5] impact in the maintenance and evolution of software systems. Code cloning is inevitable in software development, and in order to exploit the advantages of clones while lowering their negative impact, it is important to understand the evolution of clones and manage them accordingly.

Generally, a clone evolution study starts with detecting clones in multiple versions of a program, and constructing genealogies by mapping clones across the different versions. A clone genealogy tells us how the code fragments of a clone class change through versions during the evolution of a subject system. Researchers have proposed and implemented a number of clone genealogy extractors (CGEs) to study the evolution of clones. However, most of the tools were designed focusing on some particular tasks of interests and for only Type-1 and Type-2 clones. Thus CGEs rarely meet the current diverse requirements such as fast construction and classification of near-miss clone genealogies and adaptation/integration of a third party clone detection tool. Furthermore, most of the reported tools are not publicly available.

The recently developed incremental clone detection methods [2] improve the genealogy construction time considerably by integrating clone mapping with clone detection. However, these methods have their own set of limitations. First, they are unable to utilize the results obtained from a classic non-incremental clone detection tool as the detection of clones and their mapping is tightly integrated. Since most existing clone detection tools are non-incremental, they restrict developers and researchers to using a limited number of clone detection tools. For flexibility it is important to have a clone evolution analysis tool that is independent of the clone detection tools. Second, with each new revision or release of the subject system, the entire detection and mapping process needs to be repeated, because clones are detected and mapped concurrently. Since clone management is likely being conducted on a changing system, it is a disadvantage for an approach to require detecting clones for all versions each time a new revision/version is produced. Third, the incremental approach is fast enough for both detecting and mapping for a given set of revisions. However, it might not be as beneficial for the release level because there might be significant differences between releases.

In this tool demonstration paper, we describe the design and implementation of a near-miss CGE, gCad (evolved from our earlier research [6]) that can extract both exact (Type-1) and near-miss (Type-2 and Type-3) clone genealogies across multiple versions of a program, and identify their change patterns automatically. Genealogies are constructed incrementally by merging current mapping results with previously stored genealogies to give a complete result. gCad scales well to large subject systems, works for different granularities of clones, and adapts easily to popular clone detection tools. Developers and researchers also can compute many popular metrics of clone evolution by simply post processing the gCad results.

II. Approach and Implementation

This section describes the overall design and implementation of gCad. Usually gCad accepts n versions of a program and their clones, maps clone classes between the consecutive versions, and extracts how each clone class changes throughout an observation period. Therefore, it is expected that users will have detected clones in all n versions of the program before running gCad. A version may be a release or a revision. gCad mainly works in the following four steps to construct and classify genealogies: (1) Function Mapping, (2) Clone Mapping, (3) Automatic Identification of Change Patterns, and (4) Constructing Genealogies. The first three steps are

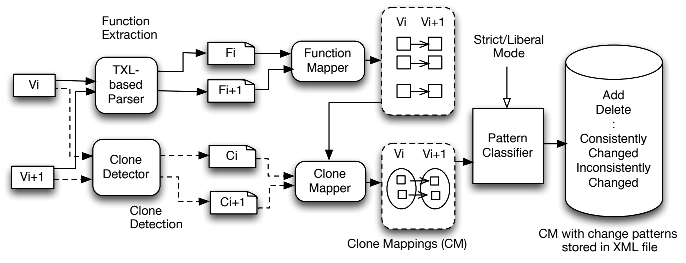


Fig. 1. The First Three Steps of the gCad Framework

performed for each consecutive version pairs. Finally, gCad merges all the results to construct genealogies for the n versions. Figure 1 shows the overall structure of gCad for two versions of a program.

A. Function Mapping

For two given consecutive versions, v_i and v_{i+1} of a software system, first gCad extracts all the function signatures from both the versions. For extracting functions gCad uses TXL [1], a special-purpose programming language that supports lightweight agile parsing techniques. We exploit TXL's extract function, denoted by [ˆ], to enumerate all the functions. For each function gCad stores 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. 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, gCad uses these attributes as a composite key to map functions between two versions, which is computationally very fast. However, in practice some functions are renamed, or could move to different files or directories during the evolution of the system. In those cases, gCad uses the longest common subsequence count (*LCSC*) similarity metrics of function name and body to find the origin of a function.

B. Clone Mapping

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 $CC^i = \{cc^i_1, cc^i_2, ..., cc^i_n\}$ be the reported clone classes in v_i where $cc^i_j = \{CF^i_{j1}, CF^i_{j2}, ..., CF^i_{jm}\}$. Here CF^i_{jk} refers to the clone fragments of the clone class cc^i_j where $1 \leq k \leq m$. In order to map clones between two versions, gCad first maps each clone fragment CF^i to its contained (parent) function, F^i in v_i using the following algorithm.

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

At this point, since all the functions are already mapped, the problem of mapping clones between two versions of a program reduces to the mapping of clones between two versions of a function. Therefore, mapping clones in gCad is computationally very `fast`. If F^i contains only one clone fragment, gCad can easily map that clone fragment in v_{i+1} since it already knows the corresponding mapped function F^{i+1} in v_{i+1}. The mapped clone fragment CF^{i+i} will be found in F_{i+1} if it has not been removed. If F^i has more than one clone fragment, gCad uses the LCSC Similarity score to map corresponding clone fragments in F_{i+1}. Once the clone mapping is completed at the fragment level, gCad maps clones at the class level. A clone class in v_i can be split into two or more clone classes in v_{i+1} due to inconsistent changes. For each clone fragment of a given clone class cc^i_j, gCad checks if they map into single or different clone classes in v_{i+1}. If all the clone fragments are mapped to the same class, cc^{i+1}_x, gCad maps $cc^i_j \rightarrow cc^{i+1}_x$. On the other hand, if they are mapped to multiple classes, $\{cc^{i+1}_x, cc^{i+1}_y, ...\}$, which usually indicates a split, gCad keeps track of them as $cc^i_j \rightarrow \{cc^{i+1}_x, cc^{i+1}_y, ...\}$.

C. Automatic Identification of Change Patterns

Automatic and accurate identification of change patterns is one of the important features of a CGE. Although identifying whether a Type-1 clone class changed consistently or inconsistently is straightforward, it is challenging for near-miss (Type-2 and Type-3) clones due to the diverse variety of clone fragments in the same clone class. gCad applies a multi-pass computationally efficient method to identify the change patterns of both exact and near-miss clones.

In the first pass, gCad identifies the clone classes that did not change in the next version (*Static*), and those clone classes that have split. The program identifies the split clone classes as an inconsistent change because it is evident that their fragments changed inconsistently, and thus they are part of two or more clone classes in the next version.

In the second pass, gCad makes 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, gCad 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 a consistent change, otherwise as an inconsistent change.

In the third pass gCad considers the rest of the clone classes (Type-2 clones and Type-3 clones with identifiers renaming). Since the clone fragments of these clone classes have variations in their identifiers, gCad cannot exploit *diff* directly because the differences will not be the same even if the fragments changed consistently. In order to deal with this issue, gCad consistently renames the identifiers of the clone fragments using TXL. For example, 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. gCad then computes the differences. As before, if the differences are the same, gCad classifies the change pattern as a consistent change, otherwise as an inconsistent change. gCad

```
<mappings>
    <version1>
    <version2>
    <mapping>
         <id>
         <clone type>
         <change pattern>
         <change of fragment>
         <clone class of v1>
             <id>
             <number of nlines>
             <number of fragments>
             <source>
                  <file name>
                  <start line>
                  <end line>
                  <source fragment id>
             </source>
        <clone class of v1>
        <clone class of v2>
            ....
        </clone class of v2>
    </mapping>
    <mapping>
        ........
    </mapping>
    ........
</mappings>
```

Fig. 2. XML Structure for Storing Clone Mappings between Two Versions

also reports add and delete patterns based on the number of fragments in a clone class in the previous and next versions. All the identified mappings and their change patterns are stored in an XML file (cf. Figure 2) for future use.

D. Genealogy Construction

At this point, gCad has all the detailed mapping information for each consecutive version $\{(v_1,v_2), (v_2,v_3), ..., (v_{n-1},v_n)\}$ stored in an XML file for each version-pair. Then gCad combines all of the results of each version-pair by matching clone class ids. For example, if a clone class cc_1^1 in v_1 maps to a clone class cc_2^5 in v_2, which again maps to cc_3^4 in v_3, $\{cc_1^1 \rightarrow cc_2^5 \rightarrow cc_3^4\}$ forms a clone genealogy. Now if a clone genealogy has any inconsistent change patterns during the propagation, it will be classified as an *Inconsistently Changed Genealogy (ICG)*. If a genealogy has any consistent change patterns but does not have any inconsistent change patterns, it will be classified as a *Consistently Changed Genealogy (CCG)*. If a genealogy has any *Add* or *Delete* change patterns, it will be classified accordingly.

III. Tool Features

1) Clone Coverage: gCad can construct genealogies for both exact and near-miss clones and classifies their change patterns automatically. gCad also works for any types of clone granularities (e.g., function clones and block clones) and clone relationships (clone pairs, clone classes, and RCF [4]).

2) Adaptability: From the clone mapping phase, it should be noted that gCad uses only file name and line numbers to find the parent function of a clone, and then uses only clone code fragments for mapping purposes. Therefore, gCad is easily adaptable to any clone detectors that report their results with this information. We have successfully tested gCad for NiCad, CCFinderX and iClones with high accuracy [6].

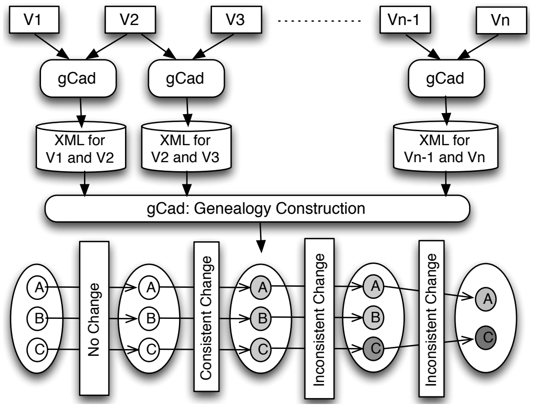


Fig. 3. Genealogy Construction

3) Supporting Languages: gCad uses TXL for function extraction and normalizing source code to classify change patterns of near-miss clones. Therefore, gCad can support the wide range of programming languages that TXL supports. We have tested gCad for C, C#, and Java programming languages.

4) Operating Modes: An issue with classifying change patterns of Type-3 clones is that whether the changes in the gap should be considered in determining the change pattern. 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, sometimes although gapped lines are textually different, they are semantically similar. Figure 4 shows such a real world example in dnsjava. Therefore, developers may miss some unintentional inconsistent changes if the CGE ignores gaps. gCad supports both ways in the form of two operational modes.

Liberal Mode: In this mode, gCad ignores changes that occur in the gaps of each clone pair in the same clone class. Therefore, a change will be identified as inconsistent change only when similar lines of any clone pair in the same clone class change differently with respect to one another.

Strict Mode: In this mode, gCad does not consider the gaps as a special case. If the changes to the clone fragments are not the same, it will be considered as an inconsistent change.

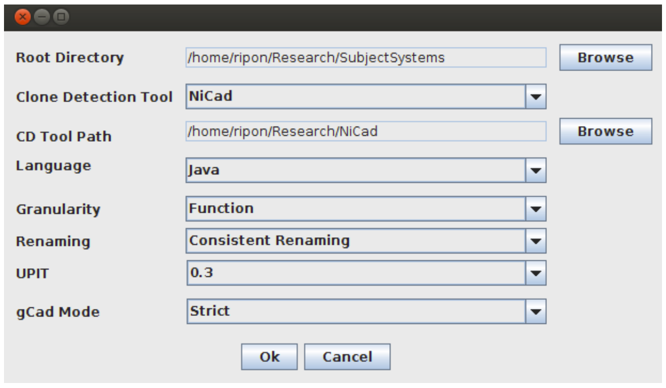
5) Flexibility of changing the scope of study: In the genealogy construction phase, we described how gCad constructs genealogy incrementally. Therefore, users can add a new version (e.g., v_{n+1}) anytime easily. She just needs to run gCad for mapping clones between v_n and v_{n+1} and gCad can add the new results to form the genealogies from version v_1 to v_{n+1}. Users can also insert a version in the middle (e.g., v_i) by running gCad for mapping clones of (v_{i-1}, v_i) and (v_i, v_{i+1}).

6) Scalablity: gCad has almost a linear time complexity with respect to the number of functions in the subject system. Therefore, gCad scales well to very large systems. A detailed analysis of the time complexity is discussed elsewhere [6]. We successfully ran gCad for 45 releases of the Linux kernel.

7) Extensibility: gCad is extensible. We have already developed an extension of gCad that populates gCad results into a MySQL database to support SQL query on clone evolution data. Development of a comprehensive clone evolution visualization tool is in progress on the top of gCad.

```
public Object sendAsync(final Message query, final ResolverListener listener) {
        final Object id;
        :
        String name = this.getClass() + ": " + query.getQuestion().getName();
        WorkerThread.assignThread(new ResolveThread(this, query, id, listener), name);
        return id;
}
public Object sendAsync(final Message query, final ResolverListener listener) {
        final Object id;
        :
        String name = getClass() + ": " + query.getQuestion().getName();
        WorkerThread.assignThread(new ResolveThread(this, query, id, listener), name);
        return id;
}
```

Fig. 4. A Semantically similar but textually different change in a Type-3 clone class in dnsjava



Root Directory	/home/ripon/Research/SubjectSystems
Clone Detection Tool	NiCad
CD Tool Path	/home/ripon/Research/NiCad
Language	Java
Granularity	Function
Renaming	Consistent Renaming
UPIT	0.3
gCad Mode	Strict

Fig. 5. GUI for gCad Settings

8) Working with gCad: gCad is very easy to configure and use. Currently gCad provides a set of GUIs for initiating various operations. Users just need to set some parameter values through a simple GUI (shown in Figure 5) as necessary to run gCad for a given system. gCad reports various results through a set of XML, HTML, and text files. The gCad tool, a sample result set for ArgoUML, and all other supporting documents (a live demonstration and user manual) are available at https://webspace.utexas.edu/rks848/www/miscellaneous.html.

IV. Application

Understanding the evolution of code clones is important to manage clones efficiently and gCad can be a useful tool for developers to make timely decisions regarding clones by collecting relevant information regarding clone evolution without additional efforts. For example, developers can understand the changing nature or maintenance effort of clones by observing the proportion of consistently and inconsistently changed genealogies. They can also choose a set of clone classes for refactoring that change consistently and frequently, or can manually check to see if an inconsistent change was intentional.

Researchers can use gCad to analyze the evolutionary data of clones from different perspectives to design new techniques to manage clones. We already performed two empirical studies using gCad to understand the evolution of Type-3 clones and to evaluate the conventional wisdom in clone removal. In the first study [7], we used gCad to compute the proportions of consistently and inconsistently changed genealogies, change frequencies, and ages of Type-1, Type-2, and Type-3 clones separately to understand if the evolution of Type-3 clones is different from that of Type-1 and Type-2. We also used gCad's strict and liberal modes to understand the extent of changes in gaps for Type-3 clones. We found that the proportion of consistently changed Type-3 clone genealogies increases considerably if we ignore gap changes. In the second study [8], we used gCad to find the clone classes that were removed from systems. Then we investigated different attributes of the code clones, such as number of clone fragments, their distributions in different files, change patterns, change frequencies, and so on, in relationship to clone removal.

V. Summary

This tool demonstration paper describes the design and implementation of a near-miss clone genealogy extractor, gCad to support the analysis of clone evolution. gCad can extract both exact and near-miss clone genealogies, classify their change patterns automatically, and provide various important information regarding clone evolution through a number of widely used metrics. Our evaluation results from a previous study [6] show that gCad is accurate, scales well to very large systems (e.g., Linux Kernel releases) and is adaptable to different clone detectors, which makes it useful for understanding the various evolutionary phenomena of code clones. We believe gCad would be helpful not only for clone researchers but also for developers or maintenance engineers in making decisions for reducing the negative impacts of code clones.

References

[1] J. R. Cordy, "The TXL Source Transformation Language," *Sci. of Com. Prog.*, 61(3):190–210, 2006.

[2] N. Göde and R. Koschke, "Studying Clone Evolution using Incremental Clone Detection," *JSME*, 25:165–192, 2010.

[3] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An Empirical Study of Code Clone Genealogies," Proc. *ESEC-FSE*, 2005, pp. 187–196.

[4] J. Harder and N. Göde, "Efficiently Handling Clone Data: RCF and cyclone," Proc. *IWSC*, 2011, pp. 81–82.

[5] F. Rahman, C. Bird, P. Devanbu, "Clones: What is that Smell?," Proc. *MSR*, 2010, pp. 72–81.

[6] R. K. Saha, C. K. Roy, and K. A. Schneider, "An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies," Proc. *ICSM*, 2011, 293–302.

[7] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the Evolution of Type-3 Clones: An Exploratory Study," Proc. *MSR*, 2013, 139–148.

[8] M. Z. Zibran, R. K. Saha, C. K. Roy and K. A. Schneider, "Evaluating the Conventional Wisdom in Clone Removal: A Genealogy-based Empirical Study," Proc. *SAC*, 2013, pp. 1223–1230.