Enhancing Generic Pipeline Model for Code Clone Detection using Divide and Conquer Approach

Al-Fahim Mubarak-Ali[1], Sharifah Syed-Mohamad[1], and Shahida Sulaiman[2]
[1]School of Computer Sciences, University Sains Malaysia, Malaysia
[2]Faculty of Computing, University Technology Malaysia, Malaysia

Abstract: *Code clone is known as identical copies of the same instances or fragments of source codes in software. Current code clone research focuses on the detection and analysis of code clones in order to help software developers identify code clones in source codes and reuse the source codes in order to decrease the maintenance cost. Many approaches such as textual based comparison approach, token based comparison and tree based comparison approach have been used to detect code clones. As software grows and becomes a legacy system, the complexity of these approaches in detecting code clones increases. Thus, this scenario makes it more difficult to detect code clones. Generic pipeline model is the most recent code clone detection that comprises five processes which are parsing process, pre-processing process, pooling process, comparing processes and filtering process to detect code clone. This research highlights the enhancement of the generic pipeline model using divide and conquer approach that involves concatenation process. The aim of this approach is to produce a better input for the generic pipeline model by processing smaller part of source code files before focusing on the large chunk of source codes in a single pipeline. We implement and apply the proposed approach with the support of a tool called Java Code Clone Detector (JCCD). The result obtained shows an improvement in the rate of code clone detection and overall runtime performance as compared to the existing generic pipeline model.*

Keywords: *Code clone detection, divide and conquer approach, generic pipeline model.*

Received August 31, 2012; accepted February 23, 2014; published online September 4, 2014

1. Introduction

Code clones are fragments of a source code that form clone pairs based on a given definition of similarity [2]. Three types of code clone include Type 1, Type 2 and Type 3 [15]. Type 1 is an exact copy without modifications with exception to white space and comments. Type 2 identifies identical copy syntactically. It only allows changes to variable, type or function identifiers. Type 3 is a copy code with further modifications. Modification involves statements that are changed, added or removed. Programmers use code clone to speed up development process. This occurs when a new requirement is not fully understood and a similar piece of code is present in the system that is not designed for reuse [8]. The programmers usually follow the low cost copy-paste technique instead of the costly redesigning approach, hence causing code clones. There are three scenarios contribute to cloning [11]. The first scenario is clones caused by poor design. These clones can be removed by replacing it with functions or through the refactoring process but technically there will be risks that might cause the clone removal process difficult. The second scenario is between long-lived clones and temporary clones [11]. Long-lived clones are clones that have existed in a program for a time while temporary clones only exist during the creation of the program. The third scenario is an essential clone that cannot be eliminated from the program [11]. This scenario occurs due to simplification of code as a prime reason for clone removal and restriction of programming language or design techniques. Furthermore, clones cannot be eliminated if the elimination of the clone affects quality of program.

Code clones cause unnecessary increase of maintenance cost. This happens due to the frequent changes carried out on clone instances [6]. If a code contains a bug, there is possibility that other code clone contains the same bug that needs to be fixed. Hence, this increases maintenance work not only due to the increase of the number of code clone but also bugs that exist in the code clone itself [17]. As software evolves rapidly, maintaining software becomes costly. Many tools exist to detect and remove code clones but as the software grows and becomes legacy, complexity of existing code clone detection techniques in detecting fully similar parts of the source code increases [2]. This makes code clones are difficult to be detected in software. Since, software maintenance is estimated more than half of a software development cost [9], therefore necessary measure needs to be taken in order to increase software maintenance productivity is by eliminating causes of code cloning. Therefore, this research highlights the needs of reducing clones in development stage.

This paper highlights the enhancement of generic pipeline model [3] using divide and conquers approach

that promotes concatenation process for code clone detection. Section 2 reviews current work related to code clone detection. Section 3 describes the proposed enhancement that has been made to the model. Section 4 explains the implementation flow while section 5 shows the experimental setup for testing purposes. Section 6 discusses the results and threats to the validity of the work and finally this paper is concluded in section 7.

2. Related Work

Various techniques and approaches such as textual comparison, token based comparison and comparison of Abstract Syntax Trees (AST) have been applied in detecting code clones [15].

Textual based comparison uses source code line to source code line comparison in order to find code clones in the same partition. It is among the earliest techniques for code clone detection. It has further evolved to the metric based comparison where the distance between metrics is used to compare the source codes. Tree based comparison [13] uses partition sub trees of the abstract syntax tree of a program based on a hash function and then compare sub trees in the same partition through a few techniques such as tree matching or dynamic programming [8]. Clone tracker [7] is an example of a tool that uses this approach with dynamic programming. DECKARD [13] uses this technique with tree matching technique. On the other hand, token-based comparison approach [14] uses token sequences, which contain lines of source code to detect code clone. It is a widely used approach for code clone detection. The tokens are differentiated uniquely using hash function. CCFinder [14] is a famous tool that uses this approach. This tool uses the combination of token-based comparison clone detecting technique with transformation rules.

Generic pipeline model [3] is a flexible yet extensible code clone detection model that contains all required steps in a code clone detection process. This model gives more freedom to the user to customize each process according to their needs. Java Code Clone Detection (JCCD) [4] is a code clone detection tool that implements this model.

Although, generic pipeline model [3] is an effective model for code clone detection due to its flexibility and extendibility, yet its inefficient way in handling source files as an input leads to a decrease in its performance. Furthermore, the generic pipeline model is highly dependent on number of pipelines to cater source file inputs, therefore causes load processing problem. The optimization on load processing has been done as non-automated process on CCFinder [14] but, it is only on a large source code prior to code clone detection process begins. Even though it is the first process done before clone detection, it is not an automated process that is embedded in the CCFinder [14] tool. Therefore, concatenation process that adopts the divide and conquers approach is proposed to improve the load processing of the source file as an input for the generic pipeline model.

3. Divide and Conquer Approach in Generic Pipeline Model

The generic pipeline model [3] uses a combination of processes to detect code clone. It is a flexible yet extensible code clone detection process that contains all required steps in a clone detection process. There are five processes involved in this model that are parsing process, pre-processing process, pooling process, comparing process and filtering process.

Parsing process is a process that transforms source code into source units. Source unit indicates the start and end of a fragment in a source file. A source unit might be represented in many forms such as a subtree of an AST, a line or a subgraph of a program dependence graph. The representation of a source unit is highly dependent on the approach that has been used. Furthermore, every approach requires different additional techniques like a line extractor, a lexer or a parser.

The source units are then normalized in the second process, which is the pre-processing process. Normalization turns the source units into a regular form and makes different source units to be more similar. It uses AST as the input and pre-processed AST as the output. It is implemented using several-cascaded processor. The normalization process changes a set of source units into regular form thus it makes different source units to be more similar.

The pre-processed AST source unit is then grouped into a set of groups according to defined characteristics based on the criteria set by users in the pooling process. The set of groups is called as a pool. The pool is then processed sequentially by comparing all contained source units recursively in the comparing process. This pool is then inputted to the filtering process in order to remove irrelevant clone candidate sets from the result set. This process is utilized in removing non-relevant candidate sets out of the result set.

The disadvantage of the generic pipeline model is that it is highly dependent on the use of single pipeline to cater all the process in the model [3]. This causes bottleneck in the pipeline due to the concurrent running of the process, which affects the runtime performance of the generic pipeline model. Another disadvantage of this model within its implemented tool that is JCCD allows pipeline manipulation for clone detection in large source files. Therefore, it causes the overhead cost such as the computer processors to increase [4]. Furthermore, each source file name and location has to be entered one by one into JCCD [4]. Hence, it increases the effort to enter a large amount of source

files into the tool. In order to, overcome the problem, we propose divide and conquer approach that includes a concatenation process as an enhancement to the current generic pipeline model [16]. It also improves code clone detection for similar code paths and solves the load processing problem.The common divide and conquer approach consists of three major steps [5].

- *Step* 1: Is breaking the source into sub problems that are themselves smaller instances of the same type of problem.
- *Step* 2: Is recursively solving these sub problems.
- *Step* 3: Is appropriately combining all the solved problems.

In order to solve the current problem, the first and second step is combined while the third step is replaced by a step named refactoring step. Below shows the pseudo code of the divide and conquer approach that is applied in the concatenation process.

Algorithm 1: *Concatenation process*

Source file, [*S*1, *S*2, *S*3, ..., *Sn*]
Sub source file:[*T*1, *T*2, *T*3, ..., *Tn*]
Function, [*F*1, *F*2, *F*3 ..., *Fn*]

*1. For each source file S*1,
2.Check function, Fn exist,
3. If function does not exist
*4. discard source file S*1,
5. Else
6. For each existing function, Fn
7. check function, Fn type
8. For each function, Fn is nested or loop
9. divide source file, S1 into sub source file [*T*1, *T*2, *T*3, ..., *Tn*]
10. For each function count >=2
11. divide source file, S1 into sub source file [*T*1, *T*2, *T*3, ..., *Tn*]
12. Repeat on other source file, [*S*2, *S*3, ..., *Sn*]
13. Refactor all the sub source files, [*T*1, *T*2, *T*3, ...*Tn*]

The combined first and second step is applied by breaking a source file into sub source files based on singular and nested functions. This process is recursively done on all the functions that exist in the source file. The combined step is then continued on other source files that exist in a program until all the source files have gone through this combined step. During this combined steps, empty source files and source files that do not have any function existence are removed from the Java application.

The third step is the refactoring step. This step is to refactor all the functions that exist in the sub source files. This is to preserve the structure of the functions that exist in the sub source file.

4. The Implementation

There are three main components involved in the concatenation process. Figure 1 illustrates the components involved together with the divide and conquer and approach that is applied as part of the concatenation process.

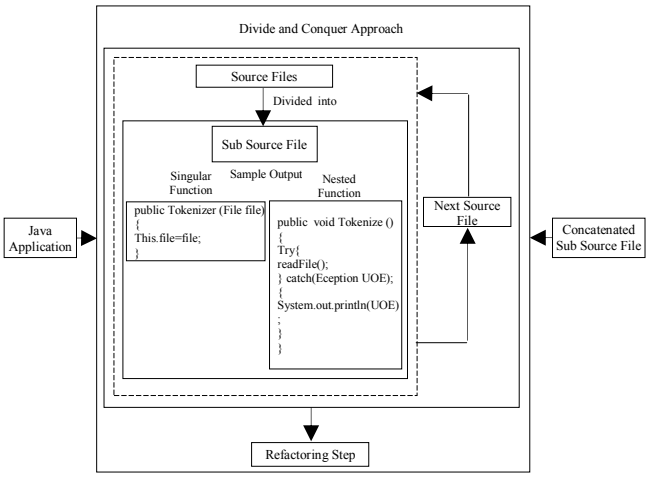

Figure 1. Divide and conquer approach in the concatenation process.

The first component is the input. The input currently limits to Java source codes since, the generic pipeline only supports this language. As for testing the enhanced generic pipeline model, this study uses Java applications of different amount of lines of codes and different amount of source files.

The second component is the concatenation process that speeds up the load processing by focusing on the smaller part of source code files before working on the large chunk of source code in a single pipeline. This concatenation process partially adopts parallel concatenation approach. Source codes in a source file are segmented iteratively based on function type, which is singular and nested type functions. This process is done until all the functions are segmented out and represented in the form of sub source files. The function definition of the concatenation process is shown below:

- F: Be the set of all source code files.
- $G=F\times N\times N\times N\times N\times C$ be the set of all sub source files.
- C: Be the set of concatenation sub source file.
- $P(F)$: Is the power set of F.

Therefore, the concatenation step is defined by a function of:

$$concat: P(F) \rightarrow P(G)$$

The third and final component is the output. The output of the concatenation process is the concatenated sub source files. These concatenated sub source files are stored in a folder. It serves as an input for the next process in the generic pipeline model, which is the parsing process. As mentioned in the previous section, parsing process is the first process in the generic pipeline model, which transforms source code into source units. The concatenated sub source files serves as a better input for the generic pipeline model.

There are three main modules used in the implementation of the concatenation process that are: readLine() function module, extractFile() function module and concatFile() function module. Figure 2

illustrates the implementation flow of the concatenation process.

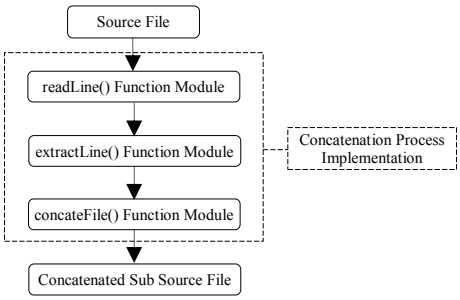

Figure 2. Concatenation process implementation flow.

The first module of the implementation is readLine() function module. Figure 3 shows the flow of readLine() function module. The purpose of this function module is to read each line from the source file and store into the array. The process starts with reading source file. This recursive process removes empty source files and irrelevant files and keeps source files that only contain Java source codes in a Java application. The process continues with the reading of source code line in the source file. This is a recursive process to make sure each line of source codes has been read and empty source code line has been removed. Each line in the source file is stored in an array separately. The set of arrays is then used in the next module function for function extraction. Figure 5 illustrates readLine() function module flow that is applied in the concatenation process.

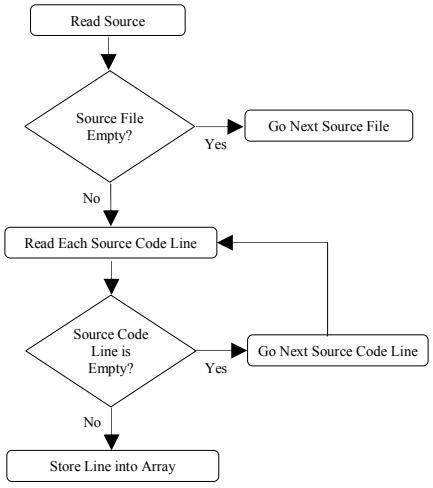

Figure 3. ReadLine() function module flow.

The second module of the implementation is extractFile() function module. extractFile() function module is the first module in the concatenation process implementation. The purpose of this function is to extract the functions from the original source file and store it into sub source file before it is being concatenated. Since, the extraction is done on functions, there are few conditions that need to be considered in order to detect the functions. These conditions are to distinguish between functions and basic declarations that exist in the source file.

The conditions are:

1. Backslash and front slash (/, \): The backslash and front slash are used for commentary purposes. This becomes an issue if there are functions highlighted in the comments.
2. Curly brackets ({, {) and semicolon (;): Curly brackets and semicolons are used to open and close the function. The issue arises for curly brackets that are not used or commented. The detected functions are stored in a newly created source file and are then concatenated. The sub source files are then used as input for the other process in the generic pipeline model.

The third module is the concatFile() function module. The purpose of this function module is to reformat and preserve each code structure in the sub source file. This is to make sure that all the criteria for a function exist and the structures of the functions are in proper.

5. Experimental Results

Three open source applications that are JHotDraw 7.0.6 [12], ANTLR 4 [1] and SableCC 3.2 [18] provide the data set to evaluate the enhanced generic pipeline model. These Java applications have proven to have occurrences of clones in their source codes [10]. The experiment used a workstation with the specification of 1.73GHz quad core CPU, 4GB of memory with Windows 7 as its operating system. JCCD [4] supports the implementation of the proposed approach and its testing. It is a tool for code clone detection that uses the generic pipeline model in detecting similar code parts. Each process of the model is implemented separately and integrated together in the tool. This research enhances the tool by adding and integrating the concatenation process, as the first process in the tool.

5.1. Code Clone Detection

Table 1 shows the amount of code clones that have been detected in JHotDraw 7.0.6, SableCC 3.2 and ANTLR 4 using both generic pipeline and enhanced generic pipeline model.

Table 1. Code clone detection results.

Application	Clone Detected using Generic Pipeline Model	Clone Detected using Enhanced Generic Pipeline Model
JHotDraw 7.0.6	2322	2336
SableCC 3.2	2072	2084
ANTLR 4	326	330

There were 2322 clones detected in JHotDraw 7.0.6 using the generic pipeline model but the number of clones detected increased 0.6% to 2336 using the enhanced generic pipeline model. As for SableCC 3.2, there were 2072 clones detected using the generic pipeline model but, the number of clones detected increased 0.57% to 2084 using the enhanced generic pipeline model. There were 326 clones detected in ANTLR 4 using the generic pipeline model but the number of clones detected increased 1.2% to 330 using

the enhanced generic pipeline model. Based on the comparison done, it shows the detection rate of code clone for all the applications increased. Therefore, the enhanced generic pipeline managed to detect more code clones as compared to the generic pipeline model.

5.2. Runtime Performance

1. Parsing Process: Figure 4 shows the runtime of the parsing process using both generic pipeline and enhanced generic pipeline model.

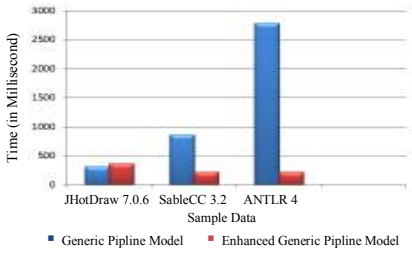

Figure 4. Parsing process runtime performance.

The runtime of the parsing process is 309.82 millisecond in JHotDraw 7.0.6 by using the generic pipeline model but the runtime increased 16% to 361.69 millisecond using the enhanced generic pipeline model. As for SableCC 3.2, runtime of the parsing process is 859.02 millisecond by using the generic pipeline model but the runtime decreased 73.9% to 224.26 millisecond using the enhanced generic pipeline model. The runtime detected in ANTLR 4 is 2785.90 millisecond by using the generic pipeline model but the runtime decreased 92.1% to 218.86 millisecond using the enhanced generic pipeline model.

Based on the comparison done, it shows that the runtime of the parsing process decreased for SableCC 3.2 and ANTLR 4 but increased for JHotDraw 7.0.6. The increase runtime of the parsing process in JHotDraw 7.0.6 is might be influenced by the length of code in a function for detecting code clone. Since, parsing is the process of producing source units by determining the start and end point, it takes more time for functions that has lengthy source code lines to be determined as source units. Therefore, the functions in JHotDraw 7.0.6 might have contained lengthy source codes thus causing the runtime to increase.

2. Pre-Processing Process: Figure 5 shows the runtime of the pre-processing process using both generic pipeline and enhanced generic pipeline model.

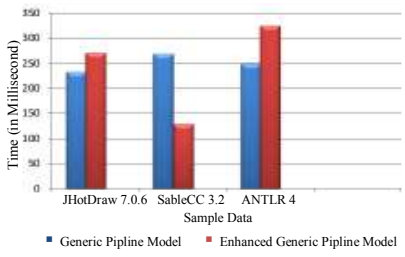

Figure 5. Pre-processing process runtime performance.

The runtime of the pre-processing process is 231.84 millisecond in JHotDraw 7.0.6 by using the generic pipeline model but the runtime increased 16.3% to 269.68 millisecond using the enhanced generic pipeline model. As for SableCC 3.2, runtime of the pre-processing process is 268.50 millisecond using the generic pipeline model but the runtime decreased 52.3% to 128.05 millisecond using the enhanced generic pipeline model. The runtime detected in ANTLR 4 is 247.76 millisecond using the generic pipeline model but the runtime increased 30.8% to 324.15 millisecond using the enhanced generic pipeline model.

Based on the comparison done, it shows that the runtime of the pre-processing process decreased for SableCC 3.2 but increased for JHotDraw 7.0.6 and ANTLR 4. The increase runtime of the pre-processing process in JHotDraw 7.0.6 and ANTLR 4 is might be influenced by the amount of white spaces that exists in the source units and also the process of adding additional notations to the source units. Since, pre-processing is the process of normalizing and adding additional notations of source units, it takes more time for functions that has a lot of whitespaces and the use of additional annotations in normalizing the source units. Therefore, the source units in JHotDraw in 7.0.6 and ANTLR 4 might have contained a lot of whitespaces in it thus causing the runtime to increase.

3. Pooling Process: Figure 6 shows the runtime of the pooling process using both generic pipeline and enhanced generic pipeline model.

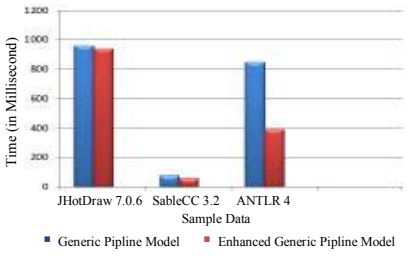

Figure 6. Pooling process runtime performance.

The runtime of the pooling process is 960.98 millisecond in JHotDraw 7.0.6 by using the generic pipeline model but the runtime decreased 2.4% to 937.69 millisecond using the enhanced generic pipeline model. As for SableCC 3.2, runtime of the pooling process is 78.43 millisecond by using the generic pipeline model but the runtime decreased 30% to 58.85 millisecond using the enhanced generic pipeline model. The runtime detected in ANTLR 4 is 847.70 millisecond by using the generic pipeline model but the runtime decreased 53.8% to 391.62 millisecond using the enhanced generic pipeline model.

Based on the comparison done, it shows that the runtime of the pooling process decreased for all the

sample data. Therefore, the enhanced generic pipeline was able to reduce the runtime of the pooling process as compared to the generic pipeline model.

4. Comparing Process: Figure 7 shows the runtime of the comparing process using both generic pipeline and enhanced generic pipeline model.

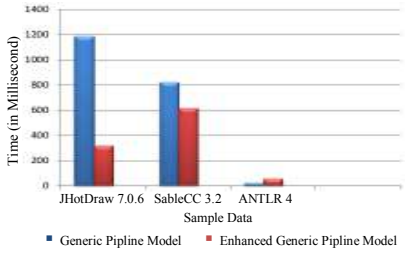

Figure 7. Comparing process runtime performance.

The runtime of the comparing process is 1181.25 millisecond in JHotDraw 7.0.6 by using the generic pipeline model but the runtime decreased 73.5% to 314.20 millisecond using the enhanced generic pipeline model. As for SableCC 3.2, runtime of the comparing process is 817.17 millisecond by using the generic pipeline model but the run time decreased 25.1% to 611.67 millisecond using the enhanced generic pipeline model. The runtime detected in ANTLR 4 is 27.02 millisecond by using the generic pipeline model but the runtime increased 81.4% to 55.02 millisecond using the enhanced generic pipeline model.

Based on the comparison done, it shows that the runtime of the comparing process decreased for JHotDraw 7.0.6 and SableCC 3.2 but increased for ANTLR 4. The increase in runtime of the comparing process in ANTLR 4 is might be influenced by the amount of pools exist in ANTLR 4. Since comparing is a recursive process of comparing pools to form similarity group, it takes more time for applications that has a lot of pools. Therefore, there might have been a lot of pools in ANTLR 4 thus causing the runtime to increase.

5. Filtering Process: Figure 8 shows the runtime of the filtering process using both generic pipeline and enhanced generic pipeline model.

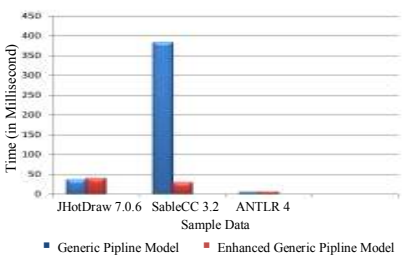

Figure 8. Filtering process runtime performance.

The runtime of the filtering process is 36.30 milliseconds in JHotDraw 7.0.6 by using the generic pipeline model but, the runtime increased 11% to 40.77 millisecond by using the enhanced generic pipeline model. As for SableCC 3.2, runtime of the filtering process is 383.03 millisecond by using the generic pipeline model but the runtime decreased 92.2% to 29.97 millisecond using the enhanced generic pipeline model. The runtime detected in ANTLR 4 by using the generic pipeline model and the enhanced generic pipeline model are the same which is 0.01 milliseconds.

Based on the comparison done, it shows that the runtime of the filtering process increased for JHotDraw 7.0.6, decreased for SableCC 3.2 and stayed the same for ANTLR 4. The increase in runtime of filtering process in JHotDraw 7.0.6 is might be influenced by the amount of irrelevant clones in JHotDraw 7.0.6. Since, filtering is a process of filtering of the similarity group in removing irrelevant code clone candidates, it takes more time for applications that has a lot of irrelevant code clones to be removed. Therefore, there might have been a lot of irrelevant code clones in JHotDraw 7.0.6 thus makes the runtime to increase. As for ANTLR 4, the unchanged runtime might be influenced by the very minimal occurrence of irrelevant code clones.

6. Overall Runtime Process: Figure 9 shows the runtime of the filtering process using both generic pipeline and enhanced generic pipeline model.

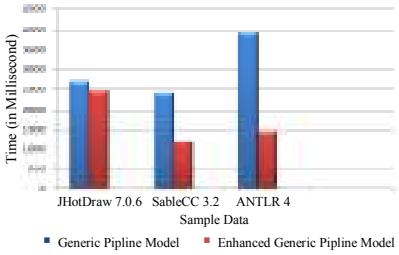

Figure 9. Overall runtime performance.

The overall runtime performance in JHotDraw 7.0.6 using the generic pipeline model is 2720.19 millisecond but the runtime decreased 10.1**%** to 2446.03 millisecond by using the enhanced generic pipeline model. As for SableCC 3.2, overall runtime performance is 2406.15 millisecond by using the generic pipeline model but the overall runtime decreased 49.9**%** to 1203.8 millisecond using the enhanced generic pipeline model. The overall runtime detected in ANTLR 4 by using the generic pipeline model is 3902.57 millisecond but decreased 62.9% to 1444.66 millisecond by using the enhanced generic pipeline model.

Based on the comparison done, it shows that the overall runtime decreased for all the sample data. Therefore, this shows that the enhanced generic pipeline model was able to reduce the overall runtime as compared to the generic pipeline model.

6. Discussion

As shown from the results, overall process time for JHotDraw 7.0.6, SableCC 3.2 and ANTLR 4 using the

enhanced generic pipeline model is lesser compared to the generic pipeline model. This shows that the proposed enhancement using divide and conquer approach in concatenation process managed to increase the performance of the previous model by reducing the process time of the model. In addition, the enhanced generic pipeline model was also able to detect the code clones for the singular and nested type of functions. Although, the results show improvement, yet there are issues that might cause threats to the validity of the results. The issues are sample selection, sample size, code structures, and hardware specification.

1. Sample Selection: Sample data used for the testing are open source applications since there is no standard sample data available for code clone detection. Furthermore, the amount of code clone and function existence are unknown in the open source applications causing the difficulty in knowing the total of code clones and functions that exist in the applications.
2. Sample Size: Sample size refers to the amount of applications used as sample data. The sample size used for the evaluation is three open source applications with different amount of Line Of Codes (LOC) and source files. The results might vary with more sample data with a bigger amount of LOC and source files.
3. Code Structures: The code structure varies for each open source application due to various reasons such as coding convention, system architecture and coding styles used by the programmers. Therefore, the code clone structure is also affected due to the variant in code structures.
4. Hardware Specification: The current workstation specification used for experiment enables the process that is limited to 400 source files or 60000 LOC. A higher memory and CPU workstation may give a better overall runtime performance.

7. Conclusions

Many techniques and tools for detecting and removing clone detection have been described in the literature. Generic pipeline model which is implemented in JCCD is an approach that contains the combination of five processes to detect code clones. However, the dependency of the processes in a single pipeline causes bottleneck problem in the generic pipeline model.

In this paper, a divide and conquer approach that includes concatenation process is proposed as an enhancement to the existing generic pipeline model in order to improve the load processing of the generic pipeline model. The output of the proposed enhancement that is the concatenated sub source file serves as a better input as compared to a normal source file. The experiment shows that the enhancement of the generic pipeline model for code clone detection is able to detect similar parts of code clones and also increase the performance of code clone detection as a whole.

References

[1] ANTLR Parser Generator., available at: http://www.antlr.org/, last visited 2012.

[2] Bellon S., Koschke R., Antoniol G., Krinke J., and Merlo E., "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577-591, 2007.

[3] Biegel B. and Diehl S., "Highly Configurable and Extensible Code Clone Detection," *in Proceedings of the 17th Working Conference on Reverse Engineering*, Massachusetts, USA, pp. 237-241, 2010.

[4] Biegel B. and Diehl S., "JCCD: A Flexible and Extensible API for Implementing Custom Code Clone Detectors," *in Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, Antwerp, Belgium, pp. 167-168, 2010.

[5] Dasgupta S., Papadimitriou C., and Vazirani U., *Algorithms*, McGraw Hill, New York, USA, 2006.

[6] Deissenboeck F., Hummel B., Juergens E., Pfaehler M., and Schaetz B., "Model Clone Detection in Practice," *in Proceedings of the 4th International Workshop on Software Clones*, Cape Town, South Africa, pp. 57-64, 2010.

[7] Duala-Ekoko E. and Robillard M., "Tracking Code Clones in Evolving Software," *in Proceedings of the 29th International Conference on Software Engineering*, Minnesota, USA, pp. 158-167, 2007.

[8] Hou D., Jacob F., and Jablonski P., "Exploring the Design Space of Proactive Tool Support for Copy-and-Paste Programming," *in Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, Ontario, Canada, pp. 188-202, 2009.

[9] Ibrahim S., Idris N., Munro M., and Deraman A., "Integrating Software Traceability for Change Impact Analysis," *the International Arab Journal of Information Technology*, vol. 2, no. 4, pp. 301-308, 2005.

[10] Ishio T., Date H., Miyake T., and Inoue K., "Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs," *in Proceedings of the 15th Working Conference on Reverse Engineering*, Antwerp, Belgium, pp. 123-132, 2008.

[11] Jarzabek S. and Xue Y., "Are Clones Harmful for Maintenance?" *in Proceedings of the 4th International Workshop on Software Clones*, Cape Town, South Africa, pp. 73-74, 2010.

[12] JHotDraw., available at: http://www.randelshofer.ch/oop/jhotdraw/, last visited 2012.

[13] Jiang L., Misherghi G., Su Z., and Glondu S., "DECKARD: Scalable and Accurate Tree-based

Detection of Code Clones," *in Proceedings of the 29th International Conference on Software Engineering*, Minnesota, USA, pp. 96-105, 2007.
[14] Kamiya T., Kusumoto S., and Inoue K., "CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, 2002.
[15] Koschke R., Falke R., and Frenzel P., "Clone Detection using Abstract Syntax Suffix Trees," *in Proceedings of the 13th Working Conference in Reverse Engineering*, Benevento, Italy, pp. 253-262, 2006.
[16] Mubarak-Ali A., Syed-Mohamed S., and Sulaiman S., "An Enhanced Generic Pipeline Model for Code Clone Detection," *in Proceedings of the 5th Malaysian Conference in Software Engineering*, Johor Bahru, Malaysia, pp. 434-438, 2011.
[17] Roy C. and Cordy J., "A Survey on Software Clone Detection Research," *Technical Report*, Queen's University, 2007.
[18] SableCC., available at: http://sablecc.org/, last visited 2012.

Al-Fahim Mubarak-Ali received his BS degree of computer science (software engineering) from University Malaysia Pahang, Malaysia in 2009 and MS degree of science (computer science) from University Sains Malaysia, Malaysia in 2012. Currently, he is pursuing his PhD in the area of software engineering in University Teknologi Malaysia, Malaysia.

Sharifah Syed-Mohamad is a senior lecturer of the School of Computer Sciences, University Sains Malaysia. She received her PhD degree in software engineering from the University of Technology, Australia in 2012. Her research interests include software reliability, software testing, software maintenance and agile development.

Shahida Sulaiman is an associate professor of the Faculty of Computing, University Teknologi Malaysia. She holds a PhD degree in computer science and Ms degree in computer science (software engineering in real time systems). Her expertise includes software design, software maintenance, software visualisation and documentation and knowledge management.