A Case for Structural Clones

Hamid A. Basit
Department of Computer Science
School of Science and Engineering
Lahore University of Management Sciences
hamidb@lums.edu.pk

Stan Jarzabek
Department of Computer Science
School of Computing
National University of Singapore
stan@comp.nus.edu.sg

Abstract

In previous work, we introduced the concept of structural clone as recurring configurations of simple clones (i.e., similar code fragments). We also described a technique for detection of certain types of structural clones. The premise of our approach is that structural clones can provide a higher level and more useful view of the similarity situation in software than simple clones alone. In this paper, we set up a ground for systematic analysis of the structural clone phenomenon: We define structural clones more formally and more generally than we did before. We show how structural clone analysis extends the benefits of analysis based on simple clones only in areas of program understanding, maintenance, reuse, refactoring, and plagiarism detection.

1. Introduction

Similar program parts are termed *software clones*. *Simple clones* are formed by textually similar code fragments. Larger program structures, formed by configurations of simple clones, are called *structural clones* [1][2]. Examples of clones are similar class methods, classes, source files, directories, any similar software components or recurring patterns (configurations) of similar components.

```
int f1(byte) {
int c;
  While (isalpha(c)) {
   if (p == buf) p=grow  buf(p);
   c = getc(finput);}
}
```

a1

```
int f2() {
int d;
  While (isdigit(d)) {
   if (p == buf) p=grow  buf(p);
   skip();
   d = getc(finput); }
}
```

a2

```
char f3(char) {
char c;
  While (isdigit(c)) {
    if (p == buf) p=grow  buf(p);
    if (c == '-') return;
    c = getc(finput);}
}
```

a3

Figure 1. A simple clone class

Figure 1 and Figure 2 show intuitive examples of textual and structural similarities considered in this paper. Three code fragments (a1,a2,a3) in Figure 1 differ in code details highlighted in **bold**. We can consider them as simple clones of each other, provided they meet a certain user-defined similarity threshold.

Suppose code fragments (b1,b2,b3), (c1,c2,c3) .. (e1,e2,e3) also form simple clone classes, and they appear in files X1, X2 and X3, as shown in Figure 2 (a). Grey code is unique to X1 and X2, and black code is unique to X3. Three groups [a1,b1,c1,d1,e1], [a2,b2,c2,d2,e2] and [a3,b3,c3,d3,e3] form a structural clone class.

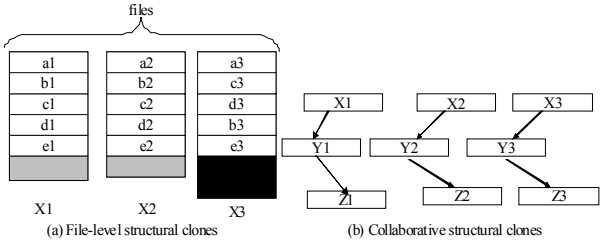

Figure 2. Structural clones

Suppose a substantial part, say at least 80%, of each of the three files is covered by structural clones (meaning that less than 20% of code is contained in grey or black areas). Then, we could consider files X1, X2 and X3 a file-level structural clone class. This abstraction step allows us to build structural clones in a hierarchical way, whereby higher-level, coarsely-grained structural clones are formed in terms of smaller-granularity ones. The "80% coverage by clones" is an example of a user-defined similarity threshold value.

Suppose further that (Y1,Y2,Y3) and (Z1,Z2,Z3) are also file-level structural clone classes and they form collaborative (via message passing) structures such as shown in Figure 2(b). Then, we consider a group of such patterns a higher-level collaborative structural clone class.

The above examples illustrate that structural clone class is formed by a group of program structures whose respective elements are similar and inter-related in similar ways.

2. An example of structural clones in a web application

Project Collaboration Environment (PCE) is a web application that supports project planning and execution.

PCE modules manage information about Staff, Project, Product, Task, Notes, and File. Each PCE module implements operations such as create, edit, delete, display or save to manage their respective records. For example, Staff module manages records of staff members, and Product module tracks the status of project deliverables. In [14], we studied trade-offs involved in refactoring clones in PCE using the server page technique of PHP.

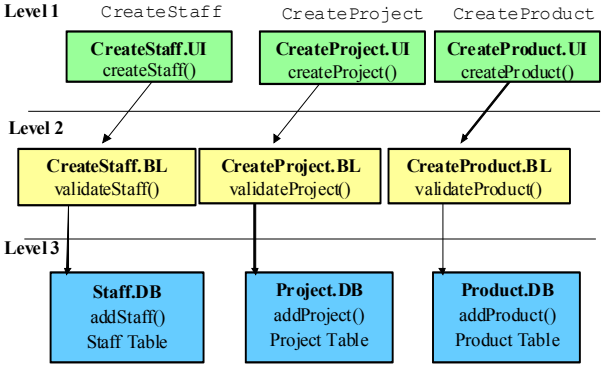

Figure 3. Structural clones in PCE

Figure 3 shows three structures implementing features `CreateStaff`,`CreateProject`,`CreateProduct`. Boxes are PHP files implementing user interface (at the top), business logic (in the middle), and database aspects of receptive features. Each of the files consists of PHP functions. Functions across files at each level (UI, BL and DB) form simple clone classes such as shown in Figure 1. As the same (or similar) configurations of simple clones recur across files at each level, we consider these configurations a structural clone class. As PHP files are densely covered by these structural clones, we consider files at each level as similar (abstraction step). Finally, we find calling relation among functions in respective PHP files, so we infer that the three structures form a collaborative structural clone class. Therefore, we consider features `CreateStaff`, `CreateProject`, and `CreateProduct` as similar (abstraction step).

It is interesting to observe that the above structural clones occur not only within a single PCE, but also across PCE product variants (e.g., PCEs used in different project teams).

3. Towards definition of structural clones

Intuitively, a *clone relation* exists between two *program entities* (*entities* for short) E1 and E2, if E1 and E2 satisfy certain thresholds of pre-defined similarity metrics. Human judgment is also an important factor in deciding whether two entities are clones of each other or not. We introduce some preliminary terms first.

An entity can be any program element that we can clearly identify such as program statement, code fragment, file, directory, class method, interface, package, component, group of collaborating components, sub-system or the whole system.

Following terms in [8], our *clone relation* is an equivalence relation (i.e., reflexive, transitive, and symmetric relation). For a given clone relation, a pair of entities is called a *clone pair* if a clone relation holds between the two entities. An equivalence class of a clone relation is called a *clone class*. A clone class is a maximal set of entities in which a clone relation holds between any pair of entities.

Code fragments (i.e., segments of contiguous code) are the simplest type of entities that can participate in a clone relation. In such a case, we call them *simple clones*.

A *program structure* (*structure* for short) is a connected mixed multigraph where nodes are entities, and (directed or undirected) edges are relationships between entities. A *relationship* is any meaningful physical or logical connection between two entities in a structure. In a mixed multigraph, the same pair of nodes can be connected by multiple edges which is useful in characterizing certain types of structures.

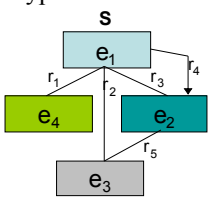

Figure 4. A structure

In Figure 4, we see structure S consisting of four entities (e_1, e_2, e_3, e_4), and five relationships (r_1, r_2, r_3, r_4, r_5) among them. Entities e_1 and e_2 are related by means of two relationships, namely r_3 and r_4, where r_4 is a directed relationship. One important relationship is the co-location of the interrelated entities in classes, files, or directories. For example, groups of code fragment in Figure 2 (a) are co-located in files. Other examples of relationships include "message passing", "inheritance", "association", or "hyperlink" among web pages. Any other physical or logical relationship can be considered.

To explain the structure hierarchies, we introduce the terms *atomic entities* and *abstract entities*. An *atomic entity* is one which has no relevant internal structure. For example, a code fragment is often considered an atomic entity in clone analysis. An *abstract entity* is one whose internal structure is abstracted away, to form a building block for higher level structures. Abstract entities allow us to define higher-level structures in terms of lower-level structures at as many levels as is useful.

We can now define more precisely a structural clone relation as a clone relation between two structures as follows:

Definition: A *structural clone relation* holds between two structures S1 and S2 if (and only if):

(a) S1 and S2 have the same graph structure

(b) a clone relation has already been established between corresponding entities in S1 and S2, and
(c) corresponding relationships in S1 and S2 are of the same type.

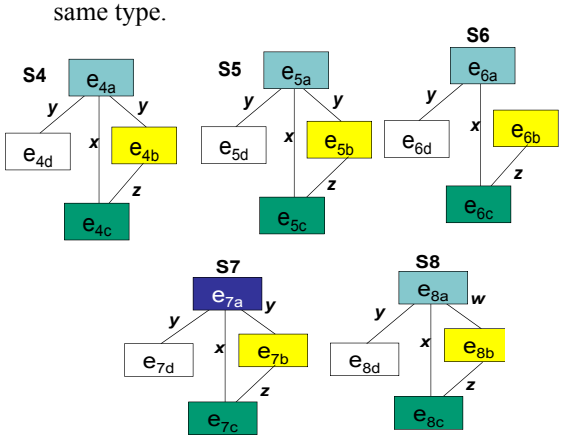

Figure 5. Five structures

In Figure 5, we see five structures S4-S8, each consisting of four entities, where *x*, *y*, *z*, and *w* are different relationship types. Any two entities identified by the same subscript letter (i.e., a, b, c or d) are corresponding entities. For example, e_{4a} and e_{5a} are corresponding entities in S4 and S5.

Suppose entities of same shade are in clone relation with each other (e.g., e_{4a}, e_{5a}, e_{6a}, and e_{8a} are clones of each other, but e_{7a} is not in clone relation with the rest). Then, only S4 and S5 are in structural clone relation. The above definition characterizes structural clones that our Clone Miner [1][2] can find..

We can relax this definition, so that it captures other types of structural clones. For example, relaxing condition (a) can allow us to include structures s6, s7 and s8 into a structural clone class together with s4 and s5. Our definition can benefit from more systematic treatment of clone similarity, and can include clone similarity criteria, metrics and their threshold values. Systematic similarity treatment is a difficult and important area of research [13].

4. Detection of structural clones

Clone Miner [1][2] can find structural clones such as shown Figure 1and Figure 2(a). Clone Miner detects simple clones first using on tokenized programs represented by suffix arrays. Then, Clone Miner detects the recurring configurations of simple clones using frequent itemset mining. Each of these recurring configurations of simple clones represents a possible first-level structural clone, where the entities are cloned code fragments and the relationship between the entities is "same container" (e.g., same file or directory). We use clustering to form groups of structural clones.

The process of detecting structural clones consists of iteratively performing the following four steps:

Step 1: Find recurring patterns of lower-level clones in different containers.
Step 2: Calculate coverage of the containers by these patterns.
Step 3: Cluster similar containers based on the patterns with significant coverage.
Step 4: Abstract similar containers into clones for the next higher level of similarity analysis.

Higher level structural clones are detected in a similar way, with highly similar containers found from the previous lower level clone analysis, being treated as cloned abstract entities. Using the above mechanism, we detect structural clones composed of patterns of simple clones, method clones, file clones or directory clones. *Method clones, file clones,* and *directory clones* are clones formed by the abstraction mechanism.

Automated clone detection is followed by further analysis with involvement of human expert. In recent work, we defined a query-based system for extracting and then visualization of cloning information [17].

Collaborative structural clones, such as in Figure 2(b) and Figure 3, involve message passing, which cannot be detected in the course of static program analysis only. We have conducted initial experiment to analyze patterns in traces of program execution to find message passing information. We transform traces so that the problem of finding cloned traces is reduced to the problem of detecting matching substrings and we apply the same suffix array based algorithm that is used to detect simple clones from the token representation of source code.

5. Benefits of the structural clone concept

For program understanding: As a structural clone comprises many simple clones, structural clones are bigger in size and smaller in number than simple clones, and often embody application domain or design concepts at a higher level than code fragments. Therefore, structural clones provide a simpler and higher-level window into the software similarity situation than simple clones. Related empirical studies are described in [1][2].

For plagiarism detection: Plagiarizers often take "anti-detection" measures such as renaming, reformatting, reordering, scattering (by extracting functions/classes), pruning (removing unused code), reducing generality, in-lining, changing comments, and changing string literals (e.g., error messages). The general effect of these anti-detection measures is to break the clone into smaller pieces and scatter them. Structural clone detection and analysis has a higher chance identifying such "disguised" clones, when compared to a simple clone detection/analysis.

For refactoring: Refactoring [6] can remove some clones. Structural clone concept can help in refactoring. Consider two files having a number of cloned methods but occurring in different order in each file. If we focus only on simple clones, these similar methods would

appear as candidates for the "extract method" refactoring. However, if we also focus on structural clones, we could also use "extract interface" (or "extract super/abstract class") refactoring here, in addition to the "extract method" refactoring. We can use the "template method" design pattern when we have similar methods that follow the same high-level algorithm but have implementation variations. Typically this translates to a structural clone of code fragments, contained in a method, and having the same order of appearance, but with gaps between them.

For clone detection: De Lucia et al [5] use textual similarity among whole web pages in terms of the Levenshtein distance to decide whether two web pages are clones of each other or not. In contrast, our approach considers the internal structure of the code files, in terms of simple clones shared by the files, to decide about their similarity. If the order of methods in the classes does not matter, then, unlike textual analysis, structural analysis can establish clone relation among classes that differ in the order of methods in class definition only. Without the details of the low-level similarities in the large granularity clones, it is not always straight-forward to take remedial actions like refactoring, as it requires a detailed analysis of low-level similarities. Structural clone concept may also improve the precision and recall of clone detection in general. Clone detectors usually detect clones larger than a certain threshold (e.g., clones longer than 5 LOC). Higher thresholds risk false negatives, while lower thresholds detect too many false positives. In comparison, a structural clone detector can afford to have a lower threshold than a simple clone detector, without returning too many false positives. This is because it can use the "presence of structures" as a secondary filter criterion, to filter out small clones that do not contribute to structural clones. For example, it can detect "clone structures longer than 5LOC, where entities are longer than 1LOC". This ability to "catch smaller pieces of a larger clone" increases the efficiency of clone detection. Gapped clones can be better captured as structural clones rather than simple clones, as multiple gaps may be present in a clone instance, and gap sizes can be more than one or two lines, unlike the assumption made by other gapped clone detection techniques [10][14].

6. Related work

Various aspects of research on software clones are discussed in [11]. The goal of structural clones is to raise clone analysis beyond the level of simple clones, in order to see a bigger picture of the similarity situation. Below, we discuss how others tried to address the same goal.

Clustering based analysis targets some specific types of structural clones, usually file-level structural clones. De Lucia et al [5] look for clusters of web pages inter-connected by hyperlinks according to some common pattern. Here, the structural clone entities are web pages and the relationships among them is hyperlink.

One motivation for raising the level of similarity analysis beyond simple clones is the huge number of clones normally detected by tools. Gemini [14] applies a wide range of visualization techniques to find program parts highly populated by simple clones detected by CCFinder [8]. Gemini also applies clustering heuristics to qualify files highly covered by simple clones as similar. This is very much in-line with our approach. Files that are significantly covered by common simple clones are one example of a structural clone. Our concept of structural clone described in this paper is of hierarchical structures defined at many levels of abstraction/granularity, and also cover entities of the structure related in more diverse ways than "same location" relationship.

Gemini can also identify gapped clones – a special type of structural clone where entities are code fragments and the relationship between entities is "same file". Extra conditions imposed on the relationship between the entities are that the ordering of the entities in different files should be preserved and the gap between entities should not exceed the "maximum gap size".

Work by Kapser and Godfrey [9] also aims at higher level comprehension of simple clone information produced by simple clone detector. Authors have implemented a tool that can assist human experts in higher level interpretation of simple clone information. The core concept behind tool operation is filtering clones of interest from the pool of clones reported by a clone detector. Our concept of structural clone provides a more formal ground for defining targets for simple clone analysis and required capabilities of tools that facilitate such analysis.

Structural clones address the issue of syntactic similarity in the systems. Work has also been done on semantically (or functionally) similar program units, called "high-level concept clones" [12]. Here, comments and identifiers are analyzed to determine semantic similarities. It is further suggested that domain concepts can be used to identify syntactic clones. In contrast, we try to identify domain concepts using clone analysis, making the two techniques complementary to each other.

Clone detection techniques using Program Dependence Graphs (PDG) [10] can also detect non-contiguous clones, where the segments of a clone are connected by control and data dependency information links. Such clones also fall under the premise of structural clones.

There is also connection between structural clone concept and detection, and the work on the design recovery and program understanding [2]. Large granularity structural clones provide useful insights into the program structure for better understanding of the program. We expect that some of the structural clones

may hint at important concepts behind a program. Clichés and *programming plans* [16] represent commonly used program structures, which may appear as file-level structural clones within or across software systems (product line members).

7. Conclusions

In the paper, we defined the concept of structural clones as similar program structures that can be built hierarchically, at many levels of abstraction, with similar code fragments at the bottom of such hierarchy. We have shown examples of structural clones and highlighted their benefits. We have shown how structural clone analysis extends the benefits of analysis based on simple clones only. Our work is rooted in research on software clones, but is also strongly tied to issues of concept recognition and design recovery, addressed in many research projects in the last two decades.

Research agenda:

The concept of structural clones, as discussed in the paper, poses a number of interesting open problems, which we plan to address in the future work. The first problem is to classify types of structural clones that represent significant design-level or requirements-level concepts in a subject program. We envision a number of specializations of our general definition of structural clones presented in the paper to reflect those different types of structural clones. The second problem is to develop techniques for detecting structural clones formed by elements that reside in different physical locations (files or directories), but still form important logical groupings, such as collaborative structural clones shown in Figure 2(b) and Figure 3. Our initial heuristics for detecting collaborative structural clones must be refined and experimentally validated. We are currently working on post-clone-detection filtering, analysis and visualization of structural clones with user involvement [17]. We define filters using queries expressed in terms of conceptual data model of clone information computed by Data Miner. Knowledge of clones is useful in reengineering legacy systems for reuse into software Product Lines. We work on specialized methods for detection and analysis across multiple systems. We also study software systems in a variety of domains to better understand how structural clones manifest themselves in programs. Ultimate goal is to make the best use of structural clone knowledge to improve programmers' productivity during maintenance, reengineering or to facilitate reuse.

8. Acknowledgemnts

The authors wish to thank NUS students Melvin Low Jen Ku, Zhang Yali, Goh Kwan Kee, Chan Jun Liang, and Alfred Sim who conducted various experimental studies on structural clones.

References

[1] Basit, A.H. and Jarzabek, S. "Detecting Higher-level Similarity Patterns in Programs," ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, September 2005, Lisbon, pp. 156-165

[2] Basit, H.A. and S. Jarzabek, "Data Mining Approach for Detecting Higher-level Clones in Software", accepted for *IEEE Trans. on Soft. Eng*.

[3] Biggerstaff, T.J. "Design Recovery for Maintenance and Reuse," *Computer* 22(7) , July 1989, pp. 36-49

[4] Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, 2002

[5] De Lucia, G. Scanniello, and G. Tortora, "Identifying Clones in Dynamic Web Sites Using Similarity Thresholds", Proc. Intl. Conference on Enterprise Information Systems (ICEIS'04). pp.391-396.

[6] Fowler, M. *Refactoring - improving the design of existing code*, Addison-Wesley, 1999.

[7] Jarzabek, S. Effective Software Maintenance and Evolution: Reuse-based Approach, CRC Press Taylor & Francis, 2007

[8] Kamiya, T., S. Kusumoto, and K. Inoue, "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code", IEEE Trans. Software Engineering. vol. 28 no. 7, July 2002, pp. 654 - 670.

[9] Kapser, C. and Godfrey, M.W. "Improved Tool Support for the Investigation of Duplication in Software," *Proc. 21st Int. Conference. on Software Maintenance*, 2005, pp. 305 - 314

[10] Krinke, J. "Identifying similar code with program dependence graphs", Proc. 8th Working Conference on Reverse Engineering, 2001, pp. 301-309

[11] Koschke, R. "Frontiers of Software Clone Management," *Proc. 24th Int. Conference. on Software Maintenance*, 2008, Beijing, pp. 119-128

[12] Marcus, A., and Maletic, J. I. "Identification of High-Level Concept Clones in Source Code," *Proc. 16th Intl. Conf. on Automated Sof. Eng.*. 2001, pp. 107-114.

[13] Roy, C. and J.R. Cordy, "Scenario-based Comparison of Clone Detection Techniques," *Proc. ICPC 2008, IEEE International Conference on Program Comprehension*, Amsterdam, June 2008, pp. 153-162

[14] Ueda, Y., T. Kamiya, S. Kusumoto, and K. Inoue, K. "On Detection of Gapped Code Clones using Gap Locations," *9th Asia-Pacific Soft. Eng. Conf.* ,APSEC'02, pp. 327-336.

[15] Rajapakse, D.C. and Jarzabek, S. "Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis," *Int. Conf. Software Eng., ICSE'07*, Minneapolis, USA, May 2007

[16] Rich, C., and Waters, R.C. *The Programmer's Apprentice* ACM Press, Addison-Wesley, 1990

[17] Zhang, Y. Zhang, Y., Basit, H., Jarzabek, S., Anh, D. and Low, M. "Query-based Filtering and Graphical View Generation for Clone Analysis," *Int. Conf. on Software Maintenance, ICSM'08*, Beijing, Sept. 2008, pp. 376-385