Solving Ccomplexity of Structural Clones by using Clone Mining Tool

Mr.S.B.Wakurdekar
Asst.Professor
BVDUCOE,Pune
sbwakurdekar@bvucoep.edu.in

Mrs.Y.C.Kulkarni
Asst.Professor
BVDUCOE,Pune
yckulkarni@bvucoep.edu.in

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 post detection 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, reusable software.*

1 INTRODUCTION

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 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.

The examples in Fig. 1 are abstracted from clones found in Project Collaboration portals developed in industry using ASP and JEE, and a PHP-based portal developed in our lab study . Structural clones are often induced by the application domain (analysis patterns, design technique (design patterns), or mental templates 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. 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 combinatory problem”.

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, small and large are bound to occur in such system variants. Software Product Line approach aims at reuse across families of similar systems. As we

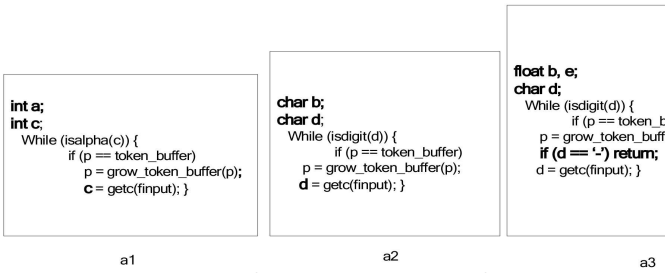


Fig. 1. A simple clone set formed by similar code fragments

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.

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. 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. Another way is to detect clones of larger granularity than code fragments. For example, some clone detectors can detect cloned files, while others target detecting purely conceptual similarities using information retrieval methods rather than detecting simple clones.

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

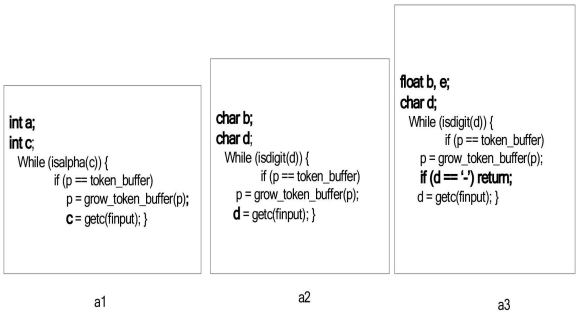


Fig. 1. A simple clone set formed by similar code fragments

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 .

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. 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. Another way is to detect clones of larger granularity than code fragments. For example, some clone detectors can detect cloned files while others target detecting purely conceptual similarities using information retrieval methods rather than detecting simple clones.

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 speci-

fying 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 filelevel 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 lowlevel 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 webspecific 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

Clone detection techniques using Program Dependence Graphs (PDG) are described in research papers. In addition to the simple clones, these tools can also detect non-contiguous clones, where the segments of a clone are connected by control and data dependency information links. Such clones also fall under the premise of structural clones. 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 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 that looks for known design patterns in source code.

PR-Miner 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 etal 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 . 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. Cliche´s, as discussed in the "Programmer's Apprentice" project , and programming plans, mentioned by Hartman and Rich and Wills , 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 cliche´s) to help in program understanding.

2. IMPLEMENTATION

CODE clones are similar program structures of considerable size and significant similarity. The following Five techniques are useful for similar programs structure recurring in software systems. Five modules have been proposed in the synopsis titled as,

1) Code Preprocessing
2) Token String with Clones
3) Pattern Mining
4) Clone Instances and Clone Regeneration
5) Identify Clone Behavior

Module 3: Pattern Mining

This pattern step is designed to handle set-typed data, where multiple values occur; thus, a naive approach is to discover repetitive patterns in the input. However, there can be many repetitive patterns discovered and a pattern can be embedded in another pattern, which makes the deduction of the template difficult. We detect every consecutive repetitive pattern (tandem repeat) and merge them (by deleting all occurrences except for the first one) from small length to large length.

To detect a repetitive pattern, the longest pattern length is predicated by the function compLvalueðList; tÞ (Line 1) in Fig.1, which computes the possible pattern length, called L value, at each node (for extension t) in List and returns the maximum L value for all nodes. For the *t*th extension, the possible pattern length for a node n at position p is the distance between p and the *t*th occurrence of n after p, or 0 otherwise. In other words, the tth extension deals with patterns that contain exactly t occurrences of a node. Starting from the smallest length i ¼ 1 (Line 2), the algorithm finds the start position of a pattern by the NextðList; i; stÞ function (Line 4) that looks for the first node in List that has L equal to i (i.e., the possible pattern length) beginning at st. If no such nodes exist, Next returns a negative value which will terminate the while loop at line 4. For each possible pattern starting at st with length i, we compare it with the next occurrence at j ¼ st þ i by function match, which returns true if the two strings are the same. The algorithm continues to find more matches of the pattern (j += i) until either the first mismatch (Line 7) or the end of the list has encountered, i.e., j þ i _ 1 _ jListj (line 6). If a pattern is detected (newRep > 0), the algorithm then modifies the list (modifyList at line 11) by deleting all occurrences of the pattern except for the first one, recomputes the possible pattern length for each node in the modified list (line 12), reinitializes the variables to be ready for a new repetitive pattern (line 5), and continues the comparisons for any further repetitive patterns in the list.

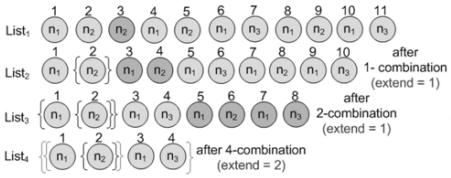
A pattern mining algorithm is shown below:

```
Procedure PatternMining(List, extent)
1. K = compLvalue(List, extend);
2. for (i = 1; i <= K; i++)
3.     st = 1;
4.     while((st = Next(List, i, st)) ≥ 0)
5.         newRep = 0;
6.         for (j = st + i; j + i − 1 ≤ |List|; j += i)
7.             if (!match(List[st..st + i − 1],
                       List[j..j + i − 1])) then break;
8.             newRep++;
9.         endfor
10.        if (newRep) then
11.            modifyList(List, st, newRep + 1, i);
13.            K = compLvalue(List, extend);
14.            st += i;
15.        else st++;
15.        endif
16.    endwhile
17. endfor
20. if (patternCanExtend(List)) then
21.    patternMining(List, extend + 1);
22. endif
```

A pattern may contain more than one occur-

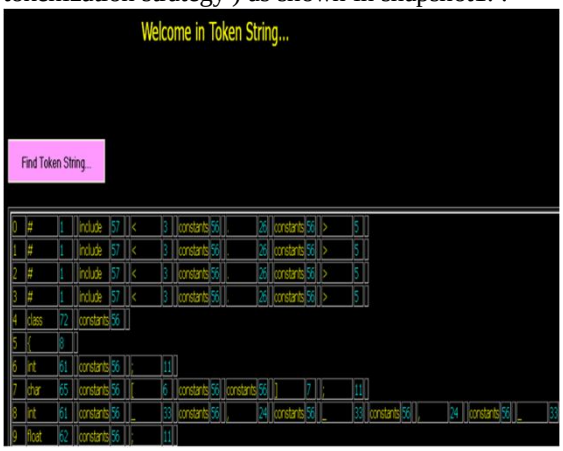
rence of a symbol; so the function recursively (with extension increased by 1) tries to detect such patterns (line 21). The termination condition is when there is no more nodes with more than one occurrence or the list cannot be extended by the function patternCanExtend, which is verified by checking if the length of List is greater than twice the length of the shortest repetitive pattern, i.e., jListj < 2ðlbÞðextend þ 1Þ, where lb is the minimum L value in the current list. The complexity of the algorithm is quadratic (Oðn2Þ; n ¼ jListj).

As an example, we apply the frequent pattern mining algorithm on List1 in Fig. 2 with extend ¼ 1. The L values for the 11 nodes are 3, 1, 2, 2, 4, 2, 4, 2, 0, 0, and 0, respectively. The patterns have length at most 4 (=K). Note that, the value of K may be changed after each modification of the list. First, it looks for 1- combination repetitive patterns by starting at the 2nd node (n2), which is the first node with L value 1. The algorithm starts at the 2nd (=st) node to compare every consecutive 1-combination of nodes,



and the comparison will continue until reaching the first mismatch at 4th node (n1). At this moment, the algorithm modifies the list by deleting the 3rd node (n2) to get List2. The new L values for the 10 nodes in List2 in order are 2, 2, 2, 4, 2, 4, 2, 0, 0, and 0 (the value of K is still 4). The algorithm looks for another repetitive pattern of length 1 in List2 starting from the 3rd node (st þ 1 ¼ 3), but finds no such nodes (the function Next returns a value -1). This will end the while loop (Line 4) and search for 2-combination on List2 from beginning (Lines 2 and 3). With L value equals 2 at the first node of List2, it compares the 2-combination patterns 1-2, 3-4 of List2 to detect a new repetitive pattern of length 2. The algorithm then deletes the second occurrence of the new detected pattern and outputs List3 with L values 2, 4, 2, 4, 2, 0, 0, and 0. The process goes on until all i-combinations, i _ K, have been tried. The algorithm then executes for the second time with extend=2 (Line 21). The new L values for List3 will be 4, 0, 4, 0, 0, 0, 0, and 0. Again, starting by 1-combination comparisons until the 4-combination, the algorithm detects a repetitive pattern of length 4 by comparing the two 4-combination 1-4 and 5-8, and finally gets List4 as a result. Finally, we shall add a virtual node for every pattern detected.

An input file is given is preprocessed(use tokenization strategy) as shown in snapshot1. :



Snapshot-1: Token string

After the source file is preprocessed, we apply pattern mining algorithm on it, which shown in snapshot2.

Snapshot-2: Repeated pattern counting

Module 4: Clone Instance

Code clones are similar program structures recurring in variant forms in software system(s). Detecting similar code fragments in software, so-called simple clones. Identification and subsequent unification of simple clones is beneficial in software maintenance. 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 typi-

cally reported by simple clone detection tools, a problem that is often dealt with post-detection visualization techniques. Detection of structural clones can help in understanding the design of the system for better maintenance and in re-engineering for reuse, among other uses.

To find clone instance following is the procedure:

1. Read the input source file(s).

2. Perform the code preprocessing.

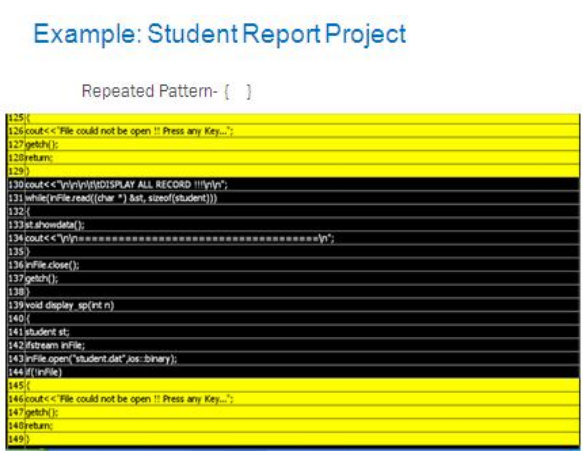
3. Apply the pattern mining algorithm to find clone instances.

A clone relation is an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code portions. A clone relation holds between two code portions if (and only if) they are the same sequences. For a given clone relation, a pair of code portions is called clone pair if the clone relation holds between the portions. An equivalence class of clone relation is called clone class. That is, a clone class is a maximal set of code portions in which a clone relation holds between any pair of code portions. Here, from clone instances regenerate the original code. As shown in snapshot3, we put following conditions to find code clones:

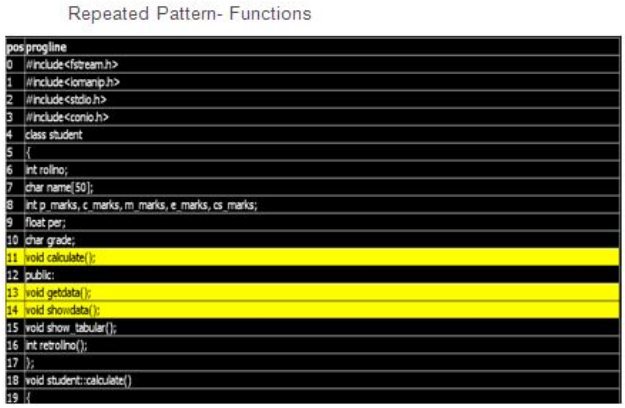
- Finding similar pattern classes.
- Finding similar pattern functions.
- Finding similar pattern structures.
- Finding similar pattern between { }
- View Program: When you click on view program button, it shows source code of program.
- View Repeated Pattern: It shows all similar patterns those are repeated in program.

As an example we give input file named StudentReport.project We get following results:

Snapshot3



Snapshot4



View Repeated Pattern

Experimental Result

Following table shows total number of tokens presents in project, Total number of clones presents in project and system resulted clones.

Program/Project Name	Total Number of tokens presents in program	Classes	Functions	Structures	{ }	View Repeated Patterns
Student Report	2565	No	Yes	No	Yes	Yes
Super Market	2551	No	Yes	No	Yes	Yes
Leap Year	137	No	No	No	Yes	Yes
Calculator	632	No	No	No	No	Yes
Inheritance	727	No	Yes	v	Yes	Yes

CONCLUSIONS 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 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. 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] Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E. The enhanced suffix array and its applications to genome analysis. In *Proc. Workshop on Algorithms in Bioinformatics*, in Lecture Notes in Computer Science, vol. 2452, Springer-Verlag, Berlin, 2002, pp. 449–463.

[2] Abouelhoda , M. I., Ohlebusch, E., and Kurtz, S. Optimal Exact Strring Matching Based on Suffix Arrays. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, pages .31-43. September 11-13, 2002.

[3] ANTLR website at http:// www.antlr.org

[4] Basit, H. A., Rajapakse, D. C., and Jarzabek, S. Beyond Templates: a Study of Clones in the STL and Some General Implications. In *Proceedings of the 28th Intl. Conf. on Software Engineering (ICSE'05)(to appear).* 2005. Draft available at http://xvcl.comp.nus.edu.sg/xvcl_cases.php

[5] Baker, B. S. On finding duplication and near-duplication in large software systems. In *Proc. 2nd Working Conference on Reverse Engineering.* 1995, pages 86-95.

[6] Baker, B. S. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal of Computing*, October 1997.

[7] Baxter, I., Yahin, A., Moura, L., and Anna, M. S. Clone detection using abstract syntax trees. In *Proc. Intl. Conference on Software Maintenance* (ICSM '98), pp. 368-377.

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

[9] Buss, E., Mori, R. D., Gentleman, W., Henshaw, J., Johnson, H., Kontogiannis, K., Merlo, E., Muller, H., Paul, J. M. S., Prakash, A., Stanley, M., Tilley, S., Troster, J., and Wong, K., "Investigating reverse engineering technologies for the CAS program understanding project"**,** *IBM Systems Journal*, 33(3):477-500, 1994.

[10] *Case Study: eliminating redundant codes in the Buffer library*. At XVCL Website, http://xvcl.comp.nus.edu.sg/xvcl/buffer/index.htm

[11] Church, K. W. and Helfman, J. I. Dotplot: A program for exploring self-similarity in million of lines of text and code. *Journal of Computational and Graphical Statistics*, June 1993, 2(2):153-174.

[12] Davey, N., Barson, P., Field, S., Frank, R., and Tansley, D. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3-4): 219-236, 1995.

[13] Ducasse, S, Rieger, M., and Demeyer, S. A language independent approach for detecting duplicated code. In *Proc*