18th IEEE International Conference on Program Comprehension

A Technique for Just-In-Time Clone Detection in Large Scale Systems

Liliane Barbour, Hao Yuan, Ying Zou
Dept. of Electrical and Computer Engineering
Queen's University
Kingston, Canada
{4lb3, 8hy14, ying.zou}@queensu.ca

Abstract— **Existing clone tracking tools have limited support for sharing clone information between developers in a large scale system. Developers are not notified when new clones are introduced by other developers or when existing clones are modified. We propose a client-server architecture that centrally detects and maintains clone information for an entire software system stored in a version control system. Clients retrieve a list of clones relevant to the code they are working on from the server. Whenever an update is committed to the version control system, the server detects and incrementally updates clone information. We propose techniques to improve the speed of the incremental clone detection. In order to reduce the number of comparisons required for clone detection, we select representative clones from the existing clone list. We build a string-based technique to compare the newly committed code with the representative clones and to update the clone list. In a case study, we show that our approach significantly reduces the clone detection time, while supporting clone detection across the entire software system.**

I. Introduction

In a large organization, many software projects are developed in parallel. To facilitate the flow of knowledge and expert experience, the code of all the software projects is centrally stored in a version control system, such as a concurrent versions system (CVS) [2] or subversion (SVN) [16]. The version control system can be used to track changes to the code and allow several developers to simultaneously update the source code of the same system. When making changes to the system, a developer checks out the relevant source code files from the version control system without checking out the entire code base. The developer can make changes to the code, and commit the modified code to the version control system.

In a large software organization, developers work on different projects in the version control system. Developers may introduce code clones, which have similar syntax or semantics due to copy-and-paste actions [3]. The existence of clones increases the maintenance efforts for large projects. For example, a developer can copy an existing version of a network driver and modify it to make a new driver for a different network card. Another developer may not be aware of code clones in the old driver. Subsequently, any bug fixes or quality improvements to the new driver code are not propagated to other clone instances in the existing drivers. This hinders the improvement of code quality and potentially leads to defects in the different software projects within an organization.

Techniques and tools are proposed and developed to keep developers aware of clones and allow them to easily track clones during the software development and maintenance process [4,5,9]. Existing clone tracking tools are focused on two major tasks: detecting new clones and tracking existing clones. To detect clones, existing clone detection tools either track the copy and paste behavior in a developer's integrated development environment (IDE) [4,9] or apply an existing clone detector to generate a list of clones [5]. These existing clone detection tools focus on detecting clones in a developer's local snapshot of the code. However, developers in large organizations might not have the complete code base stored on their local machine. Therefore, the list of clones produced from each developer's local snapshot is not complete, introducing software defects if the developer unknowingly modifies cloned code and fails to propagate changes to the clone instances in other projects.

To ease the detection and tracking of clones in a large code base, we propose techniques to improve the speed of incremental clone detection. However, it is not efficient to detect clones in newly committed code using the entire code base. Frequently cloned code is also more likely to be cloned again. Therefore, to improve the performance of the clone detection for incremental code changes, we can select representative clones from the existing clone list. Using a string-based approach, we compare the newly committed code with the representative clones, detecting new and removed clones. Using the clone detection results, we incrementally update the clone list.

The rest of the paper is organized as follows: Section 2 gives an overview of our approach for incremental clone detection. Section 3 presents a case study that evaluates the performance of our approach. Section 4 presents related work. Finally, Section 5 concludes the paper and discusses future work.

II. Overview of Our Approach

To communicate the most recent clone information with developers, we devise techniques for examining if newly committed code contains clones and for updating the list of existing clones to reflect the current status of the

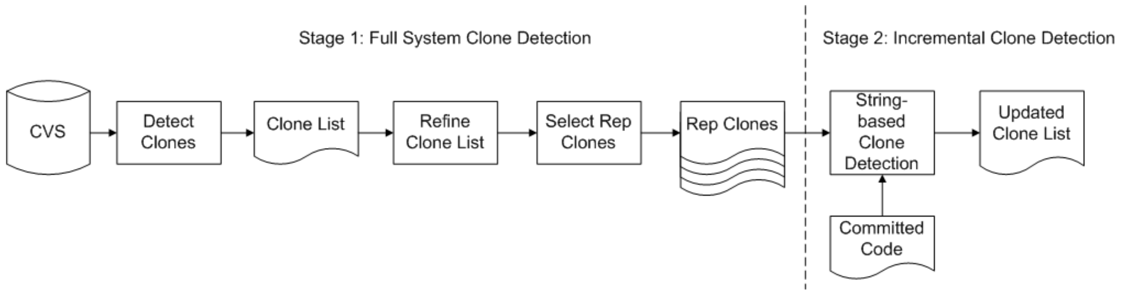

Figure 1. Detection of Clones

system. We propose a client-server architecture that centrally detects and maintains the clone information for an entire software system stored on a central source code repository. The clone detection and tracking are conducted on the server. The server also maintains and incrementally updates the clone list when a source code file is committed to the version control system by a developer.

The clients communicate with the server to access the clone list. The client automatically monitors the position of the cursor in the developer's IDE and determines which method the developer is modifying. The client then sends the method signature and the file path to the server to retrieve a list of clone classes for the current method. The client does not store a local copy of the clone list. Such centralized clone detection ensures that the clone list provides a complete view of clones in the entire code base. When the developer moves the cursor to a new method, the developer's IDE is updated with the most recent clone information for that method.

A. *Multi-staged clone detection*

To address the performance issues in detecting clones for incremental code changes, we divide the clone detection into two stages: full system clone detection and incremental clone detection. Figure 1 shows an overview of our approach. To conduct the full system clone detection, we maintain a checked-out copy of the most recent version of source code from the version control system. We periodically execute a full clone detection on the checked-out copy in order to add any new clone classes to the clone list. The full detection can be conducted at night or on weekends when development activities are not intensive and the responsiveness of the clone detection process is not a major concern.

In the incremental clone detection stage, we detect clones between the just-committed code and the existing clones to update the clone list.

B. *Collaboration of multiple clone detection tools*

We leverage the strength of existing clone detection tools and use them based on the characteristics of the code. To improve the precision of the detected clones, we mainly use SimScan [16], an abstract syntax tree (AST) based clone detection tool to create the initial clone list in the full system clone detection stage shown in Figure 1. SimScan can detect more complex clones in which lines of code have been added or removed. SimScan requires that code be syntactically correct. However, there is no guarantee that the code checked out from the version control system is syntactically correct. AST based clone detection tools may fail on certain source code files. For the failed files, we choose to use token-based clone detectors, such as CCFinder [11] or string-based clone detectors, such as Simian [15], to find exact copies of clones or clones in which the identifiers have been modified.

The candidate clones from clone detectors are refined to remove any false positive clones that are not correctly identified by the clone detectors. For example, a copied fragment may be a spurious clone or different clone classes may overlap.

C. *Fast just-in-time clone detection*

The speed of clone detection is proportional to the size of the code under analysis. Instead of extensively comparing the newly committed code with the entire code base, we select a representative clone from each clone class in the clone list. Each representative clone contains only the lines of cloned code, whether or not they are syntactically correct. Without re-scanning the entire code base, we only compare the newly committed code with the representative clone instances selected from each existing clone class. Using the clone detection results, we add any new clones to the clone list, and identify and remove obsolete clones.

Clone detectors, such as CCFinder and SimScan, must parse all code before performing clone detection. To avoid unnecessary overhead, such as tokenization and producing ASTs, we use a string searching algorithm to determine if the newly committed code is a clone by comparing it with the existing clone list. We select the Knuth-Morris-Pratt string searching algorithm [14] because it has a worst case runtime of O(n+m), where n and m are the lengths of the two strings under analysis.

A string searching algorithm can identify the clones of the exact copies. However, the algorithm cannot handle the situations where a developer copies a section of code, then changes the names of some of the identifiers. For example, a developer copies a printer driver to use as a basis for a new one and then renames some of the identifiers to reflect the new printer characteristics [12]. To extend the capability of detecting clones with more structurally equivalent code variations, we normalize all representative clones and the committed code to unify all

TABLE I. DESCRIPTIVE CHARACTERISTICS OF SUBJECT SYSTEMS USING SIMIAN TO CREATE REPRESENTATIVE CLONES

Subject System	Code Size (LOC)	Code Size (bytes)	Clone Classes (#)	Rep Clones (bytes)	Rep Clones (LOC)	Size Reduction (bytes)	Size Reduction (LOC)
netbeans-javadoc	19K	708,822	58	13,262	844	98.13%	95.56%
eclipse-ant	35K	1,503,201	52	7,809	539	99.48%	98.46%
eclipse-jdtcore	148K	7,238,510	938	179,357	12463	97.52%	91.58%
j2sdk1.4.0-javax-swing	204K	8,798,869	536	142,535	8776	98.38%	95.70%

identifiers and literals and remove whitespace (similar to the approach used by CCFinder [11]).

III. CASE STUDY

We conduct a case study to evaluate the effectiveness of our approach. The objectives of our case study are to:

1. Evaluate the performance of representative clones in reducing the amount of code for clone detection;
2. Compare the performance of our approach for incrementally detecting clones with other clone detection tools.

As our subject systems, we select four open source Java systems with sizes varying from 19 KLOC to 204 KLOC. Table I describes the characteristics of the four subject systems.

We compare the performance of our technique with two clone detectors: Simian [15], a string-based clone detection tool and SimScan [14], an AST-based clone detection tool.

A. Evalation of Code Reduction

Table I lists the number of clone classes in the initial clone list detected by Simian and the size of the representative clones in bytes and lines of code (LOC). We also calculate the reduction in the amount of code to analyze during clone detection. As shown in Table I, we can achieve on average 98.38% reduction in bytes and 95.33% reduction in LOC in all four applications. The speed of clone detection is proportional to size of the code under analysis. Thus, the representative clones contribute to an increase in incremental clone detection speed.

B. Evaluation of Performance of Incremental Clone Detection

In this experiment, we compare the performance of our technique for incremental clone detection with two existing clone detection tools. The description of the server test environment is shown in Table II.

Using three Java source files as test files from each system, we simulate incremental clone detection. We select three files to determine the effect of commit code size on incremental clone detection speed. The test files contain at least one clone to guarantee one positive clone detection result. We select the smallest, largest, and median Java source code files that contain clones, as determined by LOC. Our results are reported as an average of the three tests for each tool and system.

TABLE II. TESTING ENVIRONMENT IN THE SERVER

System	Ubuntu 9.04 Desktop 32-bit
Processor	Intel Core 2 Duo
Processor Speed	T8100 2.1Ghz
RAM	3 Gigabytes

We measure the time needed to complete incremental clone detection using our string-based approach, Simian, and SimScan. All three test files are analyzed using all three clone detection approaches. Time is measured from when each tool begins clone detection to completion. We do not include the time for updating the clone list.

For Simian and SimScan, we perform a full system clone detection three times, once for each Java source file. To prepare for tests using our string-based approach, we create representative clones by performing full system clone detection of each subject system using Simian. We then extract and normalize the cloned code segments to create the representative clones.

TABLE III. AVERAGE TIME FOR EACH ANALYSIS

	Representative Clones		**Full Subject System**	
Subject Systems	***String Based (seconds)***	***Simian (seconds)***	***SimScan (seconds)***	***Simian (seconds)***
eclipse-ant	0.048	0.141	15.565	0.249
netbeans-javadoc	0.041	0.162	172.082	0.649
eclipse-jdtcore	1.934	0.955	1619.396	5.192
j2sdk1.4.0-javax-swing	1.065	0.671	2976.976	1.760

Table III shows the average time of the three tests for each system using SimScan, Simian and our string-based approach. Simian and SimScan detect clones within the test files using the entire code base of each subject system. Our string-based approach detects clones using the representative clones. SimScan took significantly longer than Simian and our string-based approach. The speed of Simian and SimScan depends on the size of the subject system, thus the size of the test file did not affect the clone detection time. The string-based approach was sensitive to the size of the test file. The bigger the test file, the longer the detection time. The string-based approach was not affected by the size of the subject system; rather it is affected by the size of the representative clones. The test file is analyzed once for each representative clone, as compared to Simian and SimScan which also detect clones between all the source files in the subject system. Overall, our string-based approach is faster than Simian and

SimScan for all four subject systems when using representative clones.

We also tested the performance of Simian using the representative clones. Table III shows that Simian experienced a reduction in the time needed for incremental clone detection. For smaller systems, the difference in the time between Simian with representative clones and the string-based approach is small. In the case of the bigger system, Simian with representative clones was faster than the string-based approach. Simian is able to detect new clone classes between the representative clones and the committed code, so it is a better tool for just-in-time clone detection. Thus, we recommend using Simian in conjunction with representative clones for just-in-time clone detection.

IV. Related Works

Several clone tracking tools exist to support a developer within the IDE. CloneTracker [5] uses SimScan [16] as its underlying detection tool and tracks clones within the Eclipse IDE. CloneBoard [4] listens to the copy-and-paste behavior of developers to detect and track clones. Similarly, CREN [9] keeps track of all identifiers within copy-and-pasted code. By analyzing the update behavior across the clones, CREN notifies the developer if inconsistent updates to identifier names occur. Unlike our approach, these tools only track clones within the IDE, so the local clone list is incomplete if the developer does not have a local copy of the most recent software system.

Baker uses parameterized token suffix trees to detect clones within a software system [1]. Göde and Koschke extend this work and use generalized suffix trees to reflect incremental changes to the source code [8]. The clone detection is performed on the modified segments of the updated tree. Different from our approach, the entire source code must be loaded in memory, and changes are applied to the tree before clone detection can occur.

SHINOBI [13] uses a client-server architecture to detect and inform the developer about clones in a large scale system. Using the entire system code base, it builds a suffix array index to perform token-based clone detection. Unlike our approach, SHINOBI does not maintain a clone list, and clones are not reported as a clone class.

String matching techniques have been used for clone detection. Johnson [10] uses a variation of the Robin-Karp string searching algorithm to detect clones in source code that has been partitioned into substrings. Like our normalization approach, Johnson proposes pre-processing to remove whitespace and unify identifiers. Ducasse [6] proposes a string matching technique that hashes strings into buckets. The contents of each bucket are compared by creating a dot matrix. A dot appears in the matrix for each match between two strings. A diagonal in the matrix indicates a clone. Different from the aforementioned techniques, our approach uses representative clones to guide the search for clones within committed code instead of entire code base.

V. Conclusion

To increase the speed of clone detection, we proposed the selection of a representative clone from each clone class, reducing the amount of code to analyze during clone detection. We experimented with using a string-based approach in conjunction with the representative clones to detect clones in newly committed code. In the case study it was found that our approach was faster than two popular AST-based and string-based clone detection tools. A limitation of our string-based approach is that it only detects exact matches between the representative clones and committed code. In our future work, we will perform an empirical study to further evaluate the effectiveness of our approach.

References

[1] B.S. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems,” in *Proc. Working Conference on Reverse Engineering,* WCRE ‘95, July 1995, pp. 86-95.

[2] Concurrent Versions System, http://www.nongnu.org/cvs/

[3] J.R. Cordy, “Comprehending Reality – Practical Barriers to Industrial Adoption of Software Maintenance Automation,” in *Proc. IEEE 11th International Workshop on Program Comprehension,* IWPC ‘03, May 2003, pp. 196-206.

[4] M. de Wit, A. Zaidman, A. van Deursen, "Managing Code Clones Using Dynamic Change Tracking and Resolution," in *25th International Conference on Software Maintenance,* ICSM ‘09, 2009, pp.169-178,.

[5] E. Duala-Ekoko and M. P. Robillard, "Tracking Code Clones in Evolving Software," in *29th International Conference on Software Engineering,* ICSE 07, 2007, pp. 158-167.

[6] S. Ducasse, M. Rieger and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” in *Proc. IEEE International Conference on Software Maintenance,* ICSM ‘99, Aug. 1999, pp. 109-118.

[7] Eclipse, http://www.eclipse.org

[8] N. Göde, and R. Koschke, “Incremental Clone Detection,” in *13th European Conference on Software Maintenance and Reengineering,* CSMR ‘09, 2009, pp.219-228.

[9] P. Jablonski, and D. Hou, “CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE,” in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange,* Eclipse ‘07, 2007, pp.16-20.

[10] J.H. Johnson, “Substring Matching for Clone Detection and Change Tracking,” in *Proc. IEEE International Conference on Software Maintenance,* ICSM ‘94, Sept. 1994, pp. 120-126.

[11] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," in *IEEE Transactions on Software Engineering,* vol. 28; 28, pp. 654-670, 2002.

[12] C. Kapser and M. W. Godfrey, ""Cloning Considered Harmful" Considered Harmful," in *13th Working Conference on Reverse Engineering,* WCRE 06, Oct. 2006, pp. 19-28.

[13] S. Kawaguichi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, H. Iida, “SHINOBI: A Tool For Automatic Code Clone Detection in the IDE,” in *16th Working Conference on Reverse Engineering* WCRE ‘09, Oct. 2009, pp.313-314.

[14] D. Knuth, J. Morris, V. Pratt, “Fast pattern matching in strings,” *SIAM Journal on Computing*, vol 6;2, pp. 323-350, 1977.

[15] Simian, http//www.redhillconsulting.com.au/products/simian/

[16] SimScan, http://blue-edge.bg/simscan/

[17] Subversion, http://subversion.apache.org