A Novel Approach to Program Comprehension Process Using Slicing Techniques

Elham Hosnieh[1], Hirohide Haga[2*]

[1] Graduate School of Global Studies, Doshisha University, Kyoto, Japan.
[2] Graduate School of Science and Engineering, Doshisha University, Kyotanabe, Japan.

* Corresponding author. Tel.: +81-774-656-978; email: hhaga@mail.doshisha.ac.jp
Manuscript submitted July 10, 2015; accepted September 1, 2015.
doi: 10.17706/jcp.11.5.353-364

Abstract: The target of this research is to determine how program slicing contributes to program comprehension and to enhance its functionality by applying the slicing tree concept to its implementation. Slicing tree is a concept that refers to automatically repeating program slicing while the slicing criterion is changeable until the program decomposes into its atomic parts. Using this technique offers several advantages over traditional program slicing. First, it facilitates the process of understanding. In the original slicing, even after the slicing, the number of lines remains excessive for easy understanding. By using slicing trees, the final nodes of the tree point to a relatively small fragment of the code, which is the starting point to understand the program's behavior. In other words, we have a bottom-up approach and our comprehension process becomes faster and easier. Furthermore, it is much easier to use our knowledge based on recognizing the patterns in smaller fragments of code. Recognizing the patterns will play a great role in comprehending the code and intuiting the intention behind it. Third, this method answers the programmer's question about deciding the next best slicing criteria for the next step, since it will be chosen automatically and efficiently. We applied the slicing tree concept to several program codes containing basic programming functions to analyze the efficiency of this technique and its contribution to improve program comprehension.

Key words: Program comprehension, program slicing, software engineering, tree structure.

1. Introduction

1.1. Background

Program comprehension is one of the core activities of software engineering through which programmers or software engineers actively acquire knowledge about a software system using different available resources, such as software source code, documentation, other programmers, stakeholders, etc. [1]. Program comprehension is needed in almost every step of software development including debugging, evolution, maintenance, code leverage, and reuse [2]. In this article, the major focus is on understanding software artifacts through a set of existing unfamiliar code, for example, code written by another programmer or the programmer's own code that has been forgotten or neglected.

The program comprehension process is time-consuming and expensive. It has been estimated that about 60% of the software evolution process is dedicated to understanding software artifacts. In fact, more time is spent on understanding than writing new codes [3]. Moreover, comprehension is not an easy task especially if no systematic supporting documentation is available. Software evolution becomes fairly easy

once it is understood. In practice, more skills and expertise are needed for understanding and analyzing software source code than writing it [4]. This implies that more effort should be exerted to improve reading and comprehension skills in the training courses for novice programmers.

Program comprehension is an interdisciplinary field that exists at the intersection of software engineering and cognitive science. Much research has studied both cognitive models of human mind theories that explain how programmers comprehend software as well as cognitive tools, code analyzers, and mechanisms that assist programmers in the task of comprehension [2]. Both of these perspectives facilitate the establishment of mapping between the programmer's mental model of a software system and the software itself. During the past decades several cognitive software tools have been developed to provide a higher level of abstract representation of a program [5]. In this article, we focus on the second part: facilitating the cognitive assisting tools that are suitable for human processing.

The basic idea of program slicing attempts to reduce the size of the source code to reduce the size of the problem. The goal only investigates those lines of the program that are necessary for the task of understanding and relevant to a specific criterion or viewpoint. In other words, a slice of a program is a subset of the original program statements that directly or indirectly are influenced or might influence the value of the slicing criterion. The process of extracting the appropriate subprogram (slice) is called program slicing. Based on different theories, slices can be either executable or arbitrary. Depending on the goal, we choose one of the above approaches. If precision has more importance, we usually produce executable slices. If we give the priority to semantics to simplify the understanding task as much as possible, we choose the second approach. Program slicing has been divided into three different types: backward slicing, forward slicing, and dicing. In the case of backward slicing, the target is finding those statements that have affected a set of variables at a specific point of the program. In forward slicing, on the other hand, the focus is on the statements those are affected by the point of interest, and dicing combines both backward and forward slicing. Program slicing can be static or dynamic. Dynamic slicing makes some assumptions about the program's inputs, while static slicing does not. In this article, we focus on inexecutable, backward, static slicing.

1.2. Research Motivation and Objectives

The primary purpose of this article is facilitating the process of comprehending source codes. Our approach to solving this problem is to semantically decompose the source code. We chose program slicing for this purpose because it is a known decomposition and reduction technique for extracting knowledge from source code. A previous experiment showed that the program comprehension process becomes much more effective when slicing techniques are used [6]. More specifically, this article's target is improving the process of program slicing that contributes to program comprehension. This work proposes a novel approach to program comprehension using a slicing and code analyzing tool called *CodeSurfer*® by Grammatec Inc. Using this perspective enhances the functionality of program slicing for understanding by applying the slicing tree concept to its implementation.

As stated earlier, the effectiveness of program slicing in the program comprehension process is undeniable. However, some challenges and obstacles must be surmounted to apply program slicing. One is the difficulty faced by programmers when they want to specify appropriate slicing criteria [7]. The slicing criterion is a pair ***<s, v>*** in which "***s***" is a statement in code and "***v***" is the set of variables observed at statement "***s***". Improving the slicing process in such a way that slicing criteria can be chosen or suggested automatically while the semantics of the program persevere enhances program comprehension. Moreover, we know that program slicing reduces the number of lines in a program by omitting the unnecessary parts. However, especially in large programs, even after slicing is done, the number of lines in a resultant slice might remain too high to be understood easily. When the slicing result is still large, we can repeatedly apply

slicing to the result of the previous slicing. Applying slicing recursively constructs a tree-like structure of slices. Using the slicing tree will be useful to address both issues.

2. Program Slicing and Its Contribution to Program Comprehension

2.1. Definition of Program Slicing

One of the most natural ways of program understanding, which is common among programmers, is decomposing code into smaller parts or program subsets. This is simultaneously time-consuming and difficult for most programmers. Program slicing is a well-known reduction and analytic technique that reduces program complexity by decomposing it semantically, extracting the desired segments of the code, and discarding the irrelevant parts [8]. The resulting desired segments are a set of all the statements that might affect or are being affected by the values of a set of variables computed at some points of interest. That point of interest is generally a pair of a program statement or a line, a set of variables or ***C = <s, v>***, referred to as slicing criteria. Those program statements, which directly or indirectly affect the value of the set of variables specified in the slicing criteria, are called program slice *S* with respect to slicing criteria ***C***. Fig. 1 shows the example of original program (a) and its slice with the criterion ***C***=<10, product>. The process of extracting a slice is called program slicing. Mark Weiser, who first introduced program slicing in 1984, defined a program slice as the reduced version of a program whose behavior remains equal to the computation of the original program codes and is still executable, However, since that time, many researchers, such as Horwitz, Reps, and Blinkley [9], have proposed algorithms that extract non-executable slices [10], [11]. One major challenge in program slicing is that some features of the program are difficult to slice. Unstructured control-flow and unconditional jumps such as “goto”, “break,” and “continue” are very difficult to slice. Statements with such indirect structures as “pointers” and “arrays” also complicate slicing tasks [11].

```
(1)   read(n)                          read(n);
(2)   i := 1;                          i := 1
(3)   sum := 0;
(4)   product := 1;                    product := 1;
(5)   while i<= n do                   while i<= n do
      begin                            begin
(6)         sum := sum + i;
(7)         product := product *i;           product := product *i;
(8)         i := i+1;                        i := i+1;
      end;                             end;
(9)   write(sum);
(10)  write(product)                   write(product)
```

(a) (b)

Fig. 1. (a) Example program. (b) slice with slicing criterion (10, product).

Various aspects should be considered when we talk about program slicing [12]:

- **Computational method**: The method of computing the slice can be based on the data flow or the control-flow analysis. It may also be done through graph reachability in different types of dependence graphs.
- **Slicing direction**: The slice may be computed in either a forward or backward direction.
- **Scope of resultant slice**: The slice’s scope may be inter-procedural or intra-procedural.
- **Output format:** The output format of the computed slice may be in the form of code, a dependence graph, or an execution tree.
- **Different slicing techniques classes**: Program slicing techniques can be categorized into several different classes. The most important are static slicing and dynamic slicing.
- **Slicing output type**: Depending on the goal of computing the slice, it can be either executable or non-executable.

2.2. Different Variations of Program Slicing

There are three major variations of program slicing methods: backward slicing, forward slicing, and chopping. Other variations are also found, including dicing and barrier slicing [13]. A program can be traversed forward or backward from the slicing criterion or a combination of them. These terminologies have been proposed gradually to meet the various needs of program comprehension.

Backward slicing, which is the most common technique for program slicing, is widely used for code comprehension. In this method, the slicing process begins from our points of interest in the program and goes backward to investigate the program's statements that have affected the value of the variable(s) specified in the slicing criteria. Other statements are discarded [11].

Another major variation of slicing is forward slicing, which investigates the code in the opposite direction of backward slicing. Forward slicing extracts all those statements of the program that may be affected by the changes made in the value of the variable(s) at the program point specified in the slicing criteria [11].

Program chopping is a form of program slicing that combines both backward and forward slicing. It separates the slicing criteria, and the goal is to extract the statements of the program that can cause a change from the source to the target.

Our approach in this article is based on and limited to backward slicing. We made this decision because our research target is exposing the goal underlying writing a program code by starting from the final result to discover all the program statements that played a role in achieving this result and that are necessary for comprehending the program.

2.3. Different Classes of Program Slicing Techniques

Depending on the application areas, program slicing techniques can be divided into four main classes: static slicing, dynamic slicing, hybrid slicing, and amorphous slicing. When only statically available information is used for slicing the program, the process is called static slicing. This slicing technique does not make any assumption about the program inputs and analyzes the program without actually executing it. On the other hand, dynamic slicing is based on test cases and what happens at a specific execution of the program. The slicing criteria of dynamic slicing consist of three items: program input, occurrence of the statement, and a set of variables. The resultant slice of static slicing is usually very large compared to dynamic slicing. Because a dynamic slice is calculated with fixed values and is only specific to one execution path, many statements necessary to static slicing are naturally omitted from dynamic slices [10]-[12].

```
 1  read(n)              1  read(n)              1
 2  i := 1               2  i := 1               2
 3  s := 0               3                       3
 4  p := 1               4  p := 1               4  p := 1
 5  while (i<=n)         5  while (i<=n)         5
 6    s := s+i           6                       6
 7    p := p*i           7    p := p*i           7
 8    i := i+1           8    i := i+1           8
 9  write(s)             9                       9
10  write(p)            10  write(p)            10  write(p)
```

(a) Original program | (b) Static Slice for (10, p) | (c) Dynamic Slice for (10, p, n=0)

Fig. 2. Examples of static and dynamic slicing.

The other slicing classes are hybrid and amorphous slicing. Hybrid slicing, which is also called refined or efficient dynamic slicing, combines static and dynamic slicing. In this method some parts of the program are sliced using static slicing and others using dynamic slicing. All the classes discussed so far preserve the program's syntax. In other words, although some statements are omitted, the remaining statements stay

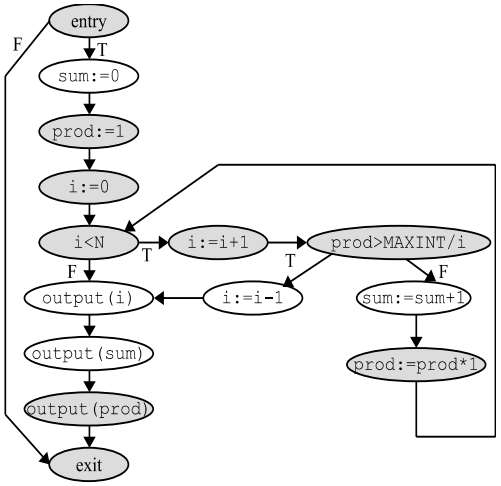
untouched. Amorphous slicing was introduced by Harman and Danicic [8] who described the type of slicing that sometimes for the sake of simplicity changes the code syntax while preserving the program semantics. Fig. 2 represents the difference of static and dynamic slices with same criterion of original program.

2.4. Program Slicing Computation Technique

There are two main approaches to program slicing computation. In both of them, to compute a program slice, different dependency relations among program statements should be analyzed. In fact, normally we need to reach an intermediate representation of the program such as dependency graphs before being able to compute a program slice [14].

The first approach [8] is based on the iterative process of data flow and control-flow analysis. In this method, first the relevant relations for each node are directly investigated in the Control-Flow Graph (CFG). Fig. 3 illustrates the souce code and its CFG. Then indirectly relevant program statements are gradually recognized and added to the slice. The process continues as long as more relevant statements remain to be found and are added to the slice. This approach was extended to address intra-procedural static slicing [8], [10]. There are two major dependencies in every program: data dependence and control dependence. There is data dependency between statements **p** and **q** in the program when a variable is assigned a value in statement **p** and is used or referenced with the same value in the **q** statement: when def-use or def-order exists between the two program statements. Control dependency also exists between statements p and q in a program if statement **p** branches one way, and then statement **q** is eventually reached; if statement p branches the other way, then statement **q** may not be reached [9]. The CFG used in Weiser's terminology is a graph whose nodes are program statements and whose edges represent the sequence and flow of the program [15].

```
begin
  sum := 0
  prod := 1
  while i<N do
    i := i-1
      if prod>MAXINT/i then
        i := i-1
        break
      fi
      sum := sum+i
      prod:=prod*i
  od
  output(i)
  output(sum)
  output(prod)
end
```



(a) Sample Program (b) Control Flow Graph

Fig. 3. (a) Example program. (b) Control-flow graph of program.

The second major approach, which was first proposed by Ottenstein & Ottenstein [16] and applied in many other research works, is program slicing through graph reachability when the program is represented as a dependence graph. In this approach, slicing is presented as a reachability problem. When this strategy was first introduced, it was limited to intra-procedure slicing based on a Program Dependency Graph (PDG) for program slicing. Later, the scope of dependence graph-based slicing was extended to inter-procedural slicing using a system dependence graph (SDG) [14]. This approach consists of two main steps. In the first, the program's dependence graph is constructed. In the next step, the graph is traversed

from the node that represents the slicing criteria of the program and the produced slice based on the related algorithm. Depending on the desired slice, if it is backward or forward, the slice is included in all the statements that are reachable from the slicing criteria node [10].

A dependence graph is a directed graph whose nodes represent program statements and whose edges represent data and control dependencies among the program statements (graph nodes). The slicing criteria in all the slicing techniques based on this approach are mapped to nodes in the dependence graph. As we stated earlier, SDG is more comprehensive than PDG because PDG represents single-procedure programs and SDG is a collection of PDGs. For inter-procedure slicing, represented by SDGs, the slicing criteria are extended to the called and calling procedures.

In this article, our approach to program slicing is based on dependence graphs. Our focus is on single-procedure programs.

2.5. CodeSurfer and Its Application to Our Research

CodeSurfer, which is a program analysis, understanding, and inspection system for ANSI C that is based on system dependence graphs, is a fundamental intermediate structure for representing programs (http://www.grammatech.com). Its most important function is program slicing. We use this software for testing and analyzing different program codes.

2.6. Recursive Decomposition Method for Program Comprehension

The idea of this research is based on the cognitive function of a programmer's mind to comprehend a program. Programmers follow these steps when they face large, complicated codes:

- Decompose target program into smaller parts;
- Understand each unit separately;
- Compose the understanding result of each unit to achieve a sufficient level of understanding.

If these steps are applied recursively, i.e., one segment of code O is decomposed into several units O_1, O_2, .., O_m and each O_i may be decomposed into smaller parts O_i^1, .., O_i^n and so on. The question is when should we stop decomposing? The answer is when we reach the atomic part. This term refers to an object that cannot be decomposed anymore. In the case of program code, this understanding is achieved when the semantics of its algorithms, all the elements of the code, and their roles and relations have been identified. In other words, when we can answer the question "what does this program do and how does it do it?" With this theory in mind, applying a recursive decomposing method to current and subsequent program slicing techniques, slicing tools may improve the program comprehension process.

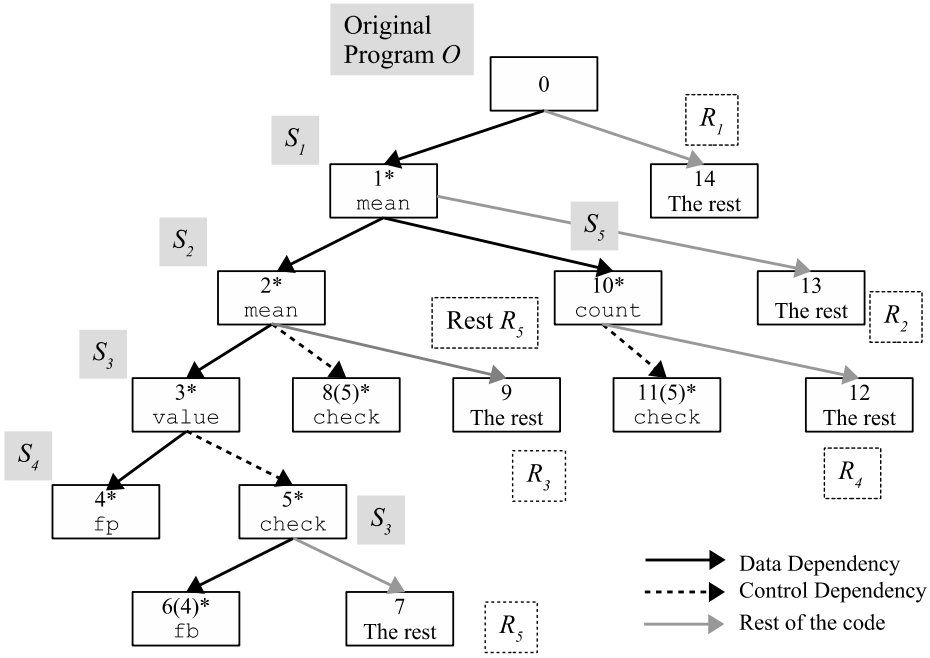
3. Discussion and Result

3.1. Slicing Tree and Its Applications to Our Research

Slicing tree is a concept that refers to automatically and recursively applying program slicing while the slicing criterion is changing [17]. In this method, the first slicing begins with regard to the slicing criteria specified manually by the programmer. That desired point usually is the return statement if the program returns any value or where the variable or variables are finalized that most help the programmer understand the semantics of the program. The slice is computed, and the rest of the code is ignored. After that, the variables on the right side of the first slicing criterion are chosen as new slicing criteria and separately continue to be sliced in the same way. This process continues until we reach an atomic object that can be easily understood, such as the first line of the program that represents the first semantic part of the program, for example, a line of code that opens a file. Note that this is a non-executable program slicing technique. Using this technique for comprehending a program or even understanding a function of a desirable variable has several advantages over traditional program slicing.

```
sum = 0;
count = 0;
fp = fopen(filename, "r");
if (fp == null) {
    puts("Cannot open this file!\n");
    return (-1);
}
check = fscanf(fp, "%d", &value);
while (check != EOF) {
    sum = sum + value;
    count++;
    check = fscanf(fp, "%d", &value);
}
if (count) {
    mean = (float)sum/count;
    printf("mean = %d\n", mean);
}
fclose(fp)
return 0;
```

Fig. 4. Example program.



S_1

```
 1: sum=0;
 2: count=0;
 3: fp=fopne("filename","r");
 8: check=fscanf(fp,"%d",&value);
 9: while(check!=EOF) {
10:     sum=sum+value;
11:     count++;
12:     check=fscaf(fp,"%d",&value);
13: }
14: if(count) {
15:     mean=(float)sum/count;
17: }
```

S_2

```
 1: sum=0;
 3: fp=fopne("filename","r");
 8: check=fscanf(fp,"%d",&value);
 9: while(check!=EOF) {
10:     sum=sum+value;
12:     check=fscaf(fp,"%d",&value);
13: }
```

S_3

```
 3: fp=fopne("filename","r");
 8: check=fscanf(fp,"%d",&value);
 9: while(check!=EOF) {
12:     check=fscaf(fp,"%d",&value);
13: }
```

S_4

```
 3: fp=fopne("filename","r");
```

S_5

```
 2: count=0;
 3: fp=fopne("filename","r");
 8: check=fscanf(fp,"%d",&value);
 9: while(check!=EOF) {
11:     count++;
12:     check=fscaf(fp,"%d",&value);
13: }
```

Fig. 5. Slicing tree of example program.

First, it facilitates the process of understanding. In the original method, even after reduction, the number of lines can still be excessive for simple comprehension. But with a slicing tree, the last nodes of the tree point to a very small fragment of the code, which is the starting point to understand the program's behavior and function. In other words, we have a bottom-up approach and our learning process will become faster and easier. Second, it is much easier to use our knowledge based on recognizing the patterns in smaller fragments of code. Recognizing the pattern will play a great role in comprehending the code and estimating the goal and intention behind it. Third, one obstacle in applying program slicing is the difficulty of identifying the most appropriate slicing criteria. The guided selection of slicing criteria seems to be an effective factor to facilitate programmers in applying program slicing. This method answers the programmer's question of which line and which variable are the best choices as slicing criteria for the next step that will be chosen automatically and efficiently.

Fig. 4 is a sample program and Fig. 5 illustrates its slicing tree. In the next section we analyze the effectiveness of applying slicing trees to program slicing using the *CodeSurfer*'s slicing tool.

3.2. Experiment

```
int main()
{
    int base, exp;
    long int value=1;
    printf("Enter base number and exponent respectively: ");
    scanf("%d%d", &base, &exp);
    while (exp != 0)
    {
        value *= base;
        exp--;
    }
    printf("Answer = %d¥n", value);
}
```

Fig. 6. Normal program slicing sample.

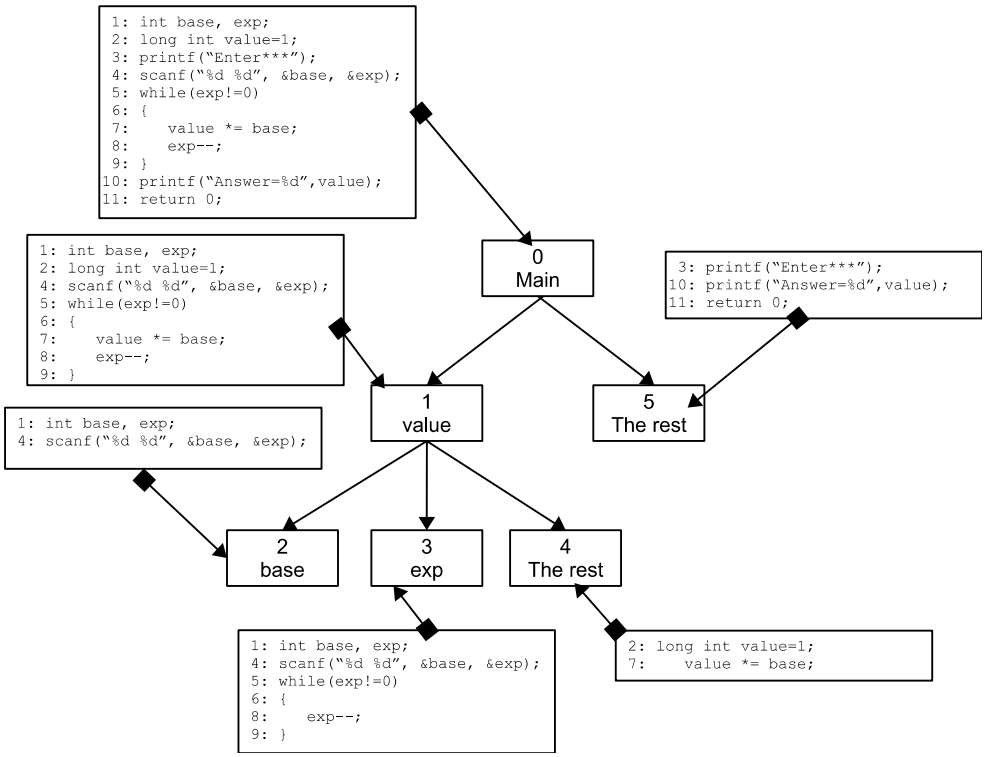


Fig. 7. Slicing tree for program sample (Fig. 6).

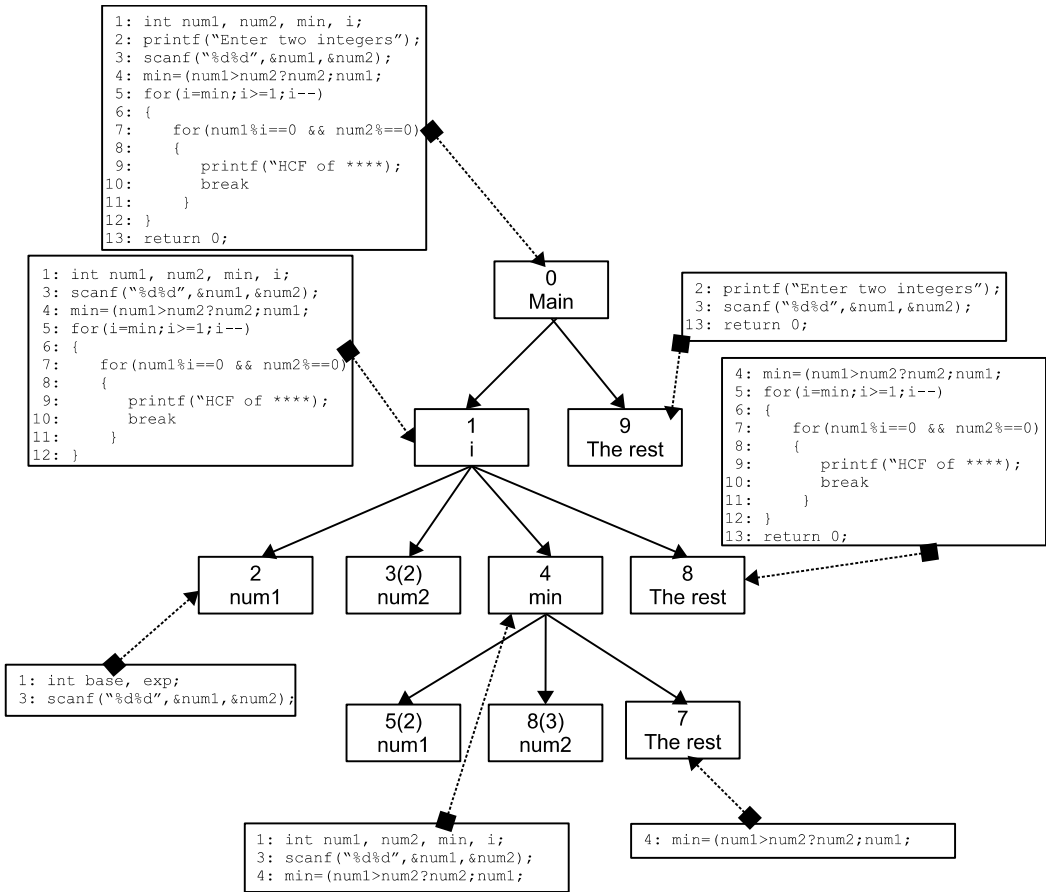


Fig. 8. Slicing tree for another program sample.

To prove the effectiveness of using slicing trees in the program comprehension process when being applied to program slicing techniques, we selected several different program codes containing such basic programming functions as bubble sorting, computing the highest common factor (hcf) of two numbers, computing the power of an integer, calculating the first n prime numbers, computing the area of a circle, and so on. Then we analyzed these programs by manually creating slicing trees for all of them and using *CodeSurfer*'s slicing tool for computing the slices at each step of creating the slicing tree. The slicing criteria at each step are decided by considering two groups of variables. The first group consists of those variables on the right side of our previous slicing criteria in the equations: those variables that have data dependency with the previous slicing criteria. The second group includes those variables that have control dependency with the previous variable. Our methodology is the analysis of the quality of comprehending the programs by comparing the results of normal slicing and slicing using the slicing tree concept. Fig. 6 illustrates an example of our experiments. In the rest of this chapter, we explain our sample program (computing power of an integer), and compare two forms of its representation. The program was represented once using the normal slicing technique and once using its slicing tree.

As Fig. 6 demonstrates, the program was sliced with regards to variable "value" at the line: "printf ("Answer = %d, value)", which is the target line of the program (the value of the number's power is shown). Therefore, we have one slicing criterion and one computed slice. Reducing the program to one big slice, especially if the program is large itself, will not be so easy to understand. Even though some irrelevant lines of code were omitted, the slicing process did not decompose the program into its atomic parts and did not provide any information about the different dependency relationships among the variables. However, in the other representation of the program, the slicing tree, slicing was repeated several times, and each time a new slicing criterion was chosen. As a result, we can clearly see which statements are necessary to

understand the behavior of each variable from a bottom to a top approach. The main code was sliced with the slicing criteria **(10, value)**, and the rest of the code was ignored. The resultant slice was sliced with regards to **(base, 7)** and **(exp, 8)**. As shown in Fig. 7, the remaining lines of the code are its atomic fragments, because no more variables can affect the computation of variable base or exp. Since the number of lines of code has been decreased to its minimum, it is very easy to understand how the values of the variable base and variable exp are computed. As the example demonstrates, the semantic fragments of code related to the computation details of each variable are sliced to points that are easily understood. In this example, there is data dependency between variables "base" and "value", and there is control dependency between variables "exp" and "value". Fig. 8 is an another example of slicing tree. These twe examples illstrate the concept of slicing tree clearly.

4. Future Works

As a next step of this research, this method can be applied to such program comprehension tools as *CodeSurfer* to improve the functionality of their program slicing options. The *CodeSurfer* application program interface (API) provides functions and types for interaction with *CodeSurfer*. *CodeSurfer*'s functionality can be extended with plug-ins, which may be written in either Scheme or C. If this function can be successfully implemented, the comprehension process will be achieved more efficiently and faster especially for large programs. First, tree-based program slicing will let programmers comprehend the whole program by dealing with smaller fragments of code. Second, by referring to slicing trees, the data and control dependencies of each variable are revealed, and such valuable information can be observed as the minimum semantic information needed to comprehend the behavior of each variable with regards to other variables.

Slicing trees are also an efficient tool for extracting familiar programming patterns (programming plans). Therefore, by adding knowledge based as another plug-in to *CodeSurfer*, additional information about the program can be provided to programmers.

Acknowledgment

This work was done while the first author (Elham Hosnieh) was with the Graduate School of Science and Engineering, Doshisha University. The authors would like to express their thanks to Grammatech Inc. for providing academic license and several supports of their product *CodeSurfer*.

References

[1] Zhang, Y. (2007). *An Ontology-Based Program Comprehension Model*. Ph.D. Dissertation, Concordia University.

[2] von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, *28(8)*, 44-55.

[3] Fjeldstad, R. K., & Hamlen, W. T. (1991). Application program maintenance study: Report to our respondents. In G. Parikh, & N. Zvegintzov (Eds.), *Tutorial on Software Maintenance* (pp. 13-30). IEEE Computer Society Press.

[4] Robson, D. J., Bennett, K. H., Cornelius, B. J., & Munro, M. (1991). Approaches to program comprehension. *Journal of Systems and Software*, *14(2)*, 79-84.

[5] Storey, M. (2005). Theories, methods and tools in program comprehension: Past, present and future. *Proceedings of 13th International Workshop on Program Comprehension* (pp. 181-191).

[6] Francel, M. A. (1999). The relationship of slicing and debugging to program understanding. *Proceedings of the 7th International Workshop on Program Comprehension* (pp. 106-113).

[7] Harman, M., Danicic, S., Sivagurunathan. Y., & Simpson. D. (1996). The next 700 slicing criteria. *Proceedings of the Second UK Workshop on Program Comprehension* (pp. 1-16).

[8] Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering, SE-10(4)*, 352–357.

[9] Horwitz, S., Reps, T., & Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, *12(1)*, 26-60.

[10] Raheman, S. R., Rath, A. K., & Bindu, M. H. (2013). An overview of program slicing and its different approaches. *International Journal of Advanced Research in Computer Science and Software Engineering*, *3(11)*, 435-442.

[11] Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Language*, *3(3)*, 121-189.

[12] Sasirekha, N., Robert, A. E., & Hemalatha, D. M. (2011). Program slicing techniques and its applications. *arXiv preprint arXiv:1108.1352*.

[13] Krinke, J. (2005). Program slicing. *Handbook of Software Engineering and Knowledge Engineering,* 307–332.

[14] Jha, L., & Patnaik, K. S. (2013). Proposed method for computing interprocedural slicing. *International Journal of Software Engineering*, *6(1)*, 83-96.

[15] Ball, T., & Horwitz, S. (1993). Slicing programs with arbitrary control-flow. *Proceedings of the 1st Conference on Automated Algorithmic Debugging* (pp. 206-222).

[16] Ottenstein, J., & Ottenstein, L. M. (1984). The program dependence graph in a software development environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (pp. 177-184).

[17] Hirakawa, I., & Sakai, S. (1995). Extracting programming knowledge through program slicing (in Japanese). *Technical Report of KBSE*, *95(173)*, 1-8.

Elham Hosnieh was born in Teheran, Iran in 1988. She received her B.Eng in information technology from Tabriz University, Tabriz, Iran and MSc in information and computer science from Doshisha University, Kyoto, Japan in 2012 and 2015 respectively.

After Finishing her master's program, she is now persuading her education toward and advanced doctoral program in global resource management and global studies in Doshisha University, Kyoto, Japan.

Miss Hosnieh is currently doing research in computational sociology. Her research interesets include computer modeling, sociology, and social psychology.

Hirohide Haga was born in Kyoto, Japan in 1954. He received his B.Eng, and M.Eng in electronics from Doshisha University, Kyoto, and Ph.D. in computer science from Kyoto University, Kyoto in 1978, 1980, and 1995 respectively.

In 1980, he joined to Hitachi, Ltd., where he was a research staff member at Systems Development Laboratory. In 1994, he moved to Doshisha University as a lecturer. Currently, he is a full professor of the Department of Intelligent Information Engineering and Science (IIES), Doshisha University, Kyoto, Japan. He served several visiting and invited positions including visiting researcher at Imperial College of Science and Technology, University of London in 1987, visiting scientist at University of Oulu, Finland in 2001, visiting professor of University of Cambridge from 2004 to 2005, and invited professor of École Centrale de Lille, France, in 2009 and 2011 respectively. He co-authored 4 books

including "*Programming in Java* (in Japanese)" and co-editied one proceedings (N. Callaos, D. Zinn, M. J. Savoie, X. Hu, R. Hill, and H. Haga (eds), *Proceedings of The 10th World Multi-Conference on Systemics, Cybernetics and Informatics*, Vol.4). He also published more than 60 academic articles.

Prof. Haga is a member of IEEE Computer Society, ACM, British Computer Society (BCS), Information Processing Society of Japan (IPSJ), Japan Society of Software Science and Technology (JSSST), and Institute of Electronics, Information, and Communication Engineers (IEICE) Japan. He was chartered as an IT professional from BCS in 2008. He was a board member of the Special Interest Group of Knowledge Based Software Engineering (SIG-KBSE) of IEICE in 2013 and a member of planning committee of JSSST from 1996 to 2000. His research interests include software engineering, especially software testing, multi-agent simulation, and digital serious gaming.