UNIVERSITI TEKNOLOGI MARA

FACILITATING NOVICES' PROGRAM COMPREHENSION IN PROGRAM SLICING VIA A KNOWLEDGE-BASED AND PROGRAM SLICING TOOL

Mohd Zanes Bin Sahid
2011895944

Dissertation submitted in partial fulfillment of the requirement
for the degree of

Master of Science (Computer Science)

Faculty of Computer & Mathematical Sciences

January 2013

ABSTRACT

Most novice programmers cannot comprehend program code effectively due to lack of knowledge, skill and domain experience. Their program comprehension capabilities are fragile, and performed at the lower syntax level of program code only, whereas experts have the capability to comprehend program code effectively at the higher semantic level. This is primarily due to two major factors – the experts' ability to abstract code effectively based on their vast programming and problem domain knowledge, and their application of program slicing technique during program comprehension. Therefore, a new programming pedagogy semi-automated program comprehension tool called Knowledge-Based Slicer (KBS) that utilizes both knowledge-based and program slicing is designed and developed to support and improve program comprehension of novice programmers. The tool is developed based on adaptation and integration of two open-source tools; Simian, a program code similarity analyzer, and Indus-Kaveri, a static program slicing tool. KBS integrates them on top of a knowledge-based, and is deployed as a new Eclipse's plugin with simplified user interfaces and new features tailored mainly for novice programmers. KBS consists of two components, the KBS Analyzer and KBS Slicer. The knowledge-based in the KBS Analyzer is developed in the form of Basic Program Plans that covers three basic algorithms, which are total, maximum and average. To test the effectiveness of the KBS, four phases of testing have been performed. The first, second and third phases testing were performed on the individual component of KBS against 30 sample program codes and 54 randomly selected actual novices' program codes. In the final fourth phase integrated testing, program codes are firstly sliced by manually choosing the last computation result as the slicing criteria. This is followed and compared with the slices based on the criteria automatically suggested by KBS Analyzer. The precision of all matching are more than 0.7. Thus suggest that the KBS is able to assist novices in program comprehension by facilitating the selection of slicing criteria. The three main contributions of this research are the program comprehension tool for novices in applying program slicing with facilitated selection of slicing criteria, the first known demonstration of practical viability of integrating program slicing and knowledge-based technique for novices' program comprehension, and local experimental data on knowledge-based cum program slicing program comprehension tool. The future works of this project include the expansion of the Program Plans to include more computing algorithms, and actual implementation of KBS in Java programming courses.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

Program comprehension is an activity where a person will read a set of program codes and understand what the meaning or purpose of the program (Rugaber, 1995). Program comprehension is part of software engineering sub-activity, where it may be employed by a person during enhancement, debugging and re-engineering (Rugaber, 2000).

During enhancement, the program comprehension may be used to help the programmer to gain knowledge on the existing program codes before he/she is able to write more program codes (Mayrhauser & Vans, 1997b). During debugging, a programmer or software maintainer must have understood the program codes very well before he/she can locate the bug and do the fixing (Mayrhauser & Vans, 1997a). Whereas in re-engineering, one surely will have to gain a complete understanding of the program codes before he/she can extract out the software design and system specification from the program (Chikofsky & Cross, 1990).

The activity of program comprehension is known to be hard, especially to someone who have little knowledge and experience in programming. This is due

to the characteristics of program codes that act as a bridge between real world application and programming language (Rugaber, 1995). Similarly, Brooks (1987) had mentioned that the underlying properties of a software system has made the program codes to be complex, and this has made the program comprehension a tough task.

It has been realized that in real life, programmers spent more time reading and understanding program codes rather than writing new codes (Fjeldstad & Hamlen, 1983). As per mentioned previously, to perform program comprehension is not an easy task. This means that more focus should be put to enrich programmers' skill and knowledge in program code reading and understanding, rather than on writing. Therefore, a lot of research has been focused to study the technique and strategy on program code reading and understanding.

Over the past few decades, a number of research findings have been reported, and various theory and tools have been revealed that can help one to understand and improve the program comprehension activity (R. Brooks, 1983; Letovsky, 1987; Mayrhauser & Vans, 1993; Pennington, 1987; Shneiderman & Mayer, 1979). Some of the researches were focused on novices, some other focus on expert programmers.

One of the main obstacles in program comprehension is due to the large number of statements in program codes. Most of the program comprehension tasks are only concern will certain function and modules of the program code. Therefore, program slicing has been recognized as an enabling approach to improve the program comprehension work (Francel & Rugaber, 1999; Lanubile & Visaggio, 1993).

Program slicing is a technique of program code analysis that has been developed to assist programmer to analyze program codes. Program slicing is an activity where the program code will be sliced into a smaller form based on certain parameter and targeted program line (Weiser, 1984). Basically, the idea of program slicing is to identify those program statements that are only relevant to the context of slicing, and remove other statements that are not relevant. The context of slicing is here defined as slicing criteria, which is a subset of variable of interest and at a specified line of program codes. Three types of program slicing are available; backward slicing, forward slicing and dicing. In backward slicing, the program slice is computed by working backwards from the point of interest. Forward slicing works by tracing forwards from the point of interest, whereas, dicing combines both backward and forward slicing.

This research work proposes Knowledge-Based Slicer (KBS), a tool that function to guide novice programmers in their program comprehension activities. KBS will be realized by integrating currently available Java open source program slicing tool, known as Indus-Kaveri, with a knowledge-based component built using a source code similarity analyzer, Simian, and both will be deployed as Eclipse plugin.

1.2 Research Motivation

The motivation behind this research is driven by the needs to offer a tool that is suitable to be utilized in the learning of program comprehension of novice programmers in local universities. Program comprehension knowledge and skill is crucial in helping programmers to carry out their duties. Therefore, formal

introduction and exposure of program comprehension is deemed important in the pedagogy of Computer Science.

Experiment reported by Francel and Rugaber (1999) shows that program comprehension is more effective for programmers that slice the program code than programmers that do not slice during debugging. Thus, program slicing is viewed as an effective technique to achieve better program comprehension. However, Gold et al. (2005) have reported that one of the obstacles in applying program slicing is the difficulty in identifying slicing criteria. Therefore, guided selection of slicing criteria is deemed as necessary element in order to facilitate programmers in applying program slicing.

On the other hand, Aljunid (2009) highlighted that his cognitive model of program comprehension can be utilized as a means for learning of program comprehension by having knowledge-based support and utilizing the program slicing technique. In the context of novices, knowledge-based support is considered crucial to help novices to better understand the program codes. This is due to the novices' fragile knowledge that is defined as inadequate and partially memorized knowledge, and hard to be retrieved (Perkins & Martin, 1986). Whereas program slicing can be utilized to reduce the program code complexity by focusing on parts relevant to certain contexts.

Many program slicing tools have been developed, and among the open-source tool for Java is the Indus (Jayaraman, Ranganath, & Hatcliff, 2005) a static program slicer. This tool is available as either standalone program or as Eclipse's plugin known as Kaveri (Jayaraman, et al., 2005). Among its purpose is to assist

program comprehension and debugging. However it is only implemented solely based on program slicing technique.

Based on reviewed literature, there is no known program comprehension tool that utilized both knowledge-based and program slicing, whereas such a tool can be beneficial to the pedagogy of program comprehension for novice programmers. Therefore, this research will propose that tool by extending the Indus-Kaveri slicing tool with knowledge-based support. This tool can be used by novices in the learning of program comprehension, by leveraging the constructivist-based cognitive model of program comprehension as proposed by Aljunid (2009).

1.3 Problem Statements

i). General Problem Statements

- Program comprehension skill is crucial but neglected in Computer Science teaching and learning.
- Program comprehension is a complex yet crucial task for novices, therefore an effective program comprehension tool for novices is required.
- Novice programmers are lack of several crucial types of knowledge compared to experts, which hamper their comprehension and programming, thus knowledge-based support considered crucial to temporarily support them.
- Available tools only provide program slicing technique without knowledge-based support.

ii). Specific Problem Statements

- Most novices does not know nor apply the program slicing when comprehending, whereas studies have shown that experts and effective novices can comprehend program better by applying program slicing.
- One of the obstacles in applying program slicing is the difficulty in identifying slicing criteria.
- No known program comprehension tool offer program slicing technique with knowledge-based support, whereas these dual complementary techniques can be used in tandem to improve novices' comprehension.

1.4 Objectives

The objectives of this research are as follows:

i. To facilitate novices in applying program slicing technique by providing guided selection of slicing criteria.
ii. To extend and integrate the existing Indus-Kaveri slicing tool with knowledge-based component built using Simian source code similarity analyzer.
iii. To evaluate the program comprehension effectiveness of the proposed tool.

1.5 Scope

i. The subject of research is focused only to novice programmers, i.e Computer Science and Information Technology undergraduate students having basic knowledge in programming.

ii. In program comprehension, the type of external artifact to be considered is only text based and not graphical (diagram) of the program code.

iii. The program slicing technique is based on backward static slicing.

iv. The knowledge-based known as Program Plans, will be limited to three basic novices' algorithm i.e (1) Calculating Total, (2) Calculating Average, and (3) Calculating Maximum.

v. The testing will be conducted based on the relevant self-written and actual novice program codes.

vi. The selected programming language to be analyzed is limited to Java, containing only single procedure.

vii. The input for tool testing and evaluation will be of 2 types: (1) Sample Programs, and (2) Real Novices' Code, and the maximum line of codes is 50 lines.

1.6 Significances

Two research significances have been identified for this research work:

i. To assist instructors and lecturers in conducting program comprehension topics for CS students, especially in practical part.

ii. To expose the program comprehension technique using program slicing and knowledge-based approach to novices for them to be able to perform better task in their future career.

CHAPTER II

LITERATURE REVIEW

2.1 Theories of Program Comprehension

Programmer is commonly known as a profession where the main activities are to write and compile programs. But, in reality, apart from writing and compiling, they have to read and understand the pre-written program codes, in order to enable them to perform other tasks, such as debugging, enhancing, and re-engineering. The activity of reading and understanding program codes is known as Program Comprehension.

Program comprehension topic has been discussed much early by the software engineering society in the first software engineering workshop (Naur & Randell, 1968). A couple of decade has passed, and numerous discussions and research findings have been achieved in program comprehension field. The movement was driven by a similar goal, which is to develop and propose methods and tools that can be utilized by the software engineering community in the diverse software engineering activities, such as inspection, reusing, debugging, enhancement and re-engineering.

Brooks (1977) was among the first who proposed the program comprehension model which was based on the context of various knowledge about the programming language and the application domain. Pennington (1987) also

proposed a model which was inspired by various knowledge base and it was blended with the theoretical concept of how one read and comprehend texts.

Quite a number of literatures have been published proposing the cognitive models of program comprehension. The motivation of these models is to explain the process that takes place in the programmers mind when they read program codes. A lot of discussions and reviews have been presented by previous researchers to further elaborate the cognitive models in various perspectives (Exton, 2002; Storey, 2006; Tilley, 2007). Basically, the cognitive models of program comprehension can be categorized into 5 models, (1) top-down model, (2) bottom-up model, (3) knowledge-based model, (4) opportunistic strategies model, and (5) integrated models.

For the *top-down model*, Brooks (1983) mentioned that programmer tends to understand program code in a top-down approach. To gain the specific idea of the program code, it starts with general idea of the background of the program. This general idea will be refined in a hierarchical structure, to a more focused idea. It is evaluated further down the hierarchy, as more program codes are being read, until the specific idea of the program code has been captured. The information used in each level of hierarchy are including beacons (R. Brooks, 1983), code features and code structures. The notion of idea is known as hypothesis. Soloway and Ehrlich (1984) observed that expert programmers use the information to decompose goals and plans into lower-level goals and plans.

In the *bottom-up model*, Shneiderman and Mayer (1979) proposed cognitive models that differentiate between syntactic and semantic knowledge of programs. They mentioned that understanding a program involves creation of

multilevel internal semantic structures in a bottom-up manner. This means that the programmers understand the function of a group of statements, consolidate them together to get higher levels of information until the entire program codes is understood. Whereas Pennington (1987) perceived that programmers will initially develop a control-flow of the program codes which captures the sequence of operations. Once this has been achieved, the data-flow abstractions of program will be established to construct the knowledge on the program goal. This is being proposed as the *program model* and *situation model* of thinking process.

Letovsky (1987) has proposed the *knowledge-based* model, where the process involves in the program understanding is made up of a recurring series of inquiry activities. This has been described as to make reasoning on the conclusion based on questions asked. The purpose of some variables or expressions in program code can be asked, and the answer can be found by conjecturing, which later will be verified by searching through the program codes or external documentation. Letovsky also perceived that this model can be achieved by exploiting both the top-down and bottom-up approach. There are three components that make up this model; (1) *knowledge base* which was defined as the programmers' expertise and background knowledge, (2) *mental model* is being defined as the programmers' current understanding on the program, and (3) *assimilation process* that described how the mental model developed using the current knowledge-based supplemented with the program codes.

The *opportunistic strategies* model has been coined by Letovsky (1987). According to him, programmers can be regarded as "opportunistic processor". They can easily change their program comprehension strategies in response to

external evidence. This dynamic ability is a crucial success factor that contributed to the effectiveness and efficiency on program comprehension demonstrated by expert programmers.

The fifth model is *integrated model* proposed by Mayrhauser and Vans (1993) combining the models mentioned previously, especially the models by Brooks (1977), Letovsky (1987), and Pennington (1987). Mayrhauser and Vans claimed that the models used may vary depending on the tasks and the programmers' command of knowledge on the problem domain and programming language. Programmers with a better understanding of domain are more likely to take the top-down model, while those with less programming knowledge prefer bottom-up model in program comprehension. But, they may also employ a hybrid model, where they will simultaneously switching between models as they progress between different levels of abstraction.

2.2 Program Comprehension of Novice Programmers

Comparison between novices and expert programmers in program comprehension have been studied and proposed in various models including Berlin (1993), Pennington (1987). Soloway and Ehrlich (1984) highlighted that expert programmers employ high-level plans, while Koenemann and Robertson (1991) concluded that experts programmers were frequently use top-down model.

Holt et al. (1987) examined programmers' cognitive model by making modification to the program, either a simple one, or a complicated one. It was achieved using three different design methodologies. Whereas Burkhar and

Wiedenbeck (1998) analyzed object-oriented program comprehension by novices and experts in three dimensions of comprehension strategies. The strategies are (1) the scope of the comprehension, (2) the top-down versus bottom-up direction of the processes, and (3) the guidance of the program comprehension activity. They found strong evidence of top-down, inference-driven behaviors, as well as multiple guidance in expert programmers' comprehension.

Robins et al. (2003) highlighted that there are three area of interest that one can find the gap between novices and expert programmers, which is by (1) their knowledge representation, (2) problem solving strategies, and (3) mental models. On the other hand, in Vessey's (1985) exploratory study of programmer's debugging processes, she had classified programmers as expert or novice based on their ability to chunk effectively. She mentioned that expert programmers used breadth-first approaches. At the same time, they were able to adopt a system view of the problem area. Whereas novices used breadth-first approaches but were unable to think in system view (Vessey, 1985).

2.3 Program Slicing for Program Comprehension

Program Slicing is a technique that allows programmer to view a subset of program codes by slicing out program codes that are not relevant to the programmer's interest. The resulting subset of program is called as program slice. The reduced program slice is achieved by analyzing either the data flow, or control flow of the program code. Even though the original program codes have been partially removed, the behavior or computation of the program is still equal to the

computation of the original program codes (Weiser, 1984). This is however dependent on the meaning of the slicing criteria, which is a subset of variable of interest and at a specified line of program codes.

The motivation behind program slicing is to aid debugging and program comprehension by reducing the program codes complexity. In debugging, when program slicing technique is applied, the total errors debugged and total errors found are slightly increased (Shuhaidan, 2006). The technique that has been employed is to remove program code lines from the source code that do not affect or being affected by the values of variables at a specified program code line. There are many ways to achieve this (Bergeretti & Carre, 1985; J. R. Lyle & Weiser, 1987; Weiser, 1984), but generally, they all will achieve the same result.

2.3.1 Program Slicing Basic Steps

As mentioned before, a program slice is computed based on slicing criteria. The slicing criterion is based on two attributes, which is (1) a specific point of interest, and (2) a set of variables. The approach to compute a program slice by Weiser is based on iterative data flow analysis (Weiser, 1979, 1984). Another important approach was proposed by Ferrante et al. (1987) is by using reachability analysis in Program Dependence Graphs (PDG).

PDG mainly consist of nodes which represent the statement of a program code, and edges which represent the control and data dependency. Either using the first or second approach, the program slice will be achieved by containing only statements that are affecting and affected by the values of the variables at the given point of interest.

The program slice is the reduced version of the original program code, as shown in Figure 2.1. Weiser (1984) mentioned that even though it has been sliced, it should be executable. Some program features are difficult to be sliced and some are easy. Unstructured control flow such as ‘goto’ statements are very difficult to be sliced. Indirection in a program such as pointer and array also makes slicing more difficult. As a matter of fact, in the general case program slicing is a an undecidable problem (Weiser, 1984).

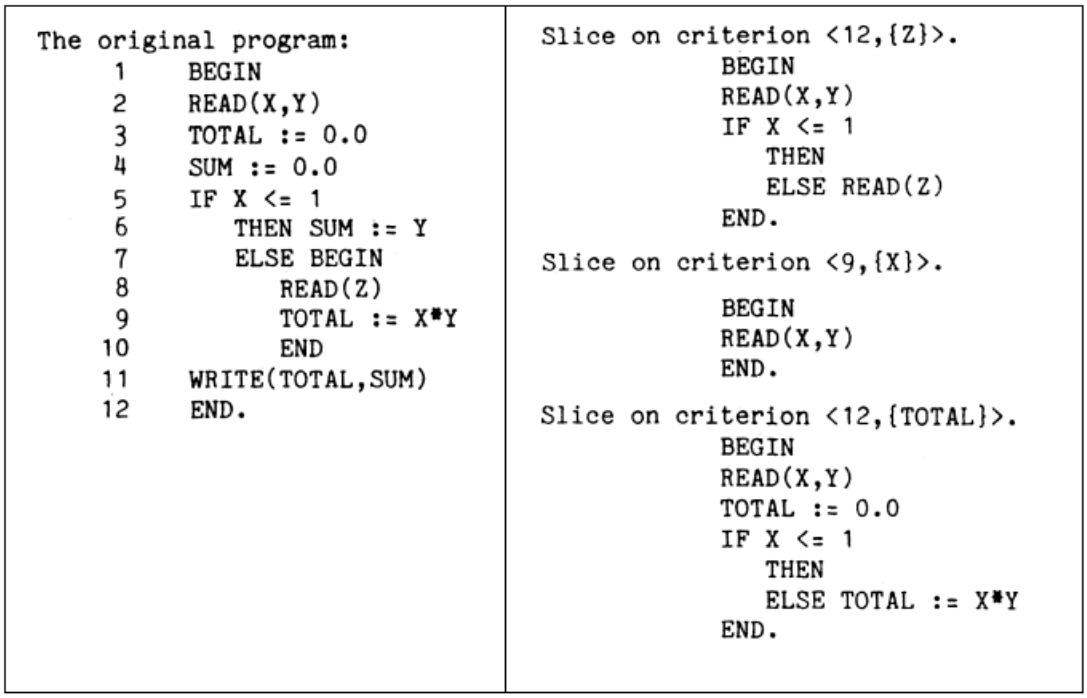

Figure 2.1 Examples of program slice
Source: (Weiser, 1984)

In order to carry out program slicing, one has to define it in such a way that a slice is only equivalent to the original program when the original program terminates. Furthermore, a strictly minimal slice cannot be found, and only an approximation can be computed. However, the approximation output

is usually good enough and program slicing is still a useful technique in reducing a program code complexity.

2.3.2 Different Types of Program Slicing Techniques

There are several different techniques for program slicing that have been proposed for the past thirty years. The most common form of program slicing technique is backward slicing. In backward slicing, the program slice is computed by working backwards from the point of interest. The process is to find all statements that can affect the value of specified variables at the targeted point of interest, and slicing out other statements deem irrelevant.

Another technique of program slicing is forward slicing. As the name implies, the process of forward slicing is the inversion of backward slicing, in which the process objective is to find all statements that can be affected by changes made in the specified variables at the point of interest. Bergeretti and Carre (1985) were the first to define the notion of a forward slice. The terminology was further elaborated by Reps and Bricker (1989).

Apart from that, J. R. Lyle and Weiser (1987) had introduced another technique called dicing. This technique will combine the result of different program slices, each was computed with respect to different variables. By combining these results, the combined program slice will expose more information on the possibility of the value of one variable is being affected by another value.

The scope of this research is limited to backward static slicing. The reason behind this undertaking is that the research is focusing on teaching and

learning of program comprehension targeting at novice programmers. To make things simpler, novices will be introduced with backward slicing by identifying the last statement containing final result of a specific calculation. From there, the slicing will be calculated by identifying previous statements that are affecting the last statement. In the remaining part of this thesis, term ‘slicing’ will be used to represent ‘backward static slicing’.

2.3.3 Three Classes of Program Slicing Techniques

The previously mentioned different techniques can be categorized into 3 classes, (1) Static Slicing, (2), Dynamic Slicing, and (3) Hybrid Slicing. Basically, Static Slicing works by statically analyzing the code, which means examining some representation of the source code without actually executing the program being analyzed. Whereas in Dynamic Slicing programmers will analyze the code by executing the program. To dynamically slice the program, one has to provide an input as part of the program criterion, therefore, the resulting program slice is only correct for a specific input. By contrast, a static slice is correct for all input. Example of static and dynamic slicing is depicted in Figure 2.2.

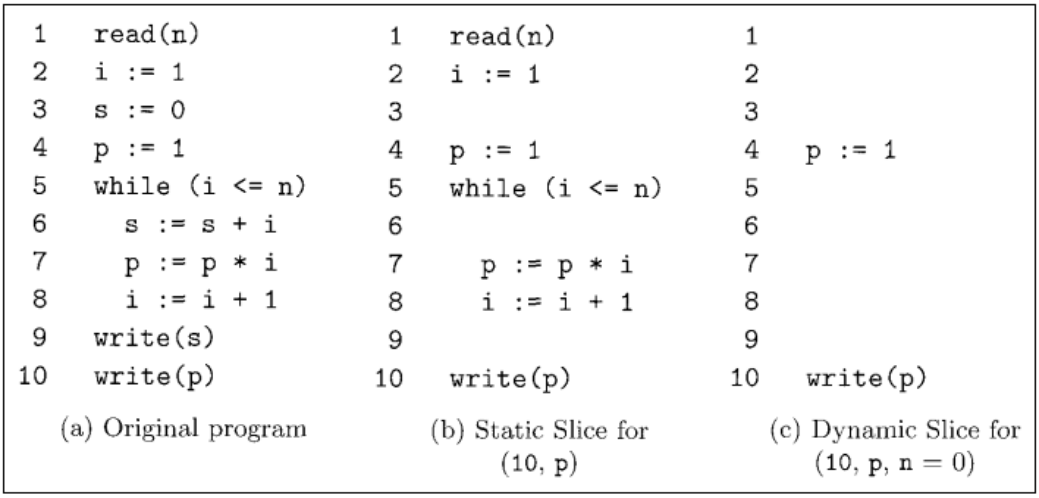

Figure 2.2 An example of static and dynamic slicing
Source: (Krinke, 2005)

As the name suggest, Hybrid Slicing is an approach that combine static and dynamic slicing. Part of the program codes are sliced using static slicing, whereas another part of the program codes are sliced using dynamic slicing. Quasi-static and Conditional Slicing are two types of Hybrid Slicing. Quasi-static slicing was introduced by Venkatesh (1991), in which he suggested the program slice to be computed with respect to an initial prefix of the input sequence to the program. Canfora et al. (1994) had proposed a notion of Conditioned Slice. Basically, the result of Conditioned Slice is a subset of program codes which preserves the behavior of the original program. It is computed with respect to a slicing criterion for a given set of execution paths.

All approaches to program slicing discussed so far have been developed based on the syntax preserving concept. The property of these approaches is the computation will leave the syntax of the original program largely untouched and simply remove irrelevant statements to create the

program slice. Another approach to program slicing where the syntax preserving aspect is being ignored is proposed as Amorphous Slicing (Harman & Danicic, 1997). By using this approach, the slicing process will utilize a code transformation program that will alter the program code to make it simpler. However, it still preserves the behavior of the program with respect to the slicing criterion. The main advantage of Amorphous Slicing is that the produced program slice is considerably smaller than their syntax preserving counterparts.

2.3.4 Program Slicing Tools

Different algorithms have been devised to implement program slicing based on different approach presented in the previous section. Each algorithm is language independent, however they might need some tweaking for the specific language they intended to slice, due to the different constructs and paradigms present in different programming language. As the result, many tools have been developed based on similar or different algorithms and program slicing techniques to demonstrate its usefulness.

Among the earliest developed tool was Wisconsin Program Slicer ("Wisconsin Program-Slicing," 1996), developed based on the System Dependence Graph (Horwitz, Reps, & Binkley, 1988) for interprocedural slicing. The tool has the can be used to perform forwards and backwards slicing of C programs, however, it only supports static slicing. The initial version of Wisconsin Program Slicer was distributed freely, which later being taken over by GrammaTech ("Static Analysis," 2000), Inc. and developed

further with a different name, coined as Codesurfer ("Code Browser," 2007) and promoted as a commercial tool.

Another widely known tool is Unravel ("The Unravel," 1998). It supports only static backward slicing of C programs, but without support of the goto construct (J. Lyle & Wallace, 1997). It is freely available slicing tool which runs under a UNIX/Linux environment.

The Kansas State University had developed a slicing tool for Java program, published as Indus (Ranganath & Hatcliff, 2007). It was developed based on Bandera (Corbett et al., 2000) program analysis framework from the same university. Indus program slicer has been presented as an Eclipse plugin called Kaveri (Jayaraman, et al., 2005). This program slicer support static forward and backward slicing. It can handle concurrent program codes.

Be the first publicly available Java implementation of program slicing, the Indus has been developed to support static forward and backward slicing. Apart from that, it allows one to slice concurrent programs, by considering data interference and other synchronization related aspect that are present in the concurrent programs. However, it does not support some advance Java features including dynamic class loading, native method and reflection (Jayaraman, et al., 2005).

The modularity attribute of Indus allows it to be utilized as command line program, or embedded inside another Java program as sub-component. To quickly utilize its power, especially in an integrated development environment (IDE) program, Kaveri has been developed as Eclipse plugin. Eclipse is well-known and widely used program development, deployment and analysis tool.

Kaveri contributes the following features to Eclipse, (1) viewing the program slice in the Java editor, (2) choosing slice criteria, (3) *chasing* dependencies (Jayaraman, et al., 2005) to support program comprehension, and (4) performing context-sensitive slicing.

2.4 Knowledge-Based Support for Program Comprehension

Knowledge-based technique has been accepted as another effective approach that can assist programmers in their program comprehension activities. This technique works by providing high-level support and explanation as per human expert level assistance to programmers. A number of previous works had proposed knowledge-based program comprehension (AlOmari, 1999; Harandi & Ning, 1988; Johnson & Soloway, 1985; Murray, 1989; Sani, Zin, & Idris, 2009).

The findings by Harandi and Ning (1988) mentioned that knowledge-based systems are able to provide syntactic and semantics aspects of program codes, and also to recognize familiar patterns of program codes that can be utilized by programmers to gain understanding of a particular program codes. The basic idea of knowledge-based program comprehension approach is by comparing the input source code and the code snippets from the library or repository. These code snippets are often called plans, clichés, chunks, etc. Since the description and meaning of plans is already known, one can easily say what a piece of source code does, if one can find a match between that piece of the source code and a plan (Taherkhani, 2011).

Finding a match between source code and program plan can also be considered as finding similarities between two different program codes. Detecting

program similarities has been a research motivation in the area of clone detection and plagiarism (Taherkhani, 2011). The initial objective of this technique is to reveal plagiarism between students work (Taherkhani, 2011), and clone detection can be employed to assist programmers in software maintenance activities, such as refactoring (Mishne & De Rijke, 2004).

There are few program similarity tools that have been developed to find similar chunk of codes from a given collection of program codes. These tools are referred here as similarity analyzer. Some tools find the similarity by using pattern-based technique (CPD, 2004; Simian, 2004), whereas other analyzers implement code signature technique (Ghosh, Verma, & Nguyen, 2002; Jones, 2001; Schleimer, Wilkerson, & Aiken, 2003).

Simian and CPD are pattern-based analyzers that finds similarities between lines of codes by using two phase of program code analysis. In the first phase, the program code is transformed into internal atomic code representation. Then, using pattern matching algorithm i.e. Karp-Rabin matching algorithm and tiling algorithm, Simian will calculate every possible combination of the transformed program code (Mishne & De Rijke, 2004). Whereas, the code-signature analyzers find similarities only if the source code contains code signatures that mark similar piece of code. Since code-signature analyzers require program codes to be decorated with signatures or annotations, it is limited to be used for analyzing program codes that adhere to certain coding convention (Mishne & De Rijke, 2004). In the learning of program comprehension, novices are more exposed to program codes that are non-uniform in nature. Hence, pattern-based analyzer, which is Simian, is more suitable for the proposed tool.

Therefore, based on proposals made by Taherkhani (2011), in order to find matching between given source code and plans, adaptation of similarity-finding technique of source codes and plans are seen as plausible technique in detecting the meaning of source codes. In the remaining sections of this thesis, code snippets resembled in the form of plans will be referred as program plans.

2.5 The Pedagogically-based Model of Program Comprehension

Cognitive models of program comprehension proposed by various researchers (R. Brooks, 1983; Letovsky, 1987; Mayrhauser & Vans, 1993; Pennington, 1987; Soloway & Ehrlich, 1984) had be focusing around the understanding of the programmers' mental processes that takes place during carrying out of some specific tasks, and mostly focused to expert programmers. As of this moment, none of those surveyed models have the inclination towards pedagogically-based model of program comprehension, except the constructivist-based cognitive model by Aljunid (2009).

Aljunid's (2009) mentioned that the pedagogy of program comprehension and debugging for novice programmer can be achieved by an iterative process of assisted understanding and debugging using knowledge-based, and code localization using program slicing. By applying these techniques over time, the novice programmers should be able to perform program comprehension and debugging by further neglecting the assisted understanding and debugging from knowledge-based (Aljunid, 2009:173).

2.6 Discussions

Program comprehension is what majority Computer Science students will do in their future profession as a computer programmer. Currently, this skill will be developed slowly as the students' progress in the real world arena of software development. However, during the initial stage of their career, they will find it very hard to comprehend program codes coming from unfamiliar software system(Bohnet & Dollner, 2007). Most program codes given to them are those having complex structure and carrying a lot of domain-specific meaning which they have not experienced before. Therefore, it is perceived that this branch of knowledge of program comprehension should be taught explicitly; theoretically and practically, in the university Aljunid (2009:3).

Program slicing is a technique that allows programmers to comprehend the program codes by reducing its complexity. The complexity of program codes is contributed by the presence of various statements to achieve various computation goals in the program design. Most of programmers' tasks are driven by a problem reduction, in which a real life problem such as finding logical error in a program will eventually result in correcting a few line of program codes. Therefore, reducing the problem space which is from thousand or even million lines of codes into a reasonable number of lines to be read will ease programmers' effort.

The rationale behind program slicing is that the technique will produce a subset of program codes by slicing out program codes that are not relevant to the programmer's interest. As mentioned in the previous section, the resulting subset of program is called as program slice. The reduced program slice is achieved by analyzing either the data flow, or control flow of the program code.

REFERENCES

Aljunid, S. A. (2009). *A cognitive Model of Automated Program Comprehension Cum Debugging for Novices.* Unpublished PhD, Universiti Kebangsaan Malaysia.

AlOmari, H. M. d. A. (1999). *CONCEIVER: A Program Understanding System.* Universiti Kebangsaan Malaysia, Bangi.

Bergeretti, J.-F., & Carre, B. A. (1985). Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst., 7*(1), 37-61.

Berlin, L. M. (1993). Beyond program understanding: A look at programming expertise in industry. *in Empirical Studies of Programmers: Fifth Workshop*, p. 8-25.

Bohnet, J., & Dollner, J. (2007, 24-25 June 2007). *Facilitating Exploration of Unfamiliar Source Code by Providing 21/2D Visualizations of Dynamic Call Graphs.* Paper presented at the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007.

Brooks, F. P. (1987). No Silver Bullet Essence and Accidents of Software Engineering. *Computer, 20*(4), 10-19.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies, 9*, 737–751.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18*(6), 543-554.

Burkhar, F. D. J.-M., & Wiedenbeck, S. (1998). *The Effect of Object-Oriented Programming Expertise in Several Dimensions of Comprehension Strategies.* Paper presented at the Proceedings of the 6th International Workshop on Program Comprehension.

Canfora, G., Cimitile, A., Lucia, A. D., & Lucca, G. A. D. (1994). *Software Salvaging Based on Conditions.* Paper presented at the Proceedings of the International Conference on Software Maintenance.

Chikofsky, E. J., & Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw., 7*(1), p 13-17.

Code Browser Static Analysis Tool for C and C++ - GrammaTech CodeSurfer. (2007). from http://www.grammatech.com/products/codesurfer/overview.html

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, et al. (2000). *Bandera: extracting finite-state models from Java source code*. Paper presented at the Proceedings of the 22nd international conference on Software engineering.

CPD. (2004). PMD. Project Mess Detector. Retrieved October, 2012, from http://pmd.sourceforge.net/

Exton, C. (2002, 2002). *Constructivism and program comprehension strategies.* Paper presented at the Program Comprehension, 2002. Proceedings. 10th International Workshop.

Ferrante, J., Ottenstein, K. J., & Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst., 9*(3), 319-349.

Fjeldstad, R. K., & Hamlen, W. T. (1983). *Application Program Maintenance Study: Report to Our Respondents.* Paper presented at the Proceedings GUIDE 48, Philadelphia, PA.

Francel, M. A., & Rugaber, S. (1999, 1999). *The relationship of slicing and debugging to program understanding.* Paper presented at the Seventh International Workshop on Program Comprehension, 1999. Proceedings.

Ghosh, M., Verma, B. K., & Nguyen, A. T. (2002). *An Automatic Assessment Marking and Plagiarism Detection System.* Paper presented at the First International Conference on Information Technology and Applications.

Gold, N. E., Harman, M., Binkley, D., & Hierons, R. M. (2005). Unifying program slicing and concept assignment for higher-level executable source code extraction: Research Articles. *Softw. Pract. Exper., 35*(10), 977-1006.

Harandi, M. T., & Ning, J. Q. (1988, 24-27 Oct 1988). *PAT: a knowledge-based program analysis tool.* Paper presented at the Software Maintenance, 1988., Proceedings of the Conference.

Harman, M., & Danicic, S. (1997). *Amorphous Program Slicing*. Paper presented at the Proceedings of the 5th International Workshop on Program Comprehension (WPC '97).

Holt, R. W., Boehm-Davis, D. A., & Shultz, A. C. (1987). Mental representations of programs for student and professional programmers. In M. O. Gary, S. Sylvia & S. Elliot (Eds.), *Empirical studies of programmers: second workshop* (pp. 33-46): Ablex Publishing Corp.

Horwitz, S., Reps, T., & Binkley, D. (1988). Interprocedural slicing using dependence graphs. *SIGPLAN Not., 23*(7), 35-46.

Jayaraman, G., Ranganath, V., & Hatcliff, J. (2005). Kaveri: Delivering the Indus Java Program Slicer to Eclipse. In M. Cerioli (Ed.), *Fundamental Approaches to Software Engineering* (Vol. 3442, pp. 269-272): Springer Berlin Heidelberg.

Johnson, W. L., & Soloway, E. (1985). PROUST: Knowledge-Based Program Understanding. *Software Engineering, IEEE Transactions on, SE-11*(3), 267-275.

Jones, E. L. (2001). Metrics based plagarism monitoring. *J. Comput. Sci. Coll., 16*(4), 253-261.

Koenemann, J., & Robertson, S. P. (1991). *Expert problem solving strategies for program comprehension.* Paper presented at the Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology.

Krinke, J. (2005). Program Slicing, *Handbook of Software Engineering and Knowledge Engineering, Vol. 3: Recent Advances* (pp. p. 307-332): World Scientific Publishing.

Lanubile, F., & Visaggio, G. (1993). Function recovery based on program slicing *In Proceedings of the Conference on Software Maintenance CSM-93*, p 396-404.

Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of Systems and Software, 7*(4), 325-339.

Lyle, J., & Wallace, D. (1997). *Using the Unravel Program Slicing Tool to Evaluate High Integrity Software.* Paper presented at the Proceedings of Software Quality Week.

Lyle, J. R., & Weiser, M. (1987). Automatic bug location by program slicing. In. *Proceedings of the Second International Conference on Computers and Applications*, p 877–883.

Mayrhauser, A. v., & Vans, A. M. (1993). From Code Understanding Needs to Reverse Engineering Tool Capabilities. *In Proceedings of the 6th International Workshop on Computer-Aided Software Engineering*, p. 230–239.

Mayrhauser, A. v., & Vans, A. M. (1997a). *Program understanding behavior during debugging of large scale software*. Paper presented at the Papers presented at the seventh workshop on Empirical studies of programmers.

Mayrhauser, A. v., & Vans, A. M. (1997b). Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance, 9*(5), 299-327.

Mishne, G., & De Rijke, M. (2004). *Source code retrieval using conceptual similarity.* Paper presented at the Proceeding of the 2004 Conference on Computer Assisted Information Retrieval (RIAO'04).

Murray, W. R. (1989). *Automatic Program DeBugging for Intelligent Tutoring Systems*: Morgan Kaufmann Publishers Inc.

Naur, P., & Randell, B. (1968). Software Engineering: Report of a conference sponsored by the NATO Science Committee. *Garmisch, Germany: Scientific Affairs Division, NATO*, Retrieved 2008-2012-2026.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*(3), 295-341.

Perkins, D. N., & Martin, F. (1986). *Fragile knowledge and neglected strategies in novice programmers*. Paper presented at the Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers.

Ranganath, V., & Hatcliff, J. (2007). Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer (STTT), 9*(5), 489-504.

Reps, T., & Bricker, T. (1989). Illustrating interference in interfering versions of programs. *SIGSOFT Softw. Eng. Notes, 14*(7), 46-55.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2), 137-172.

Rugaber, S. (1995). Program comprehension. *Encyclopedia of Computer Science and Technology, 35*(20), p 341–368.

Rugaber, S. (2000). The use of domain knowledge in program understanding. *Ann. Softw. Eng., 9*(1-4), 143-192.

Sani, N. F. M., Zin, A. M., & Idris, S. (2009). Implementation of CONCEIVER++: An Object-Oriented Program Understanding System. *Journal of Computer Science, 5*(12), 1009-1019.

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). *Winnowing: local algorithms for document fingerprinting*. Paper presented at the Proceedings of the 2003 ACM SIGMOD international conference on Management of data.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming, 8*(3), 219-238.

Shuhaidan, S. M. (2006). *The Effectiveness of Program Slicing as a Debugging Technique for Novices: an Experimental Study*. Universiti Teknologi MARA.

Simian. (2004). Simian, Similarity Analyser. Retrieved October, 2012, from http://simian.dev.java.net/

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *Software Engineering, IEEE Transactions on, SE-10*(5), 595-609.

Static Analysis for C and C++ - GrammaTech. (2000). from http://www.grammatech.com/

Storey, M.-A. (2006). Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Control, 14*(3), 187-208.

Taherkhani, A. (2011). Automatic Algorithm Recognition Based on Programming Schemas. *Proceedings of the 23th Annual Workshop of the Psychology of Programming Interest Group (PPIG'11), University of York, York.*

Tilley, S. (2007, 26-29 June 2007). *15 Years of Program Comprehension.* Paper presented at the Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference.

The Unravel Program Slicing Tool. (1998). from http://hissa.nist.gov/unravel/

Venkatesh, G. A. (1991). *The semantic approach to program slicing.* Paper presented at the Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies, 23*(5), 459-494.

Weiser, M. (1979). Program slices: formal, psychological, and practical investigations of an automatic program abstraction method". *PhD Thesis, University of Michigan.*

Weiser, M. (1984). Program Slicing. *IEEE Transactions on Software Engineering, 10*(no. 4), p 352-357.

Wisconsin Program-Slicing Project. (1996). *The Wisconsin Program-Slicing Tool, Version 1.0.1.*, from http://www.cs.wisc.edu/wpis/slicing_tool/