A Data Mining Approach for Detecting Higher-Level Clones in Software

Hamid Abdul Basit, *Member*, *IEEE*, and Stan Jarzabek, *Member*, *IEEE Computer Society*

Abstract—Code clones are similar program structures recurring in variant forms in software system(s). Several techniques have been proposed to detect similar code fragments in software, so-called *simple clones*. Identification and subsequent unification of simple clones is beneficial in software maintenance. Even further gains can be obtained by elevating the level of code clone analysis. We observed that recurring patterns of simple clones often indicate the presence of interesting higher-level similarities that we call *structural clones*. Structural clones show a bigger picture of similarity situation than simple clones alone. Being logical groups of simple clones, structural clones alleviate the problem of huge number of clones typically reported by simple clone detection tools, a problem that is often dealt with postdetection visualization techniques. Detection of structural clones can help in understanding the design of the system for better maintenance and in reengineering for reuse, among other uses. In this paper, we propose a technique to detect some useful types of structural clones. The novelty of our approach includes the formulation of the structural clone concept and the application of data mining techniques to detect these higher-level similarities. We describe a tool called Clone Miner that implements our proposed technique. We assess the usefulness and scalability of the proposed techniques via several case studies. We discuss various usage scenarios to demonstrate in what ways the knowledge of structural clones adds value to the analysis based on simple clones alone.

Index Terms—Design concepts, maintainability, restructuring, reverse engineering, reengineering, reusable software.

1 Introduction and Motivation

Code clones are similar program structures of considerable size and significant similarity. Several studies suggest that as much as 20-50 percent of large software systems consist of cloned code [2], [16], [40]. Knowing the location of clones helps in program understanding and maintenance. Some clones can be removed with refactoring [18], by replacing them with function calls or macros, or we can use unconventional metalevel techniques such as Aspect-Oriented Programming [31] or XVCL [27] to avoid the harmful effects of clones.

Cloning is an active area of research, with a multitude of clone detection techniques been proposed in the literature [2], [9], [16], [28], [34], [36]. One limitation of the current research on code clones is that it is mostly focused on the fragments of duplicated code (we call them *simple clones*), and not looking at the big picture where these fragments of duplicated code are possibly part of a bigger replicated program structure. We call these larger granularity similarities *structural clones*. Locating structural clones can help us see the forest from the trees, and have significant value for program understanding, evolution, reuse, and reengineering.

Figs. 1 and 2 show intuitive examples of simple and structural clones considered in this paper. In Fig. 1, we see an example of a simple clone set formed by code fragments (a1, a2, a3). Differences among clones are highlighted in **bold**.

Suppose groups (b1, b2, b3), (c1, c2, c3), . . . , (g1, g2, g3) also form simple clone sets. We observe configurations of simple clones recurring in files X1, X2, and X3 (Fig. 2a). We consider a group of such configurations a file-level structural clone set. Suppose that (Y1, Y2, Y3) and (Z1, Z2, Z3) are also file-level structural clone sets, and they form collaborative (via message passing) structures such as shown in Fig. 2b. Then, we consider a group of such patterns a higher-level structural clone set.

The examples in Figs. 1 and 2 are abstracted from clones found in Project Collaboration portals developed in industry using ASP [42] and JEE [53], and a PHP-based portal developed in our lab study [43]. Structural clones are often induced by the application domain (analysis patterns [17]), design technique (design patterns [19]), or mental templates [9] used by programmers. Similar design solutions are repeatedly applied to solve similar problems. These solutions are usually copied from the existing code. Architecture-centric and pattern-driven development encouraged by modern component platforms, such as .NET and J2EE, leads to standardized, highly uniform, and similar design solutions [53]. For example, process flows and interfaces of the components within the system may be similar, resulting in file or method-level structural clones. Another likely cause of this higher-level similarity can be the "feature combinatorics problem" [8].

Much cloning is found in system variants that originate from a common base of code during evolution. Often created by massive copying and modifying of program files, clones—small and large—are bound to occur in such system variants. Software Product Line approach aims at reuse across families of similar systems [12]. As we can

- *H.A. Basit is with the Lahore University of Management Sciences, Pakistan. E-mail: hamidb@lums.edu.pk.*
- *S. Jarzabek is with the National University of Singapore, Singapore 117543. E-mail: stan@comp.nus.edu.sg.*

Manuscript received 23 Feb. 2007; revised 22 Sept. 2008; accepted 21 Jan. 2009; published online 20 Feb. 2009.
Recommended for acceptance by P. Devanbu.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0079-0207.
Digital Object Identifier no. 10.1109/TSE.2009.16.

0098-5589/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

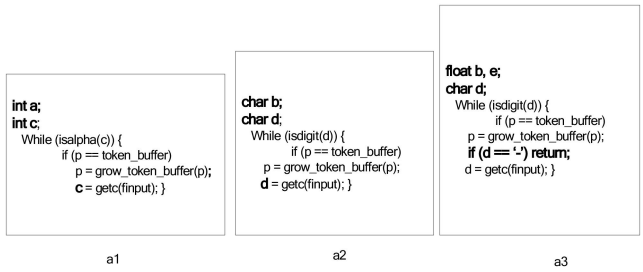


Fig. 1. A simple clone set formed by similar code fragments.

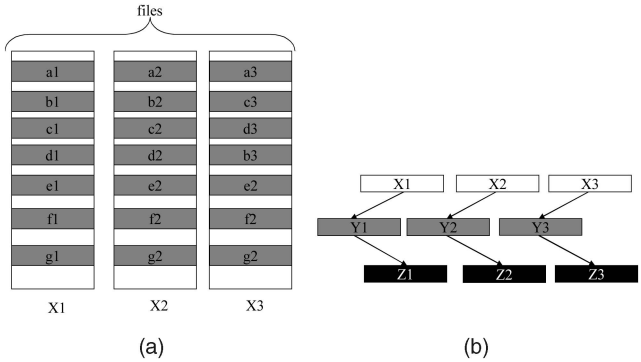


Fig. 2. Structural clones. (a) File-level structural clones. (b) Collaborative structural clones.

reuse only what is similar, knowing clones helps in reengineering of legacy systems for reuse. Detection of large-granularity structural clones becomes particularly useful in the reuse context [42].

While the knowledge of structural clones is usually evident at the time of their creation, we lack formal means to make the presence of structural clones visible in software, other than using external documentation or naming conventions. The knowledge of differences among structural clone instances is implicit too, and can be easily lost during subsequent software development and evolution.

The limitation of considering only simple clones is known in the field [51]. The main problem is the huge number of simple clones typically reported by clone detection tools. There have been a number of attempts to move beyond the raw data of simple clones. It has been proposed to apply classification, filtering, visualization, and navigation to help the user make sense of the cloning information [24], [29]. Another way is to detect clones of larger granularity than code fragments. For example, some clone detectors can detect cloned files [15], [28], while others target detecting purely conceptual similarities using information retrieval methods rather than detecting simple clones [39].

The approach described in this paper is also based on the idea of applying a follow-up analysis to simple clones' data. We observed that *at the core of the structural clones, often there are simple clones that coexist and relate to each other in certain ways*. This observation formed the basis of our work on defining and detecting structural clones. From this observation, we proposed a technique to detect some specific types of structural clones from the repeated combinations of colocated simple clones. We implemented the structural clone detection technique in a tool called Clone Miner, implemented in C++. Clone Miner has its own token-based simple clone detector [6]. Our structural clone detection technique works with the information of simple clones, which may come from any clone detection tool. It only requires the knowledge of simple clone sets and the location of their instances in programs.

We analyzed a number of commercial and public domain software systems with Clone Miner, showing that our technique can indeed find useful structural clones and is scalable.

Unique contributions of the structural clone concept are as follows. The benefits of knowing structural clones reach beyond simple clones, as structural clones comprise much bigger parts of a program (e.g., patterns of collaborating components), more meaningful to analysts and programmers than just similar code fragments. As structural clones often represent some domain or design concepts, their knowledge helps in program understanding, and their detection opens new options for design recovery that are both practical and scalable. Representing these repeated program structures of large granularity in a generic form also offers interesting opportunities for reuse [25], and their detection becomes useful in the reengineering of legacy systems for better maintenance.

When the cloned portions of code undergo arbitrary changes during evolution, they become scattered in a program. This can happen, for example, when the plagiarized code is intentionally changed to hide cloning. Such clones escape detection by simple clone detectors because of their small size, or because only some parts of the bigger cloned entity are found. Detecting structural clones improves the effectiveness of clone detection in such cases, giving a more complete picture of the cloning situation in the system.

This paper extends results published in [7] by improving the structural clone detection methodology to cover more types of structural clones, and by providing several new case studies and examples for structural clones. The description of structural clone detection techniques is detailed enough to make it possible for others to adopt and further advance our approach.

The remainder of this paper is organized as follows: In Section 2, we define types of structural clones Clone Miner can detect. Section 3 describes our clone detection mechanism. Section 4 describes the implementation of Clone Miner. Section 5 describes a mechanism to create generic representation of structural clones found in the system for better maintenance and reuse. This is further explained with two detailed case studies described in Sections 6 and 7. Section 8 describes other usage scenarios for structural clones. Sections 9 and 10 present case studies validating the usefulness of our proposed technique. Section 11 presents the related work in higher-level similarities and design recovery. Section 12 concludes the paper and presents future work.

2 Structural Clones Detected by Clone Miner

The concept of structural clones covers all kinds of large-granularity repeated program structures. Clone Miner can only find some specific types of structural clones, which are

TABLE 1
Types of Structural Clones Found by Clone Miner

Level 1	Repeating groups of simple clones	
	A	In the same method
	B	Across different methods
Level 2	Repeating groups of simple clones	
	A	In the same file
	B	Across different files
Level 3	Method clone sets	
Level 4	Repeating groups of method clones	
	A	In the same file
	B	Across different files
Level 5	File clone sets	
Level 6	Repeating groups of file clones	
	A	In the same directory
	B	Across different directories
Level 7	Directory clone sets	

listed in Table 1 and are explained later in this section as well as in the next section, where we describe clone detection process in detail.

We focused on these specific types of structural clones because their detection required only lexical analysis, making our method minimally language dependent. Furthermore, such structural clones can be easily detected by well-known data mining techniques. Finally, these types of clones can be represented in generic form with XVCL,[1] which is explained in Section 5.

Fig. 3 shows the hierarchical process of detecting higher-level structural clones given in Table 1 from the corresponding lower-level clones. The process starts from simple clones shown at the bottom of the figure. Similar to the simple clone sets (SCSets), we have method clone sets (MCSets at level 3), file clone sets (FCSets at level 5), and directory clone sets (DCSets at level 7), which consist of groups of cloned entities at successively higher levels of abstraction. The other types of clones listed in Table 1 consist of recurring groups of simple clones, method clones, or file clones. The detailed explanation and the mechanisms of detecting all these different types of structural clones are given in the following section.

3 From Simple Clones to Structural Clones

Clone Miner performs structural clone detection by finding simple clones first, and then gradually raising the level of clone analysis to larger similar program structures. The overall algorithm for structural clone detection at various levels is shown in Fig. 3.

3.1 Simple Clone Detection

Groups of similar code fragments form simple clone sets (SCSets). The output from certain simple clone detectors is in the form of clone pairs. However, we can easily form SCSets by grouping clone pairs in such a way that every member in a set is a clone of every other member [28], [46].

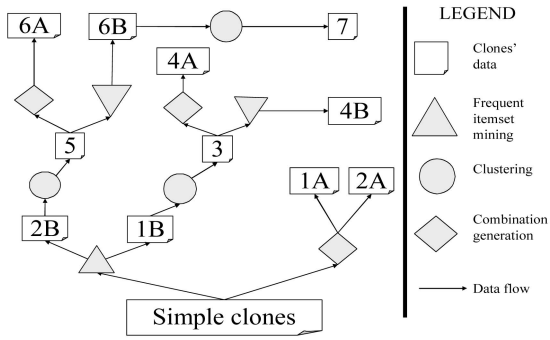


Fig. 3. A hierarchy of structural clones detected by Clone Miner and the overall detection process.

For the method-based structural clones (level 1, level 3, and level 4 structural clones from Table 1), the locations of the methods need to be provided. If the simple clone detector is based on parsing or lexical analysis, this information can be obtained directly, otherwise we can deploy program analysis tools to obtain this information.[2]

Clone Miner uses Repeated Tokens Finder (RTF), a token-based simple clone detector, as the default front-end tool [6]. RTF tokenizes the input source code into a token string, from which a suffix-array-based string-matching algorithm directly computes the SCSets, instead of computing them from the clone pairs. RTF currently supports Java, C++, Perl, and VB.net. RTF also performs some simple parsing to detect method and function boundaries. The details of simple clone detection are not in the scope of this paper and the interested readers are referred to the source [6].

3.2 Reorganizing the Simple Clone Data

Structural clones at Levels 1 and 2 are found by manipulating the simple clones' data extracted from a software system. For this, we first need to reorganize this data to make it compatible with the input format for the data mining technique that is subsequently applied on this data.

We list simple clones for each method or file, depending on the analysis level. The method level analysis only works when we know the method or function boundaries in the system and the simple clones are contained within those boundaries, without straddling them. With this arrangement of simple clones, we get a different view of the simple clones' data, with simple clones arranged in terms of methods or files. A sample of this format is shown in Fig. 4. The first data row means that the file No. 12 contains three instances of SCSet 9 and one instance each of SCSets 15, 28, 38, and 40. The interpretation is likewise for the other rows. At this stage, we can easily filter out methods or files that do not participate in cloning at all (i.e., contain no simple clones). These nonrepresentative methods or files are not apparent from the original representation of simple clones' data in terms of clone sets and clone instances.

From this data, we detect recurring groups of simple clones in different files or methods, to identify the level 2-B and level 1-B structural clones, respectively. The order in which different simple clones appear in a file or method is ignored at this stage. If code fragment A1 appears before

1. http://xvcl.comp.nus.edu.sg/.

2. For example, ctags, found at http://ctags.sourceforge.net/.

FILE ID	SCSET INSTANCES PRESENT
…	…
12	9, 9, 9, 15, 28, 38, 40
13	12, 15, 40, 41, 43, 44, 44
14	9, 9, 9, 12, 15
…	…

Fig. 4. Simple clones per file.

FILE ID	ENCODED SCSET PRESENT
…	…
12	<9,0>, <9,1>, <9,2>, <15,0>, <28,0>, <38,0>, <40,0>
13	<12,0>, <15,0>, <40,0>, <41,0>, <43,0>, <44,0>, <44,1>
14	<9,0>, <9,1>, <9,2>, <12,0>, <15,0>
…	…

Fig. 5. Transformed clones per file.

code fragment B1 in File 1, while code fragment B2 appears before code fragment A2 in File 2, with A1 and A2 being clones of each other, and the same for B1 and B2, we may still be interested in finding the clone pattern A1-B1 and A2-B2. (However, it will be somewhat harder to unify such reordered structural clones with XVCL.) The list of SCSets in a file or method is, hence, sorted to facilitate further analysis.

3.3 Finding Repeating Groups of Simple Clones

To detect recurring groups of simple clones in different files or methods, we apply the same data mining technique that is used for "market basket analysis" [22]. The idea behind this analysis is to find the items that are usually purchased together by different customers from a departmental store. The input database consists of a list of transactions, each one containing items bought by a customer in that transaction. The output consists of groups of items that are most likely to be bought together. The analogy here is that a file or a method corresponds to a transaction and the SCSets, represented in that file or method, correspond to the items of that transaction. Our objective is to find all those groups of SCSets whose instances occur together in different files or methods.

The market basket analysis uses the "frequent item set mining (FIM)" technique [21]. The difference between our problem and the standard FIM problem is that in standard FIM, the items in a transaction are considered unique, whereas in our data, one file or method may contain multiple instances of the same SCSet. We could normalize the data by removing the duplicates, but by doing so, we would miss out important information—where multiple instances of an SCSet are part of a valid structural clone across files or methods. For example, we have three instances of SCSet 9 present in both files 12 and 14 shown in Fig. 4, so 9-9-9-15 is a valid level 2-B structural clone across these two files. But if the data is normalized by removing duplicates, then we would not get this complete structural clone.

To let the FIM algorithm differentiate between different instances of the same SCSet in a given file or method, we encode the tuple <a, b> as an integer, where a is the SCSet ID and b is the occurrence index of this ID in the given file or method. For example, in the above situation, we would have three tuples <9, 0>, <9, 1>, and <9,2 > repeated across the two files, as shown in Fig. 5. This is a reversible transformation, and once we get the output from FIM algorithm, we can perform the reverse transformation to get the desired output, i.e., 9-9-9-15 as the detected level 2-B structural clone.

Mining all frequent item sets returns many frequent item sets that are subsets of bigger frequent item sets. More suitable for our problem is to perform "Frequent Closed Item set Mining" (FCIM) [21], where only those item sets are reported which are not subsets of any bigger frequent item set.

One of the input parameters for FCIM is the minimum support count, or simply *support*, of a frequent item set. In our context, it indicates the minimum number of files or methods that should contain the detected group of SCSets, for it to be reported as frequent. Due to the general nature of the FCIM problem, the standard algorithms are designed to adjust the minimum support level for FCIM. In our case, we have hard coded the *support* value at 2 so that it will report a group of SCSets, even if it is present only in two files as it could still be significant because of its length.

Another input parameter is the *minimum size* of the item set. If we assume that all the items are of equal importance, then the number of items will determine the importance of an item set. In our case, however, the number of simple clones is not the only factor that determines the importance of a repeating group. The length of the simple clones is sometimes more important than their number. Currently, these lengths are not reflected directly in the input data for FCIM. A possible future research direction for FIM and FCIM algorithms is to find weighted frequent item sets where the weight is not synonymous to the number of items.

The output from FCIM is a list of frequent item sets along with their *support* count, indicating, in our case, the number of files or methods containing those groups of simple clones that are frequent (Fig. 6). As FCIM only deals with detecting frequent item sets, an important piece of information missing here is the identification of those files or methods that contain these frequent groups of simple clones. With some postprocessing, we can find this information as well. Fig. 7 shows the algorithm for this step.

Level 1-B and 2-B structural clones can also be considered as unrestricted gapped clones [36], [50], where any number of gaps of arbitrary sizes and ordering are allowed. In future, we plan to filter the level 1-B and 2-B structural clones, where the gaps are small and the clones are more cohesive, to have more meaningful gapped clones in the conventional sense [36], [50]. Finding gapped clones in this way also provide the flexibility to detect rearranged gapped clones, where the cloned parts can occur in any arbitrary order and should not necessarily be arranged in the same way. An example of such clone is shown in Fig. 19.

FREQUENT ITEMSET	SUPPORT
9,9,9,15	2
60 44 40 42 3 49 59 63	4
…	…

Fig. 6. Sample frequent item set of SCSet with SUPPORT.

Algorithm for finding files containing the frequent itemset

1. For each frequent itemset *fit*
   a. Take the first SCSet
   b. List all files that contain instances of this SCSet
      i. For each file
         1. If *fit* is a subset of the simple clones represented in this file
            a. Keep the file in the list, else prune it
   c. Output the final list

Fig. 7. Finding files containing frequent item sets of SCSet.

Due to the limitation of the FCIM technique, only repeating groups of simple clones across different files and methods can be detected, even though such groups may be present within a given method or file. To detect level 1-A and 2-A structural clones, we apply a simple and straightforward follow-up technique to compute these locally repeating groups of simple clones separately, based on sorting and brute-force combination generation.

3.4 Finding File and Method Clones

File clone sets (FCSets at level 5) and method clone sets (MCSets at level 3) are found by the process of *clustering* from the *significant* level 2-B and 1-B structural clones, respectively. Using this mechanism, we expect to find groups of highly similar files and methods. These clusters of similar files and methods indicate even larger granularity similarities than level 1-B or 2-B structural clones, with more defined boundaries.

Clustering is a well-studied technique in the domains of data mining, statistics, biology, and machine learning [22]. It is the process of grouping the data objects into classes or clusters so that the data objects within a cluster are highly similar to one another, but are very dissimilar to data objects in other clusters, based on the attribute values describing these data objects. In our analysis, we can consider files or methods as data objects and the detected level 2-B and 1-B structural clones contained in them as their descriptive attribute values.

The *significance* of a level 1-B or 2-B structural clone set (S) is measured using the average values of two metrics at the structural clone instance (I) level, which are tailored for each specific case:

Len(I) measures the raw length of the structural clone instance. Depending on the available information, we can measure this size in terms of tokens, lines of code, or any other suitable unit.

Cover(I, C) $= (Len(I)/Len(C)) * 100$ measures the percentage coverage of the container C by the structural clone instance I. For level 1-B structural clones, C is the method that contains the particular instance and *Len(C)* is the length of that method, measured in any suitable unit, whereas for level 2-B clones, C is the file containing the clones. By default, we use number of tokens to compute the length of Level 1-B and 2-B structural clones, and also for their containers, methods and files.

It should be noted that computing *Len* for a level 1-B or 2-B structural clone is a bit complex because of the overlapping simple clones and is not simply equivalent to adding up the sizes of all simple clone instances forming the structural clone instance in a file or method.

To perform clustering of highly similar files and methods, again we have to represent the data in a suitable format for the ease of analysis. Instead of representing files and methods, we start by representing clusters. To start with, each structural clone at level 1-B and 2-B is considered as the description of a unique cluster, which contains all the methods or files containing that structural clone. The average *Len* and *Cover* values for the instances of these structural clones indicate the significance of each cluster. We let the user specify a minimum average *Len* (*minLen*) and a minimum average *Cover* (*minCover*) value to filter the *significant* clusters. These significant clusters are our file clone sets (FCSets) and method clone sets (MCSets).

In our case of clustering, it is not expected that all files or methods may become part of some cluster; a lot of these files and methods may have to be ignored as outliers. This is different from the usual clustering scenarios, where it is assumed that the majority of data objects would belong to some clusters, and a few would be detected as outliers. The former approach is also called *cluster mining* instead of *clustering*.

3.5 Finding Repeating Groups of Method Clones

In the same way as finding repeating groups of simple clones across different methods and files given in Section 3.3, we find repeating groups of method clones across different files to form level 4-B structural clones.

Another potentially useful analysis could be to detect repeating groups of Method clones across directories, but this is currently not implemented in Clone Miner. In addition, we can also possibly find the FCSets based on these repeating groups of method clones. However, these results are expected to be close to the clustering of similar files based on SCSets, as explained in Section 3.4.

Level 4-A structural clones, forming locally repeating groups of method clones within files, are again found by sorting and brute-force combination generation.

3.6 From File Clones to Directory Clones

From FCSets, we can move on to the level 6 and level 7 structural clones. For finding level 6-B structural clones, the previously formed FCSets play the same role as the SCSets in finding level 1-B and 2-B structural clones. The containers for these file clones that we currently consider are the directories. Directories provide a naïve modularization of software source code, usually reflecting the modules and subsystems of the given system. In future, we plan to let the user specify the file groupings based on the actual modularization of the software, if it is different from the directory-based grouping of files, so the directory level clones are actually substituting for the module level clones.

The transition from level 6-B to level 7 is similar to the transition from level 2-B to level 5 via clustering. As discussed earlier, we need to compute the matrices *Len* and *Cover* for clustering. One simple way is to give equal weight to each file and count the number of files that belong to a level 6-B

TABLE 2
Performance of Structural Clone Detection

	35 TOKENS	40 TOKENS	50 TOKENS
NO. OF SCSets	8,514	6,156	3,578
CPU TIME TAKEN	137s	134 s	126 s
PEAK MEMORY USED	406,772 KB	308,672 KB	235,356 KB

structural clone ($Len = \text{no. of files}$), and compare it with the total number of files in the directory to compute *Cover*. If the *Cover* of the structural clone is significant in these directories, given the user-specified threshold value for *minCover*, the directories can be abstracted as clones of each other. Depending on the nature of the analyzed system, this may unearth interesting similarity situations at a very high level.

Finally, level 6-A structural clones, representing repeating groups of file clones within directories, are detected in the same way as Level 4-A structural clones mentioned previously.

4 Tool Implementation

Clone Miner implements the structural clone detection techniques presented in this paper. Clone Miner is written in C++, and it has its own token-based simple clone detector [6]. For frequent closed item sets mining (FCIM), we are using the algorithm from [21].

For manipulation of clones' data, Clone Miner makes use of the STL containers from the standard C++ library. The output from Clone Miner is generated in the form of text files so that any visualization tool developed in the future can easily interface with Clone Miner.

For performance evaluation, we ran Clone Miner on full J2SE 1.5 source code,[3] consisting of 6,558 source files in 370 directories, 625,096 LOC (excluding comments and blank lines), and 70,285 methods, using different values of minimum clone size. For forming FCSets and MCSets, a value of 50 tokens is used for the clustering parameter *minLen*, where the *Len* is measured in terms of tokens. Likewise, for *minCover*, a value of 50 percent is used in all cases. The tests were run on a Pentium IV machine with 3.0 GHz processor and 1 GB RAM. Each time it took around two to three minutes to run the whole process from finding simple clones to the analysis of files, methods, and directories for structural clones, as mentioned above. The results are given in Table 2, with column headings indicating the minimum size of the simple clones used in each run. All times are calculated in seconds and the memory is calculated in kilobytes.

The results from Table 2 show that our technique is efficient and scalable. Considerably large systems can be analyzed using reasonable computing resources.

5 Structural Clones in Software Maintenance and Reuse

Refactoring techniques [18] are usually applied to eliminate clones from the source code [3], [4]. However, elimination of clones may not always be feasible. One reason could be the risk involved in changing a working piece of code. Only changes critical for maintaining correct software functions may be acceptable for business reasons [13]. In other situations, clones might be created intentionally, for better performance or design standardization (e.g., on J2EE or .NET). Sometimes clone refactoring may conflict with other important design goals. Yet other clones may exist in code because of limitations of a programming language [26], [32]. Because of the above reasons, many clones cannot be completely removed from programs. This is particularly true for large-granularity structural clones discussed in this paper, such as similar files or directories.

TABLE 3
Basic XVCL Commands

COMMAND	DESCRIPTION
<**x-frame** *name*>	Denotes the meta-component body that contains the code and other XVCL commands.
<**adapt** *name*>	Adapts the *name* meta-component to the current meta-component.
<**break** break-name>	Breakpoint where changes can be made by ancestor meta-component via <**insert**>.
<**insert** break-name>	Replaces breakpoints *break-name* with insert-body.
<**set** var-name = value>	Assigns list of values to *var-name*.
@expression	Evaluate *expression* and replace @ command with result.
<**select option** = *var-name*>	Select pre-defined options based on *var-name* list of values.
<**while** *var-name*>	Iterates over the while body and in each iteration i, replaces *var-name* with the i'th value defined in **set**.

We can still effectively achieve nonredundancy, with benefit for maintenance, by unifying clones with generic representations built at the metaprogram level. The technique is based on XVCL,[4] which is a method and tool for managing changes during software evolution and reuse [25]. The technique targets better software maintenance by providing a metalevel source code representation free of clones, and keeping "good" clones untouched in the actual program derived from the metaprogram.

We represent each clone set, either simple or structural, found in a system under maintenance as a generic, adaptable XVCL metacomponent. The variations between clone instances are expressed as deltas from their generic metacomponent, which is handled by suitable XVCL commands, as listed in Table 3. XVCL Processor instantiates generic XVCL metacomponents in their variant forms, as required in the subject system. In Fig. 9, S-i (where $i = 1, 2, 3$) are instances of a clone set, and S-gen is their generic metacomponent representation in XVCL.

A hierarchy of generic metacomponents reflects a hierarchy of structural clones (Fig. 3): Generic metacomponents for simple clones are at the bottom of that hierarchy, metacomponents for structural clones formed as configurations of simple clones appear above them, and so on. Small XVCL metacomponents are combined to form bigger ones, and eventually represent the structure of the entire system.

3. Downloadable from www.java.sun.com.

4. http://xvcl.comp.nus.edu.sg/.

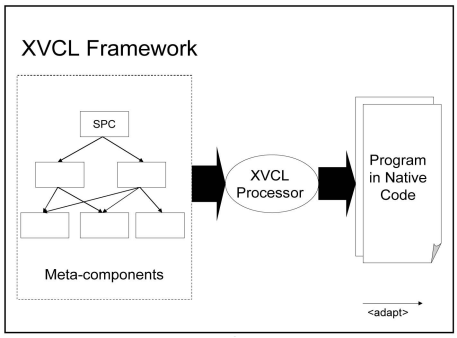


Fig. 8. Generating a system using XVCL.

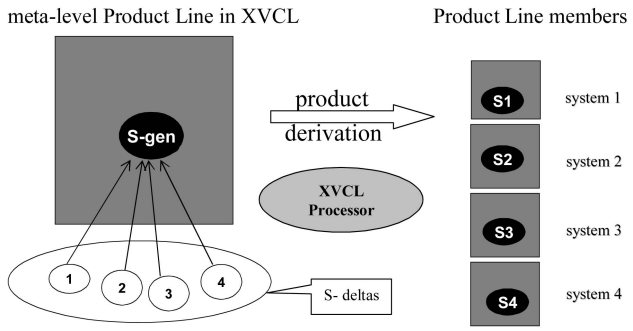


Fig. 9. Generating system variants using XVCL.

During this composition of metacomponents, variability is injected into the generic metalevel structures to handle the differences in otherwise similar program structures. The XVCL Processor reads and processes how the metacomponents are configured, after which it reconstructs the program in its native language such as Java or C++.

Structural clone detection can directly help in creating generic metacomponents. Instead of manually analyzing code for these higher-level clones [26], [27], users can automatically identify structural clones using our proposed technique.

Fig. 8 shows the overall solution: SPecifiCation metacomponent (SPC) defines deltas for metacomponents below that represent clone sets in generic form. FCSets are potential candidates for metacomponents representing groups of cloned classes (the middle layer metacomponents in Fig. 8, below SPC). The bottom layer metacomponents in Fig. 8 represent the metafragments metacomponents. Level 2-B or 4-B structural clones—representing repeating groups of simple clones or method clones across files—that are stand alone, i.e., whose constituent simple clones are not present in other files not having the complete group, can be unified with a single metacomponent. However, widely spread level 2-B or 4-B structural clones cannot be unified with a single metacomponent. In this case, separate metacomponents need to be created for the constituent simple or method clones. These metacomponents are then **<adapt>**ed by the metaclass metacomponents.

The gain due to XVCL is that we can understand and maintain a nonredundant metalevel system representation, with changes automatically propagated to the subject system. Metalevel representation is smaller and conceptually simpler than the actual system under maintenance.

The essence of the above-explained XVCL approach to maintenance is adaptive reuse of generic metacomponents that form metalevel representation of a single system.

Product Line [12] approach is based on adaptive reuse of program components across a family of similar systems. It is common practice to start a Product Line only after having developed a few similar products. Such system families originate from the same code base that is modified in various ways for each specific product. Clone detection and generic XVCL metacomponents have a role to play in this context too. In addition to unifying clones recurring within a single system (Fig. 8), we also identify and unify clones across the systems, as shown in Fig. 9. In [25] (Part II), we described how we apply XVCL to manage multiple software systems arising from evolution, in reuse-based way, as a Product Line. In reuse context, structural clone detection becomes an important element of reengineering a legacy system for reuse into Product Lines.

Clone Miner can be run on the subject systems to find the similarities among them. The mechanism of finding similarities across systems is much the same as the mechanism of finding similarities within a system. Clone Miner treats the group of systems as one big system, and the user can optionally ignore clones found only within a single system when performing analysis of multiple systems. The purpose is to find large-granularity reusable assets across systems that can be candidates for Product Line architecture building blocks and, eventually, help in converting a family of similar systems to a Product Line.

Identifying higher-level structural clones across different versions of a product can also be helpful in understanding the product evolution characteristics, like identifying parts that are more stable as compared to the rest of the system.

6 Buffer Library Case Study

Here, we describe in detail the steps in building generic metacomponents for a group of files that were detected as an FCSet by Clone Miner. These files are part of the Java Buffer Library (java.nio.* package of J2SE 1.5), for which a complete XVCL solution was built with manual analysis of similarities in an earlier case study [26], [27]. Therefore, we could easily validate if the results of automatic structural clone detection by Clone Miner reveal the same high-level similarities.

First, we present a brief overview of the library. A buffer contains data in a linear sequence for reading and writing. Buffer classes differ in features such as a buffer element type, memory allocation scheme, byte ordering, and access mode. Each legal combination of features yields a unique buffer class. That is why, even though all the buffer classes play essentially the same role, there are 74 classes in the Java Buffer library. We can consider Buffer library as a special kind of a Product Line, whose member "systems" are—similar, but also different—buffer classes. While it is rather unusual Product Line, clone detection and construction of generic XVCL metacomponents for clone classes reflect practicality of our approach in general.

In [26] and [27], it was found that the 71 buffer classes (all classes except *Buffer*, *MappedByteBuffer*, *StringCharBuffer*)

```
{  final int[] hb;
   IntBuffer(int mark, int pos, int lim, int cap,
       int[] hb, int offset) {  … }
   IntBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static IntBuffer allocate(int capacity)
   { …  return new HeapIntBuffer(capacity) }
   public static IntBuffer wrap(int[] array) {… }
   public abstract IntBuffer slice();
   public abstract IntBuffer duplicate();
    … }
```

Fig. 10. Differences of IntBuffer from other numeric buffers.

```
<x-frame [T]Buffer outfile = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable
{  final @type[] hb;
   Buffer(int mark, int pos, int lim, int cap,
       @type[] hb, int offset) {  … }
   @TypeBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static @TypeBuffer allocate(int capacity)
   { …  return new Heap@TypeBuffer (capacity) }
      … }
```

Fig. 11. [T]Buffer metacomponent for numeric buffers.

could be clustered into seven groups of highly similar classes as described below:

[T]Buffer contains seven buffer classes of type T. T denotes one of the buffer element types, namely, *Byte, Char, Int, Double, Float, Long, Short*.

Heap[T]Buffer contains seven *Heap* classes of type T.

Direct[T]Buffer[S | U] contains 13 *Direct* classes. U denotes *native* and S denotes *non-native* byte ordering.

Heap[T]BufferR contains seven *Heap* read-only classes.

Direct[T]BufferR[S | U] contains 13 *Direct* read-only classes.

ByteBufferAs[T]Buffer[B | L] contains 12 classes providing views T of a Byte buffer with different byte orderings. Here, T denotes buffer element type except *Byte*, while B denotes *Big-endian* and L denotes *Little-endian* byte ordering.

ByteBufferAs[T]BufferR[B | L] contains 12 read-only classes providing views T of a *Byte* buffer with different byte orderings (B or L). Here, T denotes buffer element type except *Byte*, while B denotes *Big-endian* and L denotes *Little-endian* byte ordering.

Detecting these seven groups of similar classes was fundamental in forming the generic solution of the library with XVCL, as groups represent important concepts in the buffer domain. Each of these groups is represented by a template in XVCL from which all members of the group could be generated by providing the appropriate parameters and other specific details.

By performing the simple clone detection of the Buffer Library with Clone Miner, 52 SCSets were detected, with the minimum clone size of 20 tokens. As expected, the clones were distributed in all the analyzed files. From the detected simple clones' data, Clone Miner discovered 38 level 2-B structural clones. Making 38 initial file clusters based on these structural clones and pruning these clusters using the default threshold *minLen* and *minCover* values (50 tokens and 50 percent, respectively, where *Len* is measured in tokens), only seven clusters were left that formed the seven final FCSets. These seven FCSets matched exactly with the seven groups of similar classes that were found manually, as mentioned above.

This shows that Clone Miner correctly detects higher-level similarities and can be a useful tool for the reengineering projects aimed at creating a generic representation of a given system with XVCL.

We illustrate the technique for building generic metacomponents for clone sets with the FCSet consisting of seven classes in the group **[T]Buffer**, where [T] refers to the seven primitive data types in the Java language, and shows how Clone Miner's output can guide us to form a generic representation of the system with XVCL. These classes include *IntBuffer, ShortBuffer, FloatBuffer, LongBuffer, DoubleBuffer, CharBuffer*, and *ByteBuffer*.

A closer analysis of the Clone Miner's output reveals that numeric type buffer classes (*IntBuffer, ShortBuffer*, *FloatBuffer*, *LongBuffer*, and *DoubleBuffer*) differ from each other in type names only, i.e., all the methods in each of these files are either exact or parameterized simple clones of the corresponding methods in other files. This is evident from the same list of simple clones and method clones present in these files, and an almost 100 percent coverage of these five files by this repeating group of simple clones. *CharBuffer* and *ByteBuffer* classes have extra methods that are missing from the other classes in the group and also other differences, indicated by a lesser *Cover* value for the level 2-B structural clone defining this group of seven files. Fig. 10 highlights in **bold** the places where *IntBuffer* differs parametrically from other numeric buffers.

Fig. 11 shows a generic metacomponent **[T]Buffer** parameterized by two XVCL variables, namely, "Type" and "type." These variables can easily be spotted by using a simple *diff* utility on the five files. The top level metacomponent **SPC**, where we specify the actual values for the metavariables, contains the following code to derive *IntBuffer* class from the generic **[T]Buffer**:

```
<x-frame SPC >
<set Type = Int />
<set type = int />
<adapt [T]Buffer />
```

Attribute "outfile" in metacomponent **[T]Buffer** defines the name of a file, IntBuffer.java, where we want XVCL Processor to emit code for this class.

Using the same metacomponent **[T]Buffer** and the following code in the **SPC,** we can derive all five numeric buffer classes:

```
<x-frame SPC >
<set Type = Int, Short, Float, Long, Double />
<set type = int, short, float, long, double />
<while Type, type>
    <adapt [T]Buffer/>
<while>
```

Now we extend the generic solution to incorporate classes CharBuffer and ByteBuffer that have more differences with the numeric buffer classes, but still enough similarities to be clustered in the same FCSet by Clone Miner and to be derived from the same metacomponent **[T]Buffer**.

```
public abstract class CharBuffer extends Buffer
extendsBuffer implements Comparable,CharSequence
{       final char[] hb;
        CharBuffer(int mark, int pos, int lim, int cap,
              char[] hb, int offset) {   … }
        public String toString() { different implementation }
        many extra methods in Char Buffer:
        public static CharBuffer wrap(CharSequence csq) { }
  etc. }
```

Fig. 12. Differences between classes IntBuffer and CharBuffer.

Fig. 12 shows some of the places where CharBuffer differs from other numeric buffer classes. For CharBuffer, we must update "implements" clause (the second line), redefine implementation of method **toString()**, and insert extra methods required in class CharBuffer, but not needed in numeric buffer classes.

The updated metacomponent **[T]Buffer** is shown in Fig. 13, while the updated **SPC** now looks like the following:

```
<x-frame SPC >
<set Type = Int, Short, Float, Long, Double, Char />
<set type = int, short, float, long, double, char />
<while Type, type>
   <select option = Type>
     <option Char>
       <adapt [T]Buffer />
   <insert implements >
     ,CharSequence
   <insert toString >
     implementation of method toString() for CharBuffer
   <insert extraMethods >
     implementation of extra methods for CharBuffer
     <otherwise>
       <adapt [T]Buffer />
</while>
```

<**option** Char> of <**select**> defines customizations required for class CharBuffer, but not needed in other classes. We use <**insert**> commands in the <**adapt**> body to update the "implements" clause, to override the implementation of method **toString()**, and to add extra methods. <**break** toString> in metacomponent **[T]Buffer** contains implementation of method *toString()* for all five numeric buffer classes as default. If no <**insert**> affects the <**break**>, the default contents of the <**break**> is processed as if there was no <**break**>. Any <**insert**> affecting the <**break**> overrides the default contents of the <**break**>.

```
<x-frame [T]Buffer outfile = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable <break
implements>
{ final @type[] hb;
   Buffer(int mark, int pos, int lim, int cap,
       @type[] hb, int offset) {   … }
<break toString >
   implementation of method toString() for numeric
classes
<break extraMethods >
   implementation of methods specific to CharBuffer
}
```

Fig. 13. New *[T]Buffer* metacomponent to incorporate changes for *CharBuffer*.

```
<x-frame SPC >
<set Type = Int, Short, Float, Long, Double, Char, Byte />
<set type = int, short, float, long, double, char, byte />
<while Type>
     <select option = Type>
          <option Char>
               <adapt [T]Buffer>
                    customizations for CharBuffer
          <option Byte>
               <adapt [T]Buffer>
                    customizations for ByteBuffer
          <otherwise>
               <adapt [T]Buffer/>
     </select>
</while>
```

Fig. 14. **SPC** to derive seven *[T]Buffer* classes.

At the bottom of the <**select**> there is <**otherwise**> clause that caters for all the numeric buffer classes that are derived from metacomponent **[T]Buffer** as shown before, without any further customizations. <**otherwise**> is processed five times, in iterations when none of the other <**option**>s under <**select**> is processed, producing five numeric buffer classes.

Class *ByteBuffer* has yet other extra methods, not found in other **[T]Buffer** classes. The solution is the same as for extra methods in class *CharBuffer*, and the resulting **SPC** is shown in Fig. 14 (metacomponent **[T]Buffer** is the same as in Fig. 13).

The overall structure of XVCL representation for [T]Buffer classes is shown in Fig. 15.

The size of the XVCL representation was 68 percent smaller than buffer classes in Java (in terms of lines of code, without blanks or comments). XVCL representation was also simpler to understand and to work with, than its Java counterpart, due to a smaller number of elements a programmer must comprehend. A conceptual element in a Java program is a class, method/constructor, declaration section, or a fragment of method/constructor implementation that plays a role in the Buffer domain or in class design. In the entire library, there were 1,385 Java conceptual elements comprising 6,719 LOC versus 324 conceptual elements comprising 2,080 LOC in the XVCL representation. This simplification was achieved by representing each of the clone sets in a generic form.

In XVCL, classes can be comprehended in groups rather than individually. One can see exact similarities and

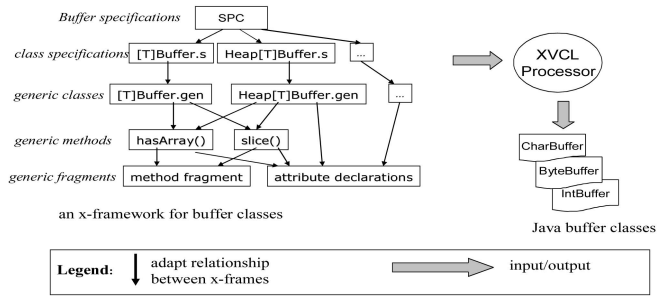


Fig. 15. Deriving seven [T]Buffer classes from generic XVCL representation.

differences among specific classes in a group. This information helps in reusing existing classes when designing new buffer classes, and reduces ripple effects of changes: If we want to change one class, we can check if the change also affects other similar classes.

In a controlled experiment, we extended the Buffer library with a new type of buffer element, *Complex* [26]. We compared the effort involved in changing Java classes and XVCL representation. In Java, class *ComplexBuffer* could be implemented based on the class *IntBuffer*, with 25 modifications that could be automated by an editing tool and 17 modifications that had to be done manually. On the other hand, in the XVCL representation, all the changes had to be done manually, but only five modifications were required. To implement class *HeapComplexBuffer*, we needed 21 "automatic" and 10 manual modifications in Java versus three manual modifications in the XVCL. To implement class *HeapComplexBufferR*, we needed 16 "automatic" and five manual modifications in Java versus five manual modifications in XVCL.

Despite potential benefits, applying XVCL also induces certain complexities. Designing generic, reusable, and maintainable solutions is more difficult than building a concrete program. A concrete program is only a prerequisite for applying XVCL. An XVCL representation is expressed at two intermixed levels, in base programming language(s), and wrapped in XVCL metacomponents. Thinking about a program in terms of such a mixed-level representation is different and more challenging than thinking in terms of conventional program. This creates extra difficulties.

The detailed mechanism of XVCL can be found on the XVCL website and has been described in [25]. XVCL case studies in lab settings [5], [27], [53] and industrial settings [42] show the usefulness of the technique in terms of improved maintainability and enhanced reuse of the software.

7 Web Portal Case Study

In this study, we analyzed a J2EE web portal Common Application Platform (CAP-WP), developed by our industry partner ST Electronics (Info-Software Systems) Pte. Ltd. Similarities induced by the J2EE architecture and patterns (such as MVC) were expected. In addition, results of previous study to find similarity patterns in CAP-WP by manual inspection were known to us [53]. Therefore, again, we had an objective data against which we could compare the results obtained by Clone Miner.

CAP-WP is composed of 25 portlet modules. Each module contains all the code required for implementing a particular core feature in CAP-WP (i.e., Forum, News, etc.). Each module has the same architectural design, which is promoted by the five-layer J2EE reference model.

Files belonging to the presentation, business logic, and data access components were analyzed with Clone Miner. First, the similarity patterns across the different modules were analyzed. Given the repetitive structure of the system architecture, a high level of similarity across modules was expected and was also found by the analysis.

From the manual analysis of the structural clones found by Clone Miner, it was observed that similarity patterns occur across files with parametrically similar file names. Taking [p]*ViewHandler* group of files as an illustration, with [p] referring to the module name, all 25 modules contained this type of files and each file possessed a similar program code structure, as shown in Fig. 16. The differences between each view handler class lie in the ***bold*** portions and the contents of the method *render()*, which is highlighted in the figure. Such similarity patterns of class structure were common across the different modules in CAP-WP.

```
class AddContentTypeViewHandler implements ViewHandler {
        private PortletConfig _config;
        public void init (PortletConfig config) {
                config = config;}
        public void render (PortletRequest req, PortletResponse resp,
                                AppResult [] result) throws ViewException {
                ...     }
        public void renderException (PortletRequest req, PortletResponse resp,
                                PortletException ex) throws ViewException {
                PortletContext context = _config.getPortletContext ();
                PortletRequestDispatcher disp =
                        context.getNamedDispatcher ("ADMIN_ERROR");
                req.setAttribute (RequestKey.EXCEPTION, ex);
                req.setAttribute (RequestKey.ERROR_PAGE_TITLE,
                        "Add Content Type Fail");
                try {disp.include (req, resp);}
                catch (Exception e) {
        Log.error ("AddContentTypeViewHandler.renderException - Unable to
                                render error page: " + e.getMessage ());
                        throw new ViewException
                (ErrorCode.ERROR_CREATE_CONTENT_TYPE, e);}
        }
}
```

Fig. 16. View Handlers similarity pattern in CAP-WP.

Some of the modules had more than one view handlers. For example, in the Admin module, there were four view handler classes:

AddContentTypeViewHandler, UpdateContentTypeViewHandler, DeleteContentTypeViewHandler, and *ViewContentTypeViewHandler*.

All of these files, however, had the same class structure as shown in Fig. 16, and were detected as instances of the same FCSet by Clone Miner.

The analysis of similarity patterns in CAP-WP done with Clone Miner agreed with the analysis presented in [53]. The only difference was that in [53], it is stated that intra-portlet similarities were mostly confined to simple clones. This was, however, not the case in the Clone Miner analysis as significant intra-portlet similarities in the form of structural clones were also detected.

As explained by the developers of CAP-WP, the similarity patterns found were largely caused by the common architectural design used and the standardization of interfaces as stated in the Portlet API (each portlet was built based on implementing the interfaces of this API, resulting in similar class structures and methods). The developers did acknowledge that the design of the system could be improved and made more maintainable by unifying the identified similarity patterns. This fact substantiates the benefits that can be achieved using analysis of structural clones for the redesigning of a system.

Using the file clustering mechanism of Clone Miner, six file clone sets (FCSets) were identified:

[X]FileAction: six files that belong to the *action* package. **X** refers to the operations *Delete, Download, Update, Upload, UploadMultiple*, and *View*.

[X]FileACLAction: group of two files, also from the *action* package, where **X** refers to operations *Add* and *Delete*.

[X]FileRequestHandler: group of six files from the *request* package, where **X** stands for operations *Delete, Download, Update, Upload, UploadMultiple*, and *View*.

[X]FileACLRequestAction: another group of two files from the *request* package for operations **X** = *Add* and *Delete*.

[X]FileViewHandler: group of six files in the *view* package, where **X** refers to operations *Delete, Download, Update, Upload, UploadMultiple*, and *View*.

[X]FileACLViewAction: again a group of two classes in the *view* package, supporting operations **X** = *Add* and *Delete*.

In [53], a generic representation with XVCL was built to unify similarities in the entire CAP-WP in order to improve maintainability of CAP-WP and reusability of CAP-WP modules in other similar portals. In that analysis, portlet classes corresponding to different actions but present in the same portlet (e.g., *AddFileACLAction.java* and *DeleteFile ACLAction.java*) were not considered for unification. Clone Miner, however, identified similarity between each class in the group *[**O**]FileACLAction* to be as high as 78 percent, where [O] refers to the action. This illustrates that some obvious similarities may be missed by manual analysis, even with good domain knowledge, and automated tools help in guaranteeing that all the similarities are detected and analyzed.

8 Other Usage Scenarios for Clone Miner

8.1 Improved Clone Detection

Detecting structural clones improves the effectiveness of clone detection in general. Simple clone detectors usually detect clones larger than a certain threshold (e.g., clones longer than five LOC). Higher thresholds risk false negatives, while lower thresholds detect too many false positives. In comparison, Clone Miner can afford to have a lower threshold for simple clones, than a stand-alone simple clone detector, without returning too many false positives. This is because it can use the grouping as a secondary filter criterion to filter out small clones that do not contribute to structural clones. These small simple clones may just be noise when considered individually, but when they are combined to form structural clones, they can indicate bigger cloned entities. These small clones may otherwise slip detection by simple clone detectors working with bigger simple clones only. Such situations arise when a programmer clones a larger piece of code in the system, but with successive changes during evolution, the cloned parts become scattered with arbitrary sized gaps appearing in the middle. A simple clone detector may only detect the smaller cloned fragments of the bigger clone, but Clone Miner may identify the original complete clone.

8.2 Program Understanding

Design recovery (or reverse engineering) [3], [4] is about analyzing programs to identify important design information or concepts in programs. Such information is invaluable in program understanding, maintenance, reuse, and reengineering. Our work on detection of structural clones is related to the issues of concept recognition and design recovery. By enhancing concepts behind a program, detecting structural clones appears as a novel, general, and scalable approach to design recovery.

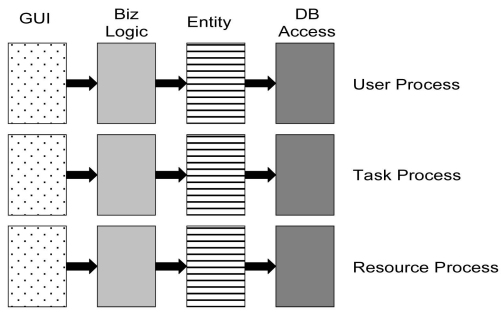


Fig. 17. Similar process flows.

Currently, Clone Miner can find structural clones that are repeating groups of simple, method, or file clones, related to each other by the "same location" relationship. As shown by the case studies presented in Section 9, these structural clones found by Clone Miner represent important domain/design concepts that can lead to better domain analysis and program understanding.

In future, we plan to address yet other relationship types to find other types of structural clones. Ongoing work in this direction is to trace the method calls from FCSets. Depending on the similarity between the files in an FCSet, many similar method calls emerging from these files can be expected. If most of the calls end up to the same file, or to similar files that are part of another FCSet, we have hit upon a process flow structural clone in the system. By further tracing method calls, leading to or emerging from these related structural clone sets, the complete pattern of process flow can be extracted, as depicted in Fig. 17. Here, boxes represent files, and the same shading indicates membership to the same FCSet. The entire process can be repeated to trace other similar process flows within the system.

Identification of these similar process flows is important for understanding a system's design, as we only need to understand one instance of this process flow, and information about the other copies could be deduced from that.

8.3 Change Impact Analysis

For efficient maintenance of software, good understanding of the system is required to deal with ripple effects of changes and update anomalies. A system having lot of clones hinders change [41]. To avoid update anomalies and to assess the impact of change, it is helpful to find all the clones affected by change. By studying the structural clones covering the potential point of change, a broader picture of the change impact can be seen as compared to analyzing only the simple clone containing that point of change.

Another scenario is that of a bigger simple clone that evolved over time into a structural clone consisting of a group of much smaller simple clones with arbitrary sized gaps in-between, induced by evolution. In such situations, structural clone detection has a higher chance of detecting all copies of the clone containing the point of change than simple clone detection.

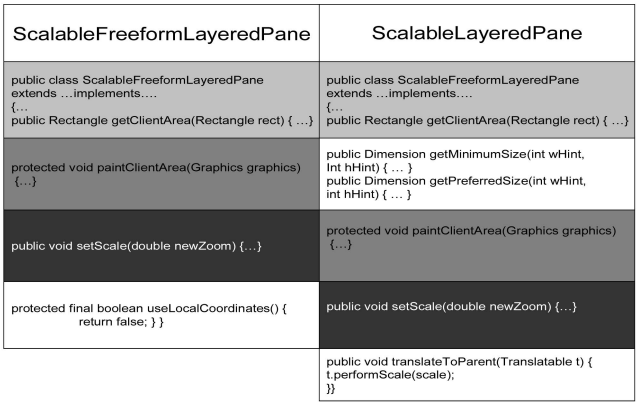


Fig. 18. A structural clone found in Eclipse GEF depicting a gapped clone.

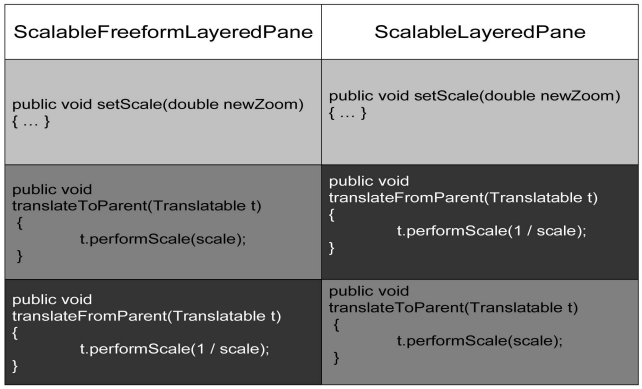


Fig. 19. Reordered structural clone found in Eclipse GEF.

8.4 Refactoring

Various restructuring or refactoring techniques can be applied to improve the design of a software system without changing its functionality [18]. Analysis of structural clones is helpful in locating places where high-level duplication is present that can be restructured or refactored.

Analysis of structural clones can be useful in redesigning the legacy code to enhance its maintainability. A good point to start is to analyze the FCSets. After choosing some FCSets for refactoring, simple clones or method clones within these groups of files can be more easily refactored because of the context information. It may also be possible to apply several small refactorings simultaneously, for example, moving together several cloned methods to the parent class, or simply changing the inheritance structure to remove duplicates. Having only the knowledge of simple clones, possibility of making such bigger changes is not very apparent and one has to go step by step with the risk of missing the bigger picture altogether.

Analysis can also be done at the level of code fragments, methods, or directories level, depending on how intense the cloning is and how major restructuring is practical.

Another scenario where the structural clone detection by Clone Miner can lead to a more effective refactoring as compared to the detection of simple clone only is the situation where "template method" design pattern is applicable. 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 into a level 1-B structural clone with simple clones appearing in the same order across different methods.

8.5 Plagiarism Detection

Plagiarizers may take intentional measures to avoid detection by breaking up the cloned code into smaller pieces and scattering them. These antidetection measures may include renaming variables, reformatting code layout, reordering statements, extracting or in-lining functions, changing comments, and string literals. Clone Miner may have a better recall in detecting such "disguised" clones as compared to a simple clone detector.

9 Summary of Other Case Studies

We performed a number of other case studies on open source software to explore further the types of similarities that could be unearthed from the structural clones found by Clone Miner at various levels. Systems that we studied include

- Eclipse Graphical Editing Framework
- Eclipse Visual Editor
- OpenJGraph 0.9.2
- J2ME Wireless Toolkit 2.2
- Java Pet Store 1.3.2

All of these systems can be downloaded from websites of Java Technology[5] and Sourceforge.[6]

9.1 Eclipse Graphical Editing Framework

The Eclipse Graphical Editing Framework (GEF) is an open-source development platform for the creation of graphical editors from an existing application model. Clone Miner detected some interesting structural clones in this system. Fig. 18 shows a structural clone present in files *ScalableFreeformLayeredPane* and *ScalableLayeredPane* that has gaps in between the corresponding locations of the constituent simple clones. The parts of the file having the same shade indicate simple clones, which are actually similar methods. This example also shows that the gapped clones are better captured as structural clones rather than simple clones, as gap sizes can be more than one or two lines, as has been assumed by other gapped simple clone detection techniques [36], [46]. Using these techniques, when the possible gap size is increased, more false positives are reported, which is not the case with the Clone Miner technique.

A probable reason for such gapped clones occurring in different files is the piece meal copy-paste cloning; initially some part of a file is copied, and later on, the programmer realized that more sections of the previous files can also fit in the new situation, so more sections are copied, not necessarily in the same order as they occurred in the original file, facilitated by the copying of methods whose ordering does not affect their functionality. Another example from the same files is shown in Fig. 19, where

5. http://java.sun.com.
6. http://sourceforge.net.

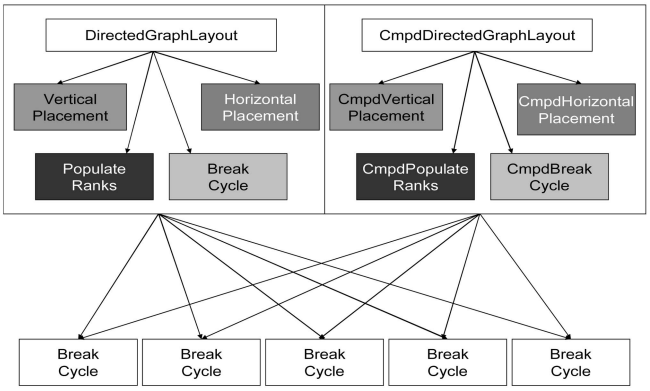


Fig. 20. Structural clone based on method calling found in Eclipse GEF.

the ordering of SCSets that constitute a level 2-B structural clone between the two files is different in both files.

An even higher-level interesting structural clone was also found in GEF, with some manual analysis based on Clone Miner's output. Shown in Fig. 20 is a structural clone that is based on method calling relationship. All of the corresponding five files in the two modules, represented by container boxes, are structural clones of each other, but not all of them are present in the same directory. Rather, the first file in each module (*DirectedGraphLayout* and *CmpdDirectedGraphLayout*) calls methods from each of the other files in the same module, while all five files in each module collectively call methods from the group of files in the bottom box, which are not part of the structural clone. The arrows in Fig. 20 represent method calling. In addition to the automatic clone detection, some manual analysis is required to dig this type of complex structural clones, as Clone Miner currently does not support detection of dynamic relationships between cloned entities. This feature is currently being implemented with some promising initial results.

9.2 Eclipse Visual Editor

The Eclipse Visual Editor (VE) is an open-source development platform for creating GUI builders from an existing model. It provides extensible tool implementations for many different products. VE was analyzed with Clone Miner, again with some interesting results. Some of the detected structural clones seem to be peculiar to the domain.

Manual analysis revealed that since the complex graphical objects are actually composed of simpler objects, the operations performed on complex graphical objects are composed of similar operations performed on the constituent simple objects. This results in the emergence of structural clone sets that also have much interclass similarity. This can be understood from the example in Fig. 21.

In this example, there are three structural clones representing repeating groups of simple clones. SCSets 1, 2, 5, and 7 are common to all three structural clones. These SCSets are actually the basic methods that are used across some drawing classes. For example, SCSet 1 is a method that checks the validity of the given input, i.e., height, width, x-coordinate, and y-coordinate. SCSet 2 is a method that ensures the drawing of a nicely formed shape based on the input values, provided they are valid.

STRUCTURAL CLONE	SCSETS
1	1,2,5,7,8,9
2	1,2,4,5,7,10
3	1,2,3,4,5,6,7

Fig. 21. Similar structural clones.

Looking at the details, we found that SCSet 4 is a constructor class that indicates inheritance. It was also noted that files having structural clones 2 and 3 were inherited classes of some files having structural clone 1. SCSets 3, 6, and 10 are new specialized drawing methods added in by the inherited class. For example, SCSet 3 is a method that validates the depth of an object, which is previously not used in classes that draw two-dimensional objects.

This example illustrates that interesting design information can be revealed by analyzing structural clones that can help in program understanding, instead of just looking at simple clones. There is also the possibility of capturing certain code patterns based on the nature of structural clones they contain, (similar to micropatterns [20]) so that in future when we see the occurrence of this type of structural clones, we can guess about the code pattern it is suggesting.

9.3 OpenJGraph

OpenJGraph is an open-source Java library to work with graphs. It contains the well-known graph algorithms like *ShortestPath*, *MinimumSpanningTree*, etc. Considerable similarity was detected between these algorithms in the form of structural clones, with 82 percent of the *BreadthFirstTraversal* algorithm being similar to the *DepthFirstTraversal* (implemented with a queue and a stack, respectively) and 94 percent of the *MinimumSpanningTree* algorithm being similar to the *ShortestPath* algorithm. The percentages are based on the simple clones present in the system.

9.4 J2ME Wireless Toolkit 2.2

J2ME Wireless Toolkit 2.2 (WTK) is a Java development platform for wireless applications, mainly for mobile devices such as cell phones and PDAs.

Structural clones found by Clone Miner in this application went beyond forming groups of similar files. It was found that directory-level cloning is present as well, where similar files in different directories implement similar functionalities.

Analysis showed file-level cloning between files that follow a similar programming structure. It was noticed that these files were actually part of different modules, where they performed a similar functionality. A sample level 6-B structural clone is shown in Fig. 22. Here, file clones are shown in similar shade between the directories *Socket* and *Datagram*.

9.5 Java Pet Store 1.3.2

Java Pet Store 1.3.2 is developed as a model application for the Java 2 Platform to demonstrate the capabilities of J2EE for enterprise applications. It provides a template to rapidly develop enterprise solutions.

Structural clones found by Clone Miner in this application also include directory-level cloning as shown in Fig. 23,

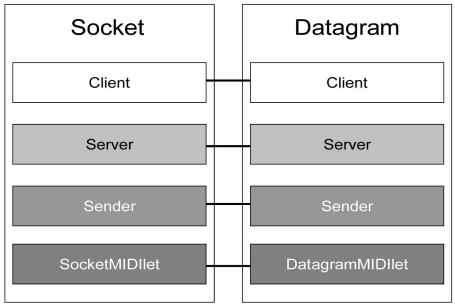
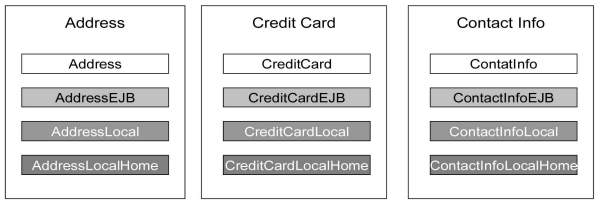


Fig. 22. Directory-level cloning in J2ME WTK.



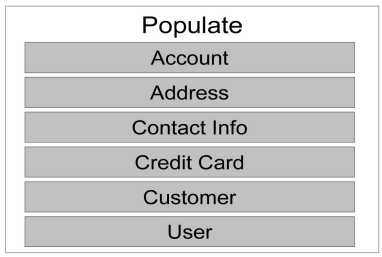


Fig. 24. File-level cloning in Pet Store.

where the directories *Address*, *ContactInfo*, and *CreditCard* contain similar files. Cloning relationship is shown by the same shading of the files. The reason for this rampant cloning in this model application seems to be its modular design for easy configuration by the users who use this template for their own applications.

Another different type of directory-level structural cloning was also detected in this system. It was observed that in some places, similar files were placed together in a common directory, as shown in Fig. 24. These six files populate the database for different entities and belong to the same FCSet, as detected by Clone Miner.

10 Coverage Analysis

The objective of the coverage analysis is to show that structural clones, as detected by Clone Miner, are bigger entities and less in number as compared to the simple clones, yet at the same time they capture most of the similarity present in the system. This would indicate that dealing with structural clones would be more meaningful and manageable by the user as compared to dealing with just simple clones to achieve any user objective of program understanding or reengineering. However, these numbers should be treated only as indicative, as different threshold values selected at different stages of the analysis will result in different numbers.

We performed two detailed case studies on systems that showed high levels of cloning, namely, CAP-WP (as discussed earlier in Section 7) and Linux device drivers (abbreviated to WP and LD, respectively). Relevant systems' statistics are shown in Table 4. The method-based analysis was only performed on WP system.

TABLE 4
Case Study Systems

	LD	WP
No. of files	4,130	1,919
LOC	3,182,873	98,329
No. of Directories	209	336
No. of Methods	--	8,581

TABLE 5
Simple Clone Sets

	LD	WP
No. of SCSets	14,287	1,979
No. of instances of all SCSets	35,793	16,328
Average no. of instances / SCSets	2.55	8.25
Average length of all instances	91 tokens	64 tokens

TABLE 6
Repeating Groups of Simple Clones

	LD	WP	LD	WP
	Level 2-B		Level 2-A	
No. of groups	5,036	6,049	2059	522
Groups as % of SCSets	35%	306%	14%	26%
No. of instances of all groups	16,583	44,866	5,493	2,220
Average no. of instances / group	3.3	7.4	2.7	4.2
SCSets covered by groups	7,579	1,533	7,122	756
% of SCSets covered by groups	53%	77%	50%	38%

TABLE 7
File Clone Sets (FCSets)

	LD		WP	
minCover	50%	90%	50%	90%
No. of FCSets	112	21	85	31
No. of Files covered by FCSets	254	57	929	96
% of files covered by FCSets	6%	1%	48%	5%
No. of SCSets covered by FCSets	982	96	274	167
% of SCSets covered by FCSets	7%	1%	14%	8%

Table 5 shows the statistics of the SCSets found in the two systems with Clone Miner's own simple clone detector. LD was analyzed with minimum simple clone size threshold of 50 tokens, while WP was analyzed with 30 tokens threshold setting.

Statistics about the level 2-A and 2-B structural clones in the two systems are presented in Table 6.

In case of LD, the number of level 2-B structural clones is only 35 percent of the total number of SCSets, yet covering 53 percent of those. Even better in terms of the coverage of simple clones by structural clones, level 2-A structural clones are only 14 percent of the total number of SCSets, yet cover 50 percent of them. Similarly in WP, the level 2-A structural clones give similar kind of advantage as is present in LD.

Table 7 gives the analysis of file clone sets (FCSets). Two different values were used for *minCover* to assess the

TABLE 8
Repeating Groups of File Clones

	LD	WP	LD	WP
	Level 6-B		Level 6-A	
No. of Groups	12	110	231	259
No. of FCSets covered by groups	12	46	149	65
% of FCSets covered by groups	10%	54%	91%	76%
No. of files covered by groups	31	804	335	221
% of files covered by groups	1%	42%	8%	12%
Max. no. of files in a group	7	366	27	40
Min. no. of files in a group	2	2	2	2
Avg. no. of files in a group	2	61	6	13
No. of directories having groups	23	167	63	94
% of directories having groups	11%	50%	30%	28%

intensity of cloning. In LD, the percentage of SCSets or files covered by the FCSets is not so significant, but still the actual number of files that are covered by FCSets is quite amazing. There are 57 files, grouped into 21 file clone sets that show almost 90 percent similarity with files in the same group. In WP, where we already expected that a high percentage of files would be covered by FCSet, 48 percent of the files were actually found to belong to the FCSets when *minCover* value of 50 percent is set.

Analysis of repeating groups of file clones across and within directories (level 6-a and level 6-B structural clones) with *minCover* value of 50 percent is given in Table 8. The results of level 6-B structural clones are interesting in the WP system; 42 percent of the files are clustered into only 110 groups of minimum 50 percent similarity of files within each group.

Analysis of method-level cloning was only done for the WP system. The method clone sets (MCSets) detected from the repeating groups of simple clones across different methods for two different *minCover* values are presented in Table 9.

Because of the inherent similarities present in the methods in WP due to its design, 20 percent of the methods (2,107 methods) are found to be grouped into only 228 method clone sets having a *minCover* value as high as 90 percent. As the number of these MCSets is only 11 percent of the number of SCSets, it is more convenient and meaningful to deal with MCSets as compared to SCSets.

Analysis of repeating groups of method clones across files (level 4-B structural clones) using two different *minCover* values is shown in Table 10. Although, the percentage of methods covered by these structural clones

TABLE 9
Method Clone sets (MCSets)

minCover	50%	90%
No. of MCSets	297	228
MCSets as % of SCSets	15%	12%
No. of Methods covered by MCSets	2107	1696
% of methods covered by MCSets	25%	20%
Min. No. of methods in an MCSet	2	2
Max. no. of methods in an MCSet	380	380
Avg. no. of methods in an MCSet	7	7

TABLE 10
Repeating Groups of Method Clones across Files

minCover	50%	90%
No. of groups	283	186
No. of MCSets covered by groups	153	107
% of MCSets covered by groups	51.5%	47%
Methods covered by groups	1,417	1,075
% of methods covered by groups	17%	13%
Min. no. of methods in a group	2	2
Max. no. of methods in a group	192	180
Avg. no. of methods in a group	39	46

is not very significant, yet the total number of such methods is still quite high.

11 Related Work

Clone detection tools produce an overwhelming volume of simple clones' data that is difficult to analyze in order to find useful clones. This problem prompted different solutions that are related to our idea of detecting structural clones.

Some clone detection approaches target large-granularity clones such as similar files, without specifying the details of the low-level similarities contained inside them. For example, in [15], the authors consider a whole webpage as a "clone" of another page if the two pages are similar beyond a given threshold, computed as the Levenshtein distance. Without the details of the low-level similarities in the large-granularity clones, it is not always straightforward to take remedial actions such as refactoring or creating generic representation, as these actions require a detailed analysis of low-level similarities. Moreover, Clone Miner goes a step ahead in clone analysis, by looking at the bigger similarity structures consisting of groups of such highly similar files.

In contrast, Gemini [49] determines the similarity between pairs of files based on file coverage by the common simple clones, as detected by CCFinder [28]. However, Gemini does not go as far as to identify explicitly the files as clones of each other but only provides a similarity value. Another limitation of these tools in terms of identifying file-level similarities is that only pairs of files are compared rather than finding groups of similar files, as found by Clone Miner.

In Clone Miner, not only do we identify complete sets of large-granularity clones, such as groups of similar files, methods, and directories, but we also provide all the low-level similarity details that are necessary for refactoring or creating generic representations to unify these similarities.

Rieger's idea of "clone class families" [46], where clone sets are grouped together based on their location, is the same as a level 2-B structural clone detected by Clone Miner. Kapser and Godfrey [29] have also explored the idea of linking simple clones with the system architecture.

The work of De Lucia et al. [14] involves detecting web-specific types of structural clones, where a clone consists of several webpages linked by hyperlinks. A graph-based pattern-matching algorithm is used for identifying this type of clones.

Marcus and Maletic [39] approach the detection of structural clones from a different perspective. This work defines "high-level concept clones" as manifestation of higher-level abstractions in the problem or solution domain, giving the example of the ADT *list* that has been duplicated in one form or another throughout a system. The clone detection method is based on examining source code text (comments and identifiers) to identify similar high-level concepts. An information retrieval approach is used to determine the semantic similarities in the source code. It is proposed to use these similarity measures to guide the simple clone detection process. They sum up their work as an attempt to show that domain concepts can be used to identify clones (in contrast to common approach of trying to identify domain concepts using clone analysis). While there are similarities in the goals of their work and ours (i.e., both approaches try to find the higher-level similarities), the promises made and the methods used are very much different and complementary. Structural clone detection is an attempt to move towards high-level similarity patterns, yet firmly rooted in patterns of concrete similarities at implementation level. A structural clone may indicate a cloned concept (in the requirements or design space). A "high-level concept clone" stems from a similarity in concepts. There is no emphasis on the structure of the clone found, although it may be a structural clone as well. There may be some overlap between similarities found by both methods, also there may be many "concepts" that are not captured in a "structure" (e.g., two List containers implemented in very different ways), and finally, there may be many structural clones that are not limited to a single concept (e.g., a structure made up of several interlinked concepts). When there is structural as well as conceptual similarity present between two program parts, but developers rename the identifiers in one part, a structural clone detector should detect such program parts as structural clones. The detection method in [39] fails in such cases. However, this seems to be a weakness of a particular detection method, rather than a flaw in the concept of high-level concept clones.

Clone detection techniques using Program Dependence Graphs (PDG) are described in [33] and [36]. In addition to the simple clones, these tools can also detect noncontiguous 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. While our technique detects structural clones with segments related to each other based only on their colocation, with or without information links, the PDG-based techniques relate them using the information links only. Moreover, the clustering mechanism in Clone Miner, to identify groups of highly similar methods, files, or directories based on their contained clones, is missing from these techniques.

Micropatterns [20] are implementation level patterns that are mechanically recognizable and can be expressed as a formal condition on the structure of a class. Some micropatterns may appear as structural clones, but given the nature of variability that is allowed in the actual implementation of a micropattern, they may not appear as code clones at all. Structural clones, on the other hand, are system-specific similarity patterns that may not necessarily reflect best programming practices, and hence, may not be described as micropatterns. However, the benefits provided by structural clone information, such as avoiding the risk of update anomalies, help in refactoring, or forming the generic representation of a system or a Product Line, cannot be realized by micropatterns. There is also a fundamental difference in searching for micropatterns and detecting structural clones. When looking for micropatterns, we already know precisely what we are looking for, but detection of structural clones is finding of unknown patterns. The same is the case with Pinot [47] that looks for known design patterns in source code.

PR-Miner [37] is another tool that discovers implicit programming rules using the frequent item set technique. Compared to structural clones found by Clone Miner, these programming rules are much smaller entities, usually confined to a couple of function calls within a function. The work by Ammons et al. [1] is also similar, finding the frequent interaction patterns of a piece of code with an API or an ADT, and representing it in the form of a state machine. These frequent interaction patterns may appear as a special type of structural clone, in which the dynamic relationship of cloned entities is considered. Similar to Clone Miner, this tool also helps in avoiding update anomalies, though only in the context of anomalies to the frequent interaction patterns.

There is also strong connection between clone detection and the work done previously on the design recovery and program understanding of large legacy systems for ease of maintenance and reuse [10], [11], [35]. Clones, especially structural clones of large granularity, 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, as discussed in the "Programmer's Apprentice" project [44], and *programming plans*, mentioned by Hartman [23] and Rich and Wills [45], represent commonly used program structures, which may appear as file-level structural clones within or across software systems (Product Line members). Software was searched for these plans (or clichés) to help in program understanding.

12 Conclusion and Future Work

In this paper, we emphasized the need to study code cloning at a higher level. We introduced the concept of structural clone as a repeating configuration of lower-level clones. We presented a technique for detecting structural clones. The process starts by finding simple clones (that is, similar code fragments). Increasingly higher-level similarities are then found incrementally using data mining technique of finding frequent closed item sets, and clustering. We implemented the structural clone detection technique in a tool called Clone Miner. While Clone Miner can also detect simple clones, its underlying structural clone detection technique can work with the output from any simple clone detector. We evaluated the performance of Clone Miner and assessed its usefulness by analyzing structural clones found in a number of commercial and public domain software systems. We believe our technique is both scalable and useful. Structural clone information leads to better program understanding, helps in different

maintenance related tasks, and points to potential reusable components across a Product Line. Structural clones are also candidates for unification with generic design solutions. After such unification, programs are easier to understand, modify, and reuse. In the future work, we plan to extend our technique for finding other, more complex types of similarities and to form a taxonomy of these structural clones. Experimentation with recovery of higher-level design similarities in various application domains and performing analytical studies to measure the precision and recall of the technique are also part of our future work.

Implementing good visualizations for higher-level similarities is currently underway [52]. Analysis of clones can also be much facilitated by querying the database of clones. We have already developed a mechanism of creating a relational database of structural clones' data and a query system to facilitate the user in filtering the desired information.

Currently, our detection and analysis of similarity patterns is based only on the physical location of clones. With more knowledge of the semantic associations between clones, we can better perform the system design recovery. Using tracing techniques to find associations between classes and methods, we can automate and build a clearer picture of the similarity in process flows within a system to further aid to the user in design recovery.

ACKNOWLEDGMENTS

The authors wish to thank Professor William F. Smyth (for all the help with the algorithms) and students Melvin Low Jen Ku (for CAP-WP design recovery case study), Zhang Yali (for computing statistics in Section 10), and Goh Kwan Kee and Chan Jun Liang (for case studies in Section 9). The authors are also thankful to the anonymous reviewers for their valuable comments and feedback. This work was supported by NUS research grant RP-252-000-239-112.

REFERENCES

[1] G. Ammons, R. Bodik, and J.R. Larus, "Mining Specifications," *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 4-16, 2002.

[2] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proc. Second Working Conf. Reverse Eng.,* pp. 86-95, 1995.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Partial Redesign of Java Software Systems Based on Clone Analysis," *Proc. Sixth Working Conf. Reverse Eng.,* pp. 326-336, 1999.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring," *Proc. Seventh Working Conf. Reverse Eng.,* pp. 98-107, 2000.

[5] H.A. Basit, D.C. Rajapakse, and S. Jarzabek, "Beyond Templates: A Study of Clones in the STL and Some General Implications," *Proc. 28th Int'l Conf. Software Eng.,* pp. 451-459, May 2005.

[6] H.A. Basit, S. Puglisi, W. Smyth, A. Turpin, and S. Jarzabek, "Efficient Token Based Clone Detection with Flexible Tokenization," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 513-516, Sept. 2007.

[7] H.A. Basit and S. Jarzabek, "Detecting Higher-Level Similarity Patterns in Programs," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 156-165, Sept. 2005.

[8] D. Batory, V. Singhai, M. Sirkin, and J. Thomas, "Scalable Software Libraries," *Proc. ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 191-199, Dec. 1993.

[9] I.D. Baxter, A. Yahin, L. Moura, M.S. Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 368-377, 1998.

[10] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *Computer,* vol. 22, no. 7, pp. 36-49, July 1989.

[11] E. Buss, R.D. Mori, W. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Muller, J.M.S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster, and K. Wong, "Investigating Reverse Engineering Technologies for the CAS Program Understanding Project," *IBM Systems J.,* vol. 33, no. 3, pp. 477-500, 1994.

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

[13] J.R. Cordy, "Comprehending Reality: Practical Barriers to Industrial Adoption of Software Maintenance Automation," *Proc. 11th IEEE Int'l Workshop Program Comprehension,* (keynote paper), pp. 196-206, 2003.

[14] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, "Reengineering Web Applications Based on Cloned Pattern Analysis," *Proc. 12th Int'l Workshop Program Comprehension,* pp. 132-141, 2004.

[15] A. De Lucia, G. Scanniello, and G. Tortora, "Identifying Clones in Dynamic Web Sites Using Similarity Thresholds," *Proc. Int'l Conf. Enterprise Information Systems,* pp. 391-396, 2004.

[16] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 109-118, 1999.

[17] M. Fowler, *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997.

[18] M. Fowler, *Refactoring—Improving the Design of Existing Code.* Addison-Wesley, 1999.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1997.

[20] J.Y. Gil and I. Maman, "Micro Patterns in Java Code," *Proc. 20th Object Oriented Programming Systems Languages and Applications,* pp. 97-116, 2005.

[21] G. Grahne and J. Zhu, "Efficiently Using Prefix-Trees in Mining Frequent Itemsets," *Proc. First IEEE ICDM Workshop Frequent Itemset Mining Implementations,* Nov. 2003.

[22] J. Han and M. Kamber, *Data Mining: Concepts and Techniques.* Morgan Kaufman Publishers, 2001.

[23] J. Hartman, "Technical Introduction to the First Workshop Artificial Intelligence and Automated Program Understanding," *Proc. Workshop Notes of the AAAI-92 Workshop Program: AI & Automated Program Understanding,* pp. 8-30, July 1992.

[24] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "ARIES: Refactoring Support Environment Based on Code Clone Analysis," *Proc. Eighth IASTED Int'l Conf. Software Eng. and Applications,* pp. 222-229, Nov. 2004.

[25] S. Jarzabek, *Effective Software Maintenance and Evolution: Reused-Based Approach.* CRC Press, Taylor and Francis, 2007.

[26] S. Jarzabek and S. Li, "Eliminating Redundancies with a 'Composition with Adaptation' Meta-Programming Technique," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 237-246, Sept. 2003.

[27] S. Jarzabek and S. Li, "Unifying Clones with a Generative Programming Technique: A Case Study," *J. Software Maintenance and Evolution: Research and Practice,* vol. 18, no. 4, pp. 267-292, July 2006.

[28] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 654-670, July 2002.

[29] C. Kapser and M.W. Godfrey, "Toward a Taxonomy of Clones in Source Code: A Case Study," *Proc. Int'l Workshop Evolution of Large Scale Industrial Software Architectures,* pp. 67-78, 2003.

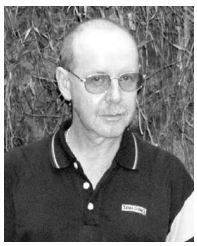
[30] C. Kapser and M.W. Godfrey, "Improved Tool Support for the Investigation of Duplication in Software," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 305-314, Sept. 2005.

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming,* pp. 220-242, 1997.

[32] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," *Proc. Int'l Symp. Empirical Software Eng.,* pp. 83-92, 2004.

[33] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Proc. Eighth Int'l Symp. Static Analysis,* pp. 40-56, 2001.
[34] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," *Proc. 13th Working Conf. Reverse Eng.,* pp. 253-262, 2006.
[35] W. Kozaczynski, J. Ning, and A. Engberts, "Program Concept Recognition and Transformation," *IEEE Trans. Software Eng.,* vol. 18, no. 12, pp. 1065-1075, Dec. 1992.
[36] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. Eighth Working Conf. Reverse Eng.,* pp. 301-309, Oct. 2001.
[37] Z. Li and Y. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code," *ACM SIGSOFT Software Eng. Notes,* vol. 30, no. 5, pp. 306-315, Sept. 2005.
[38] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Trans. Software Eng.,* vol. 32, no. 3, pp. 176-192, Mar. 2006.
[39] A. Marcus and J.I. Maletic, "Identification of High-Level Concept Clones in Source Code," *Proc. Int'l Conf. Automated Software Eng.,* pp. 107-114, 2001.
[40] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 244-254, 1996.
[41] D. Parnas, "Software Aging," *Proc. 16th Int'l Conf. Software Eng.,* pp. 279-287, 1994.
[42] U. Pettersson and S. Jarzabek, "Industrial Experience with Building a Web Portal Product Line Using a Lightweight, Reactive Approach," *Proc. European Software Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 326-335, Sept. 2005.
[43] D.C. Rajapakse and S. Jarzabek, "Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis," *Proc. Int'l Conf. Software Eng.,* May 2007.
[44] C. Rich and R.C. Waters, *The Programmer's Apprentice.* ACM Press, Addison-Wesley, 1990.
[45] C. Rich and L.M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software,* vol. 7, no. 1, pp. 82-89, Jan. 1990.
[46] M. Rieger, "Effective Clone Detection without Language Barriers," PhD thesis, Univ. of Bern, 2005.
[47] N. Shi and R.A. Olsson, "Reverse Engineering of Design Patterns from Java Source Code," *Proc. 21st IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 123-134, Sept. 2006.
[48] I. Sommerville, *Software Engineering,* fifth ed. Addison-Wesley, 1996.
[49] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance Support Environment Based on Code Clone Analysis," *Proc. Eighth IEEE Symp. Software Metrics,* pp. 67-76, 2002.
[50] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "On Detection of Gapped Code Clones Using Gap Locations," *Proc. IEEE Ninth Asia-Pasific Software Eng. Conf.,* pp. 327-336, 2002.
[51] A. Walenstein, A. Lakhotia, and R. Koschke, "The Second International Workshop Detection of Software Clones: Workshop Report," *SIGSOFT Software Eng. Notes,* vol. 29, no. 2, pp. 1-5, Mar. 2004.
[52] Y. Zhang, H. Basit, S. Jarzabek, D. Anh, and M. Low, "Query-Based Filtering and Graphical View Generation for Clone Analysis," *Proc. 24th IEEE Int'l Conf. Software Maintenance,* Sept. 2008.
[53] J. Yang and S. Jarzabek, "Applying a Generative Technique for Enhanced Reuse on J2EE Platform," *Proc. Fourth Int'l Conf. Generative Programming and Component Eng.,* pp. 237-255, 2005.

Hamid Abdul Basit received the BS degree from GIK Institute, Pakistan, in 2000 and the PhD degree from the National University of Singapore in 2006. He is an assistant professor at Lahore University of Management Sciences. He worked as a software engineer from 2000 to 2002. His research interests include software reengineering, design recovery, reusable software assets, and software maintenance. He is a member of the IEEE.



Stan Jarzabek received the PhD degree from Warsaw University. He is an associate professor in the Department of Computer Science at the National University of Singapore. Before joining the university, he spent 12 years in industry. He is interested in all aspects of software design and, in particular, design of adaptable software with the aid of generative techniques. He is a member of the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.