Search Based Slicing for Program Dependency Structures of Interest

Tao Jiang

King’s College London, Strand, WC2R 2LS, UK
(tao.jiang@kcl.ac.uk)

A Dissertation
Presented to the Faculty
of King’s College London
in London University for the Degree
of Doctor of Philosophy

December 2008
Abstract

Mark Weiser introduced program slicing: creating a subprogram defined by a set of variables and a specific statement. A large number of applications have since been developed based on this technique, from debugging and testing through program maintenance and integration to program metrics and refactoring. Recent developments in the efficiency of slicing tools (i.e. Codesurfer) allow for slices to be computed for each program statement in a reasonable time.

However, the technique presents a new challenge: How are we to identify useful subsets of slices to represent dependency structures of particular interest? This thesis explores dependency structures in programs using a Search-Based Slicing (SBS) framework as a way of responding to this challenge. The framework combines general Search-Based Software Engineering (SBSE) theory and the program slicing technique to make it possible to explore structures of interest in the large number of potential subsets of slices.

The goal of the SBS framework put forward in this thesis is as follows: to identify optimal subsets of slices guided by fitness functions in search based approaches. To begin with, a fitness function was defined to decompose programs with maximal coverage and minimal overlap between slices in the optimal set of slices. This application of the framework proved that the SBSE framework can be applied to program slicing for source code analysis.

Secondly, a greedy-optimisation splitting algorithm was introduced. The technique transferred passive metrics measurement – cohesion – to active procedure structure analysis. It was noted that the technique may apply to procedure reuse, to refactoring and to program understanding and maintenance. Finally, the empirical study considered over 800 procedures to summarise ‘patterns’ – procedure attributes which affect procedure ‘splitability’.

These experiments provided evidence that SBS is an attractive approach to source code analysis for exploring program dependency structures of interest.
Acknowledgements

First of all, special thanks to my first supervisor - Professor Mark Harman for his unparalleled and supreme instruction. I could never have accomplished this thesis without his supervision and advice. He gave me so much confidence, encouragement and support during my research.

Secondly, I would like to thank my second supervisors - Dr. Youssef Hassoun for his contributions and valuable suggestions. He put lots of energy into this thesis and the work throughout the whole process of my Ph.D. Dr. Nicolas Gold for his very helpful advice through my whole Ph.D. period.

I would like to thank Dave Binkley for his provision of source code and for his long term assistance in my work.

I would also like to thank Derek Jones for his suggestive advice and communication with regard to procedure splitting and program structure exploration.

I would like to thank Dr. Jens Krinke for practicing my mock Ph.D. viva which is very helpful for my success in the final viva. Also, many thanks to my examiners Dr. Marc Roper and Prof. Norman Fenton for their suggested comments on improving the quality of my thesis.

I particularly express my appreciation to my parents and wife for all their love and belief in me and for their sustaining my life.

I gratefully acknowledge the British Council for my financial support with the provision of the ORS scholarship.

Finally, many thanks to all the members of the CREST Centre. Their help has been priceless and I am highly in their debt.
Publications


## Contents

Abstract .................................................................................................................. iii  
Acknowledgements ............................................................................................. iv  
Publications .......................................................................................................... v  

1 Introduction ........................................................................................................ 10  
1.1 Program slicing ............................................................................................... 11  
1.1.1 Concept ....................................................................................................... 11  
1.1.2 Slicing for program dependency ................................................................... 11  
1.1.3 The issue of slicing-based dependency analyses ........................................... 13  
1.2 Search-Based Software Engineering (SBSE) ....................................................... 14  
1.2.1 Representation ............................................................................................ 14  
1.2.2 Fitness .......................................................................................................... 15  
1.2.3 Search-based approaches ............................................................................. 15  
1.3 Motivation of Search-Based Slicing ................................................................... 15  
1.4 Objectives of the Search-Based Slicing (SBS) framework ................................. 16  
1.5 Contributions of Search-Based Slicing framework .......................................... 17  
1.6 Overview of the thesis structure ...................................................................... 19  

2 Literature Review ............................................................................................... 21  
2.1 Overview .......................................................................................................... 21  
2.1.1 Static slicing ................................................................................................. 22
2.1.2 Dynamic slicing ................................................. 25
2.1.3 Conditioned slicing ........................................... 26
2.2 Static Slicing Algorithm .......................................... 27
  2.2.1 Weiser’s dataflow equations ................................. 27
  2.2.2 The Program/System Dependence Graph’s algorithm .... 31
2.3 Dynamic Slicing Algorithm ....................................... 39
2.4 Conditioned Slicing Algorithm ................................... 41
2.5 Applications of Program Slicing ................................. 43
  2.5.1 Program integration ......................................... 43
  2.5.2 Testing .................................................... 46
  2.5.3 Program metrics ............................................ 48
  2.5.4 Program debugging ......................................... 53
  2.5.5 Program maintenance ........................................ 55
  2.5.6 Refactoring of the program .................................. 58
2.6 Search Based Software Engineering and Applications ........ 67
  2.6.1 Search based determination of refactoring .................. 67
  2.6.2 Search based software maintenance ......................... 68
  2.6.3 Search based amorphous slicing ............................. 69
2.7 Summary .......................................................... 70

3 The Framework of Search Based Slicing ....................... 71
  3.1 Search Space .................................................... 72
  3.2 Representation of Solutions & Slices .......................... 73
  3.3 Fitness Function ................................................ 73
  3.4 Significance of the Framework .................................. 74

4 Decomposition of Programs * .................................... 79

*This chapter is based on the work published on Journal of Information and Software Technology (IST) [66]
4.1 Problem Description .............................................. 79
  4.1.1 Fitness Function for the Decomposition ................. 80
4.2 Search Algorithms ................................................. 82
  4.2.1 Genetic Algorithm ........................................... 83
  4.2.2 Hill Climbing ............................................... 86
  4.2.3 Greedy Algorithm .......................................... 87
  4.2.4 Random Algorithm ........................................ 88
4.3 Empirical Study .................................................... 88
  4.3.1 Which algorithm is the best? ............................... 88
  4.3.2 How similar are the results for each algorithm? ....... 92
  4.3.3 Visual Evidence for the Presence for Clones .......... 102
  4.3.4 Flexibility of the Framework .............................. 105
4.4 Empirical Study with Greedy Algorithm ..................... 109
4.5 Summary ............................................................ 114

5 Analysis of Procedure Splitability * .............................. 115
  5.1 Problem Description ............................................. 115
  5.2 The Splitting Algorithm ...................................... 117
  5.3 Empirical Study .................................................. 124
    5.3.1 RQ1: How frequently are there splits? ............ 124
    5.3.2 RQ2: Splitability of Procedures ..................... 126
    5.3.3 RQ3: Exploration of correlation between procedure size and
        splitability ................................................ 133
  5.4 Threats to validity ............................................. 134
  5.5 Summary .......................................................... 135

*This chapter is based on the work published on 15th Working Conference on Reverse Engineering
(WCRE 2008) [67]
# Contents

6 Patterns of splitability of procedures 136

6.1 The results for case studies of splitable procedure patterns . . . . . . 137

6.2 Program Model and Terminology . . . . . . . . . . . . . . . . . . . . 138

6.3 Splitting Patterns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

6.3.1 Pattern 1 : Completely Independent Blocks (CIB) . . . . . . . 141

6.3.2 Pattern 2 : Independent Blocks with Shared Block (IBSB) . . 146

6.3.3 Pattern 3 : Independent Collections & Blocks (ICB) . . . . . . 149

6.3.4 Pattern 4 : Partially Dependent Collection & Block (PDCB) . 153

6.3.5 Pattern 5 : Independent Collections (IC) . . . . . . . . . . . . 155

6.3.6 Pattern 6 : Partially Shared Collection (PSC) . . . . . . . . . 157

6.3.7 Pattern 7 : Independent Collections with Shared Block (ICSB) 160

6.3.8 Pattern 8 : Partially Dependent Blocks (PDB) . . . . . . . . . 163

6.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167

7 Conclusions and Future Work 168

7.1 Program Decomposition . . . . . . . . . . . . . . . . . . . . . . . . . 168

7.2 Splitable procedures and pattern analyses . . . . . . . . . . . . . . . 170

7.3 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171

7.3.1 Application to other programming languages . . . . . . . . . . . 171

7.3.2 Automation of the process of refactoring . . . . . . . . . . . . . 172

7.3.3 Extension to multi-objective approaches . . . . . . . . . . . . . 172
List of Figures

1.1 An example of program slicing . . . . . . . . . . . . . . . . . . . . 12
2.1 Source code of Control Flow Graph . . . . . . . . . . . . . . . . . . 23
2.2 Control Flow Graph of the program in Figure 2.1 . . . . . . . . . . . 23
2.3 The PDG example corresponding to Figure 2.1 . . . . . . . . . . . . 24
2.4 An example of Weiser’s program slicing algorithm . . . . . . . . . . 29
2.5 Weiser’s inter-procedural algorithm . . . . . . . . . . . . . . . . . . 32
2.6 An example of PDG extension definition . . . . . . . . . . . . . . . . 34
2.7 The PDG extension definition . . . . . . . . . . . . . . . . . . . . . . 35
2.8 An example for the SDG. The example originally derives from the
   program used by Horwitz et al. [63]. . . . . . . . . . . . . . . . . . . . 36
2.9 SDG for the program in Figure 2.8 . . . . . . . . . . . . . . . . . . . . 37
2.10 An example of dynamic slicing: original program . . . . . . . . . . . 40
2.11 An example for dynamic slicing: the tracing of the program in Figure
   2.10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.12 An example of dynamic slicing: slice(n = 3, 9^{12}, sum) . . . . . . . 41
2.13 An example of program integrations: Base program . . . . . . . . . . 45
2.14 An example of program integrations: Variation 1 . . . . . . . . . . . 45
2.15 An example of program integrations: Variation 2 . . . . . . . . . . . 45
2.16 An example of program integrations: Integration program . . . . . . 46
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.17</td>
<td>An example of 4 types of cohesion: low level</td>
<td>49</td>
</tr>
<tr>
<td>2.18</td>
<td>An example of 4 types of cohesion: control level</td>
<td>50</td>
</tr>
<tr>
<td>2.19</td>
<td>An example of 4 types of cohesion: data level</td>
<td>50</td>
</tr>
<tr>
<td>2.20</td>
<td>An example of 4 types of cohesion: function level</td>
<td>50</td>
</tr>
<tr>
<td>2.21</td>
<td>An example of dependence clusters and pollution: original program</td>
<td>51</td>
</tr>
<tr>
<td>2.22</td>
<td>An example of dependence clusters and pollution: transformed program</td>
<td>52</td>
</tr>
<tr>
<td>2.23</td>
<td>An original program for the decomposition slicing example</td>
<td>57</td>
</tr>
<tr>
<td>2.24</td>
<td>The complement of the decomposition slicing $\text{slice}(\text{SumX})$</td>
<td>57</td>
</tr>
<tr>
<td>2.25</td>
<td>An example for refactoring the function in the program shown in Figure 2.23</td>
<td>60</td>
</tr>
<tr>
<td>2.26</td>
<td>An example of refactoring the function based upon transformation based procedure extraction: original program</td>
<td>63</td>
</tr>
<tr>
<td>2.27</td>
<td>An example of refactoring the function based upon transformation based procedure extraction: transformed programs</td>
<td>64</td>
</tr>
<tr>
<td>2.28</td>
<td>An example of refactoring the function with syntax-preserving extraction: original program</td>
<td>65</td>
</tr>
<tr>
<td>2.29</td>
<td>An example of refactoring the function with syntax-preserving extraction: transformed programs</td>
<td>65</td>
</tr>
<tr>
<td>2.30</td>
<td>An example of refactoring the function with amorphous extraction: original program</td>
<td>65</td>
</tr>
<tr>
<td>2.31</td>
<td>An example of refactoring the function with amorphous extraction: transformed programs</td>
<td>66</td>
</tr>
<tr>
<td>2.32</td>
<td>An example of amorphous slicing</td>
<td>70</td>
</tr>
<tr>
<td>3.1</td>
<td>An example of splitable procedures</td>
<td>77</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of non-splitable procedures</td>
<td>78</td>
</tr>
<tr>
<td>4.1</td>
<td>The Genetic Algorithm Used in the Study</td>
<td>83</td>
</tr>
</tbody>
</table>
4.2 The Hill Climbing Algorithm Used in the Study ........................................... 84
4.3 The Greedy Algorithm Used in the Study ...................................................... 85
4.4 The Random Algorithm Used in the Study ..................................................... 85
4.5 Box plot of results for backward slicing in term of Fitness Function 1 defined in Section 4.1.1. The results show that the greedy algorithm performs the best. The low variance for the Hill Climbing algorithm (HC) suggests either a low order of modality in the landscape or a multi-modal landscape with similar valued peaks. ........................................... 89
4.6 Box plot results for backward slicing in term of Fitness Function 2 defined in section 4.1.1. The results confirm the result from Fitness Function 1 (presented in Figure 4.5) that the greedy algorithm performs the best. The programs are presented in increasing order of size (top-to-bottom, left-to-right), providing evidence that the gap in performance between the greedy algorithm and the others increases with program size. The low variance for the Hill Climbing algorithm (HC) also replicates the finding for Fitness Function 1. ........................................... 90
4.7 Visualized results for backward slicing based on Fitness Function 1 with the program \textit{sum}. The GA produces smallest results for this very small program. ......................................................... 95
4.8 Visualized results for backward slicing based on Fitness Function 1 with the program \textit{hello}. ................................................................. 96
4.9 Visualized results for backward slicing based on Fitness Function 1 with the program \textit{informationflow}. Note that the greedy algorithm produces the best results and also achieves this with the fewest slices. 97
4.10 Visualized results for backward slicing based on Fitness Function 1 with the program \textit{acct}. The greedy algorithm produces the best results with the fewest slices. ................................................................. 98
4.11 Visualized results for backward slicing based on Fitness Function 1 with the program \textit{newton}. Note the image for the random search appears to be a ‘grainy’ version of that for the others and that the greedy algorithm result contains fewer slices. There are more similarity in the intelligent searches; random produces a ‘poor imitation’. 99

4.12 Visualized results for backward slicing based on Fitness Function 1 with the program \textit{tss}. Note the image for the random search appears to be a ‘grainy’ version of that for the others and that the greedy algorithm result contains fewer slices. 100

4.13 Clone detection to \textit{newton}. 103

4.14 Clones present in the program \textit{newton}. 104

4.15 Clone detection to \textit{tss}. 105

4.16 Clone code in \textit{tss}. 106

4.17 Four classifications of function cohesion. The A, B and C represent the processing elements; A1, A2 and A3 represent the processing elements are all in the same control block such as \textit{if}, \textit{for} or \textit{while}. 107

4.18 The results of splitting or refactoring the functions. En and Re represent subfunction entry and return value, respectively; Fa and Fb represent function calling sites to subfunctions of processing element A and B, respectively. Note that the new subfunctions in (b) all include control statements in which block A1, A2 and A3 belong. 108

4.19 Frequency distributions of percentage of decomposed functions. The X axis is the percentage of the number of the optimal set of slices to all the slices in a function; the Y axis is the number of decomposed functions which ‘drop in’ the corresponding percentage. 111

5.1 Procedure splitting algorithm. 120
5.2 An example for the splitting algorithm. The digits represent the SDG nodes of the procedure. 

5.3 An example of recovering the statement $gamma = asin(sin(alpha) * b)$ from the corresponding six SDG nodes. Solid lines and dashed lines represent control and data dependency, respectively. The SDG node type (call-site, actual-in and expression) is shown under the SDG nodes.

5.4 An example of the ‘struct’ semantic constraint. The source code is from the procedure `grphaexc` in the project `space` at the ‘GNU’ website.

5.5 Distribution of splitable procedures. A column at the horizontal position $x$ of the height $y$ means that $y/N\%$ of procedures are split into $x$ sub-procedures, where $N$ is the total number of procedures.

5.6 Overlap frequency distribution of splitable procedures that can be split into two sub-procedures. The $x$-axis measures overlap using $\sigma_{12}$ defined in (I). The $y$-axis represents the percentage of 2-way splitable procedures that lie in the corresponding overlap range.

5.7 Overlap frequency distribution of splitable procedures that can be split into three sub-procedures. The $x$-axis measures overlap using $\sigma_{123}^{min}$ defined in (II). The $y$-axis represents the percentage of 3-way splitable procedures that lie in the corresponding overlap range.

5.8 Overlap frequency distribution of splitable procedures that can be split into three sub-procedures. The $x$-axis measures overlap using $\sigma_{123}^{max}$ defined in (III). The $y$-axis represents the percentage of 3-way procedures that lie in the corresponding overlap range.

6.1 An example of program schema.

6.2 An instantiation of program schema in Figure 6.1.

6.3 Program Schema for Pattern 1 (CIB)

6.4 Sub-procedures with Program Schema for Pattern 1 (CIB)
6.5 **Pattern 1 (CIB)** of splitable procedures ........................................... 144
6.6 The procedure corresponding to **Pattern 1 (CIB)**. The chosen example is the procedure `adddef` in `space.c` of the project `space` from the `Gnu` website. .................................................. 145
6.7 Program Schema for **Pattern 2 (IBSB)** .............................................. 147
6.8 Sub-procedures with Program Schema for **Pattern 2 (IBSB)** .......... 148
6.9 **Pattern 2 (IBSB)** of splitable procedures ...................................... 149
6.10 The procedure corresponding to **Pattern 2 (IBSB)**. The chosen example is the procedure `create_file_names` in `main.c` of the project `byacc` from the `Gnu` website. ................................................. 150
6.11 Program Schema for **Pattern 3 (ICB)** ............................................ 151
6.12 Sub-procedures with Program Schema for **Pattern 3 (ICB)** ........... 152
6.13 **Pattern 3 (ICB)** of splitable procedures ...................................... 152
6.14 The procedure corresponding to **Pattern 3 (ICB)**. The chosen example is the procedure `prepare_termcap` in `tabs.c` of the project `termutils` from the `Gnu` website. ................................................. 153
6.15 **Pattern 4 (PDCB)** of splitable procedures .................................... 154
6.16 Program Schema for **Pattern 4 (PDCB)** ...................................... 154
6.17 Sub-procedures with Program Schema for **Pattern 4 (PDCB)** ...... 154
6.18 The procedure corresponding to **Pattern 4 (PDCB)**. The chosen example is the procedure `yy_include_push` in `sheets-map.c` of the project `a2ps` from the `Gnu` website. ................................................. 155
6.19 Program Schema for **Pattern 5 (IC)** .......................................... 156
6.20 Sub-procedures with Program Schema for **Pattern 5 (IC)** .......... 156
6.21 **Pattern 5 (IC)** of splitable procedures ...................................... 157
6.22 The procedure corresponding to Pattern 5 (IC). The chosen example is the procedure version in tabs.c of the project termutils from the ‘Gnu’ website. ................................................................. 157

6.23 Program Schema for Pattern 6 (PSC) ........................................ 158

6.24 Sub-procedures with Program Schema for Pattern 6 (PSC) ..... 159

6.25 Pattern 6 (PSC) of splitable procedures ................................. 159

6.26 The procedure corresponding to Pattern 6 (PSC). The chosen example is the procedure buffer_save in buffer.c of the project a2ps from the ‘Gnu’ website. ................................................................. 160

6.27 Program Schema for Pattern 7 (ICSB) ................................. 161

6.28 Sub-procedures with Program Schema for Pattern 7 (ICSB) . 161

6.29 Pattern 7 (ICSB) of splitable procedures ................................. 162

6.30 The procedure corresponding to Pattern 7 (ICSB). The chosen example is the procedure tputs in termcap.c of the project termutils from the ‘Gnu’ website. ................................................................. 162

6.31 Program Schema for Pattern 8 (PDB) ........................................ 164

6.32 Sub-procedures with Program Schema for Pattern 8 (PDB) ..... 165

6.33 Pattern 8 (PDB) of splitable procedures ................................. 165

6.34 The procedure corresponding to Pattern 8 (PDB). The chosen example is the procedure output_trailing_text in output.c of the project byacc from the ‘Gnu’ website. ................................................................. 166
## List of Tables

4.1 An example of looking for optimal properties in program slicing sets. 80
4.2 The subject programs studied. 91
4.3 Execution time of each algorithm for each program in condition of the machine—Ram 512M; Pentium4 3.2GHz. F1 and F2 represent the fitness function (1) and (2) defined in section 4.1.1, respectively; the measurement is based upon the milli second. 93
4.4 Comparison of slicing sets between the programs of the source code, $100 \cdot \frac{|(A,B) \cap |}{|Min(A,B)|}$. 101
4.5 Program descriptions. 110
4.6 Percentage and execution time. Percentage is the ratio between the optimal set of slices to the set of all the slices of the whole program; execution time measured in seconds was obtained from execution of decomposing all the functions on a machine, Pentium4 3.2GHz with Ram 512M. 112
4.7 Percentage of redundancy slices. 113
5.1 The backward slices of the splitable procedure in Figure 5.2. A 1 represents a SDG node that is included in the slice, while a 0 represents a SDG node that is not included in the slice. 121
5.2 The set of programs studied. Entry ‘Loc’ represents the lines of code; ‘Descriptions’ for the functionality of programs; ‘Ver’ for the number of SDG nodes corresponding to source code; ‘No of Procs’ for the total number of procedures in the entire program; ‘Split procs’ represents how many procedures in the program can be split into two or more; ‘Count’ for the number of procedures which can be split and ‘Perc’ for the percentage of ‘Count’ to ‘No of Procs’. 124

5.3 Correlation between procedure size and splitability of procedures where splitable procedures can be split into two sub-procedures in terms of $S$ defined in (IV). 131

5.4 Correlation between procedure size and splitability of procedures where splitable procedures can be split into three sub-procedures in terms of $S_{max}$ defined in (V). 132

6.1 Distribution of patterns with a set of splitable procedures. The definitions of the eight patterns are described in the following sections. 137

6.2 Distribution of patterns with a set of splitable procedures. The entry ‘No. of sub’ represents the number of sub-procedures in the corresponding pattern; ‘Percentage’ represents the number that can be split into the number of corresponding sub-procedures to the total number of procedures in the pattern. 138
Chapter 1

Introduction

Software engineering has played a very important role in developing systematic, qual-
ifiable and maintainable software [42]. Its purpose is to improve the reliability, un-
derstandability and maintainability of software.

In software engineering, the maintenance process is that of modifying software af-
fter development in order to improve performance or obtain extra required attributes,
or to adapt to new system environments. As software becomes more and more com-
plex, the cost of software maintenance becomes more expensive. Thus, it is essential
to use efficient techniques and tools to assist the process of software maintenance.

Program slicing has been applied to many aspects of maintenance of software
systems, from program testing [41] and integration [62] to software metrics [101] and
refactoring [48, 82].

This section mainly focuses on the introduction of program slicing including its
power and issue to program dependency analysis. Therefore, the combination of
slicing and search-based software engineering is motivated for helping dependency
analysis. Finally, the section overviews the thesis structure.
1.1 Program slicing

1.1.1 Concept

Program slicing is an approach used to extract a subset of a program according to a set of variables in a specific statement as a slicing criterion. Mark Weiser originally defined backward slicing [115] and used data flow and control flow analyses for computing intra/inter procedural backward slices.

Algorithms for program slicing have been developed for decades from static slicing [63, 115] to dynamic slicing [2, 76] and conditioned slicing [36] and applied to many aspects of software engineering [16, 41, 62, 87, 101, 108].

As an illustrative example, consider the popular program \textit{Sum,Product} [112]. Figure 1.1 shows the original program and a slice of this program with respect to slicing criterion (11, product). As can be seen in the figure, all computations not relevant to variable \textit{product} have been ‘sliced off’.

1.1.2 Slicing for program dependency

Program slicing is considered as a technique for extracting a subprogram from an entire program in terms of the variables in a specific statement. A subset of statements in a program on which the variables of interest depend simplifies the program for debugging, testing and maintenance [50, 56, 87]. All applications of program slicing lie in the character of dependency, that is, slicing is referred to as a method of dependence analysis.

Due to the nature of program slicing, maintainers can use it to ease program dependence analysis where large and complicated programs are error-prone and tedious to debug. Dependence analysis has been applied to several stages of the software engineering process, such as program restructuring [29, 82], program comprehension [38], regression testing [17] and program integration [62]. It can also be an effec-
// Original program
1 read(n);
2 i : = 1;
3 sum : = 0;
4 product : = 1;
5 while i <= n do
6   sum : = sum + i;
7   product : = product * i;
8   i : = i + 1;
9 end
10 write(sum);
11 write(product);

// slice(11,product)
12 read(n);
13 i : = 1;
14 product : = 1;
15 while i <= n do
16   product : = product * i;
17   i : = i + 1;
18 end
19 write(product);

Figure 1.1: An example of program slicing
tive way of understanding the dependency structure of a program [16, 19, 81] and of measuring dependence-related attributes such as cohesion and coupling [14, 92]. For these applications, sets of slices are used to reveal properties of interest in the program under analysis, such as the presence of dependence clusters and the cohesive (and less cohesive) parts of the program.

1.1.3 The issue of slicing-based dependency analyses

Since every statement and variable can be sliced, combinations of these slices might represent dependency structures of interest (e.g. program decomposition and clones) in programs. However, as far as the dependency structure of programs is concerned, few investigations are considered. This idea motivates the thesis of the Search Based Slicing framework for the finding of these kinds of dependency structures in programs.

The advent of commercial, scalable and robust tools for slicing such as Grammatech’s Codesurfer [54] makes it possible to construct all possible slices for large programs in reasonable time. The Codesurfer scripting language based on Scheme-STK [54] is used to process program code represented as a System Dependence Graph SDG [63] and to calculate the slices. Slicing is performed on every SDG node, therefore, for a program of $n$ nodes, the possible solutions consists of $2^n - 1$ subsets of slices.

By constructing the set of all slices of a program, it is possible to analyse the dependency structure of the program. This allows slicing to be used to capture the dependence of every point in the program, allowing analysis of the whole program dependence structure. This raises an interesting research question:

“How can useful dependency structures of interest be formed in amongst the mass of dependence information available?”

In the thesis, dependence is analysed using program slicing, and so this question
is reformulated as the hypothesis of this thesis:

“Of the set of all possible slices of a program, which subsets reveal depen-
dency structures of interest?”

Of course, for a program consisting of $n$ program points, there will be $n$ possible
slices and, therefore, $2^n - 1$ subsets of slices. Since the number of program points is
always at least as large as the number of statements in the program, the powerset of all
possible slices will be extremely large; too large to enumerate for any realistically sized
program. This is merely a reflection of the mass of dependence information available
and that would need to be considered by any whole program dependence analysis.
The overwhelming quantity of information creates the problem of dependence analysis
considered in this thesis and motivates the search based approach introduced in this
thesis to find out the optimal subset.

1.2 Search-Based Software Engineering (SBSE)

Search-Based Software Engineering is a general framework that applies metaheuristic
techniques such as genetic algorithms, simulated annealing and tabu search to soft-
ware engineering. The key issue is to re-construct software engineering problems into
search based problems. Thus, three aspects need to be formulated in terms of the
specific problem tackled.

1.2.1 Representation

To accomplish the goal of re-constructing software engineering problems into search
based problems, a proper representation of solutions is essential for the transformation
between classic and search-based solutions. In general, there are typical solution
representations such as binary code and floating point numbers [9].
1.2.2 Fitness

A fitness function is referred to as a ‘guide’ to search-based techniques for finding the optimal solution. In general, a fitness function is a mathematical expression that leads search-based methods developed to seek maximum or minimum values.

1.2.3 Search-based approaches

After having representation and fitness, final optimum is achieved with search techniques using different operations. Which search approach should be used depends on specific problems and definition of fitness both of which mainly affect the performance of different search techniques. In other words, problem specifications and fitness determine the landscape of distribution of solutions.

1.3 Motivation of Search-Based Slicing

This thesis introduced an approach to locating dependence structures, based on the principles of SBSE [32, 60]. Using this formulation, the problem becomes one of a search for a set of slices that exhibits dependency structures of interest. The choice of what constitutes a ‘dependency structure of interest’ is a parameter to the overall approach, making it highly flexible. In SBSE, a fitness function is defined to capture such a property of interest. In the case of search-based slicing, it captures the properties of a dependency structure that make it interesting to a particular analysis.

The search process is realized by an algorithm that uses a fitness function to guide a search for finding optimal or near optimal solutions. In order to experiment with the search-based slicing approach, the thesis presented the results of an empirical study on the search for slice sets that cover the program with minimal overlap. The fitness function used in the empirical study is motivated by work on slicing as a decomposition technique [51, 113].
This instantiation of the search-based slicing approach formulated the decomposition problem as a set cover problem [46]. However, it must be stressed that this represents merely an *instantiation* of a parameter to the approach (the fitness function). The search based slicing approach derives a great deal of flexibility from the fact that the fitness function (and therefore the property of interest) is merely a parameter; in order to search for a different kind of dependence structure, only the fitness function needs to be changed.

### 1.4 Objectives of the Search-Based Slicing (SBS) framework

The objective of the SBS framework is to investigate the possibility of program dependency structures using the SBS framework and to explore the flexibility of the framework to different applications.

The objectives encompass the following aspects:

1. To re-construct the source code analysis problem as a search-based problem.

2. To formulate an appropriate representation for encoding an individual solution for manipulation with search-based techniques.

3. To compare the effects of different search techniques for different problems.

4. To justify that the search-based slicing for dependency structure analysis is applicable and flexible.

5. To explore applications based upon the search-based slicing framework.
1.5 Contributions of Search-Based Slicing framework

The contributions of the thesis can be divided into several aspects as follows:

Program Decomposition:

1. An approach that identifies dependency structures is introduced as a search problem within the powerset of the set of all possible program slices. This approach allows search-based algorithms to be used to search for dependency structures of interest.

2. A fitness function is introduced that seeks to optimise the search towards solutions in which the program is decomposed into a set of slices that collectively covers the whole program with minimal overlap. Four search algorithms are implemented in order to experiment with this fitness function.

3. The results of an empirical study are reported, showing that the greedy algorithm performs better than the random, hill climbing and genetic algorithm approaches to the problem. This is an attractive finding, since greedy algorithms are extremely simple and efficient.

4. A simple visualization is introduced to explore the results and their similarity. This shows a higher degree of similarity in the results of the ‘intelligent’ techniques than in those of the random search. This visual impression is augmented by computational analysis of results. The similarity of results for intelligent searches shows that the results are consistent and meaningful.

5. The visualization also has an interesting side effect, which may be a useful spin off: the presence of code clones becomes visually striking in some of the examples.
6. The study also reports results on redundancy – that is, how often a slice is completely included in another one. The results suggest that the redundancy phenomenon is universal. However, it is shown that this redundancy does not affect the greedy algorithm advocated in the study.

7. Based upon the performance comparison of the four search algorithms, the greedy algorithm is further applied to six larger programs to decompose each procedure of each program. The results show that the majority of procedures can be decomposed into sets of slices efficiently.

**The splitting procedure approach:**

1. An approach is introduced to split a procedure into a set of sub-procedures. The approach provides uses in analysing and potentially improving procedure structures by splitting procedures.

2. The empirical study shows that a surprisingly large proportion of the procedures considered are splitable.

3. The experiment provides evidence for a strong statistical correlation between procedure size and splitability.

**Patterns analysis of splitable procedures:**

1. The empirical study verifies the validity of the splitting algorithm by looking into the 194 splitable procedures and generalising the categories of attributes that make procedures splitable.

2. The investigation summarises 8 patterns among the 194 ‘typical’ splitable procedures. Each pattern represents a classification of procedures that can be split in a specific way.
1.6 Overview of the thesis structure

The rest of the thesis is organized as follows:

- **Chapter 2, Literature Review** surveys the previous work on program slicing, focusing on its algorithms and applications. In addition, the Search-Based Software Engineering (SBSE) framework and its universal applications in software engineering are also reviewed.

- **Chapter 3, The Framework of Search Based Slicing** introduces the search-based slicing framework by defining its search space, representation of solution and fitness function. The significance of the search-based slicing framework is highlighted.

- **Chapter 4, Decomposition of Programs** applies the framework to program decomposition. The idea is to partition the program into a set of slices that can cover the program with minimal overlap amongst the slices.

- **Chapter 5, Analysis of Procedure Splitability** introduces another application of search-based slicing framework. From the cohesion point of view, the thesis defines splitability of procedures and describes how procedures can be split into a set of sub-procedures in terms of ‘multi-task’ attributes of procedures. The thesis considers procedure splitting as searching for independent sub-modules of procedures. This chapter also includes an empirical study based on the splitting algorithm.

- **Chapter 6, Patterns of Splitability of Procedures** provides evidence that splitable procedures are split according to structural patterns. The aim of pattern analyses is to testify the splitting algorithm and to find the patterns that make procedures splitable.
• Chapter 7 **Conclusion and Future Work** concludes the thesis and provides future work. All the results and contributions are summarised and possible future research directions in this area are suggested.
Chapter 2

Literature Review

The research of program slicing mainly focuses on two aspects. One is the research of slicing algorithms that are divided into three classifications in terms of static slicing, dynamic slicing and conditioned slicing. The other is applications of program slicing. Program slicing has the power of applying dependence analysis to program analysis. This chapter reviews the literature of program slicing algorithms and its applications including debugging, testing, integration, metrics, maintenance and refactoring.

2.1 Overview

Program slicing was introduced by Weiser’s PhD thesis in 1979 [115]. Weiser claims that a slice corresponds to the mental abstractions that programmers make when they are debugging a program, and advocates the integration of program slicers in debugging environments. A variety of notions of program slices have since been proposed, as well as a number of methods to compute slices. The main reason for this variation is that different applications require the different properties of slices. Weiser defines a program slice as a reduced, executable program obtained from the program by removing statements, such that the slice replicates the part of the behavior of the program. Another definition of a slice is a subset of the assignments and control
predicates of a program that directly or indirectly affect the values computed at a criterion, but this subset does not necessarily constitute an executable program.

The section briefly reviews the variety of slicing formulations: static, dynamic, and conditioned. There is an important distinction between a static and a dynamic slice, that is, the former is computed without making assumptions regarding a specific program’s input, whereas the latter relies on some specific input. Conditioned slicing is a theoretical bridge between the two extremes of static and dynamic slicing [36]. The sections below will introduce the each of them in detail as well as the algorithms for computing them.

2.1.1 Static slicing

Weiser’s original form of program slicing is static and backward. A static slice is computed with respect to a slicing criterion that consists of a program location number and a set of variables. Program slices originally are referred to as backward static slices in which the direction of control flow graph are traversed back and they are computed as the solution to a static analysis problem without depending on any specific program’s input.

Weiser’s original slicing algorithm is based on data-flow analyses. In his approach, slices are computed by computing sets of relevant statements backwards, according to data flow and control flow dependences. Only static information is used to compute slices; thus, this type of slice is referred to as a static slice. An alternative algorithm, first introduced by Ottenstein and Ottenstein [103], takes backward slices using the Program Dependence Graph (PDG) by traversing the dependence edges backwards. The method of computing static slices transforms the problem of static slicing into a reach ability problem within the representation of a PDG [47].

A PDG is a graph with nodes (vertices) based on the Control Flow Graph (CFG), which correspond to assignment and control predicates, and edges which correspond
procedure SumandProduct( P : integer; SumN, Total : integer )
1     I : integer;
2     SumN = 0;
3     Total = 0;
4 for I := 1 to P do
5         SumN = SumN + I;
6         Total = SumN * I;
7         SumN = I * I;
8         SumN = SumN * SumN;
9 end
10 write(SumN);
11
Figure 2.1: Source code of Control Flow Graph

Figure 2.2: Control Flow Graph of the program in Figure 2.1
Figure 2.3: The PDG example corresponding to Figure 2.1
to data and control dependences. For example, Figure 2.2 and Figure 2.3 show the
CFG and PDG corresponding to the program in Figure 2.1, respectively.

A slicing criterion is identified with a node (vertex) in the PDG, and a slice corre-
sponds to all PDG nodes (vertices) from which the node (vertex) under consideration
can be reached. Various program slicing approaches discussed below utilise modi-

The slices mentioned above are computed by gathering assignments and control
predicates by backward traversal of the program’s Control Flow Graph/Program De-
pendence Graph CFG/PDG, starting at a slicing criterion. Horwitz, et al. [105] put
forward the notion of forward slicing which is taken by computing what statements
are affected by the value of a variable at a statement. Informally speaking, a for-
ward slice consists of all assignments and control predicates dependent on a slicing
criterion. A statement is dependent on the slicing criterion if the values computed at
that statement depend on the values computed at the slicing criterion. Backward and
forward slices are computed in a similar way; the latter requires tracing dependences
forwards.

2.1.2 Dynamic slicing

Korel and Laski introduced the notion of dynamic slicing [76] – a slice is computed for
a particular input at the running time. Dynamic slices are smaller than static slices
due to the specific input added to slicing criteria. Similar to the Weiser’s algorithm,
Korel and Laski’s algorithm uses a control-flow graph as a program representation.
Agrawal and Horgan [2] later present a dynamic slicing algorithm that uses the pro-
gram dependence graph as a program representation.

In dynamic program slicing, only the dependence that is in a specific execution of
a program is taken into account. A dynamic slicing criterion specifies the input, and distinguishes between different occurrences of a statement in the execution history; typically, it consists of a triple (input, occurrence of a statement, variables). In general, the difference between static and dynamic slicing is that latter assumes a specific input for a program, whereas former does not make any assumption about any specific inputs.

2.1.3 Conditioned slicing

Conditioned slicing extends the traditional slicing criterion with a condition (called program conditioning) that captures a set of initial program states. Program conditioning consists of removing a set of statements which cannot be reached when a condition holds at some point in a program. Conditioned slicing has been applied to problems in maintenance, software testing, reusability and re-engineering [50, 56, 62, 89]. All current approaches to program conditioning rely on symbolic execution and reasoning (or theorem prover) about symbolic predicates. The reasoning can be performed ‘heavily’, which may impose unrealistic performance constraints.

This additional condition may simplify a program before applying a traditional static slicing algorithm. Such pre-simplification is called conditioning and it is achieved by eliminating statements which do not contribute to the computation of the variables with a condition state. Techniques for constructing conditioned slices are still at the early stages of development; results concerning the nature of conditioned slices and their applications are more limited than those available for the static and dynamic slicing and therefore have been included simply in this review.
2.2 Static Slicing Algorithm

In this section, based on the literature of the fields of program slicing, several algorithms of program slicing will be introduced with respect to different intermediate representations for both intra- and inter-procedures. These algorithms essentially compute the same information in different ways.

2.2.1 Weiser’s dataflow equations

Intra-procedural slicing

Weiser’s original definition of program slicing is based on iterative solution of dataflow equations [115]. A slicing criterion consists of a pair \((n, V)\) where \(n\) is a node in the Control Flow Graph (CFG) of a program, and \(V\) a subset of program variables in the node \(n\).

Finding slices using dataflow analysis begins with tracing possible influences of the variables backwards. Slicing, in effect, is to know which statements can affect the variables observed through a slicing criterion. In general, for each statement in the program, there will be some set of variables whose values can affect the variables as the slicing criterion through a chain of statements. For instance, if the statement ‘\(c := b\)’ is after the statement ‘\(b := a\)’, then the value of \(a\) before the statement ‘\(b := a\)’ can affect the value of \(c\) after the statement ‘\(c := b\)’.

In the paper [115], Weiser introduced dataflow equations: \(R[0, C](n)\) is the set of variables at the statement \(n\) whose values can directly (without control dependency) affect what is observed from the criterion \(C\). The zero here means the direct effect. Thus, different levels of indirect control dependency can be expressed as: \(R[1, C]\), \(R[2, C]\)... which are referred to as increasing levels of indirect effect.

To begin with, Weiser’s algorithm considers an intra-procedure, which describes an iterative algorithm for computing backward slices and can be summarized as two
phases of iteration as follows:

1. Tracing back transitive ‘zero level’ data dependency,

2. Tracing back the different levels of indirect control dependency step by step.

For each such predicate, phase 1 is repeated to include the statements on which the slice is dependent.

The formal definition of $R[0, C](i)=\text{all variables } v$ is formulated by Weiser as follows:

1. if the statement $i$ is $n$ itself, $v$ is in $V$,

2. if $i$ is a predecessor of a node $n_0$, either: a), $v$ is in $\text{REF}(i)$ and there is a $v_0$ in both $\text{DEF}(i)$ and $R[0, C](n_0)$; b), $v$ is not in $\text{DEF}(n_0)$ and $v$ is in $R[0, C](n_0)$.

In this definition, variables altered are referred to as $\text{DEF}$; variables referenced as $\text{REF}$; $\text{DEF}(n)$ is a set of variables that are modified at node $n$. $\text{REF}(n)$ is a set of variables that are used at node $n$. Thus, the relationship between variables and variables is that, if there are two variables $a$ and $b$, for $a$ in $\text{DEF}(n)$ and $b$ in $\text{REF}(n)$ for the statement $n$, $a$ depends on $b$. Also, $S[0, C]$ represents the statements included in the slice by $R[0, C]$; $S[0, C]$ is defined as: $S[0, C]$ is all nodes $n$ where $R[0, C](n)$ intersecting $\text{DEF}(n)$ is non-empty.

As mentioned above, the second step of iteration in Weiser’s algorithm consists of taking control dependency into account. Variables used in the control predicates, such as an $\text{if}$ or $\text{while}$ statement, are referred to as a new criterion for the indirect level of dependency. The sets $R[i + 1, C]$ and $S[i + 1, C]$ are increasing subsets of the programs variables and statements, respectively; that is, $R[i + 1, C]$ is union of $R[i, C]$ and the statement in $B[i, C]$ which are indirectly relevant due to the control influence on nodes in $S[i, C]$. Similarity, $S[i + 1, C]$ is all the nodes $n$ in which $\text{DEF}(n)$ intersecting $R[i + 1, C](n)$ is non-empty. In summary:
1. \( R[C] = \cup R[i, C] \).

2. \( S[C] = \cup S[i, C] \).

As an example, consider slicing the program of Figure 2.4 with respect to criterion \((\text{SumN,10})\). From the information in the figure, and the definition of a slice, we obtain \( S[0, C] = \{1, 2, 7, 8, 10\} \), \( B[0, C] = \{4\} \) and \( S[1, C] = \{1, 2, 4, 7, 8, 10\} \). For the example, indirectly relevant variables is reached at set \( S[1, C] \) because there is only one level of control dependence. The corresponding slice criterion \( C = (\text{SumN,10}) \) as computed by Weiser’s algorithm is identical to the program shown in the second part in Figure 2.4.

```plaintext
/* Original program */
I : integer;
SumN := 0;
Product := 0;
for I := 1 to P do
   SumN = sumN + I;
   Product = SumN * I;
   SumN = I * I;
   SumN = SumN * sumN;
end
write(SumN, Product);

/* slice(SumN,10) */
I : integer;
SumN = 0;
for I := 1 to P do
   SumN = I * I;
   SumN = SumN * sumN;
end
write(SumN, Product);
```

Figure 2.4: An example of Weiser’s program slicing algorithm

**Inter-procedural slicing**

For inter-procedural slicing, it is important to understand the distinction between two different but related slicing problems described by Horwitz et al. [63]:

29
1. “The slice of a program with respect to program point $p$ and variable $x$ consists of all statements and predicates of the program that might affect the value of $x$ at point $p$.

2. The slice of a program with respect to program point $p$ and variable $x$ consists of a reduced program that computes the same sequence of values for $x$ at $p$. That is, at point $p$ the behavior of the reduced program with respect to variable $x$ is indistinguishable from that of the original program.”

For intra-procedural slicing, a solution to Version 1 is the same as that to Version 2. However, for inter-procedural slicing, using the same method to Version 1 for Version 2 just to reduce the statements and predicates can cause a program slice to be syntactically incorrect. The reason is because of multiple calls to the same procedure [106]. Weiser’s algorithm produces a solution to Version 2 even though it doesn’t consider the ‘context-call’ problem addressed by Horwitz et al. [63] and thus it is imprecise.

Weiser’s inter-procedural algorithm extends his intra-procedural dataflow algorithm. Essentially, inter-procedural slicing algorithm iteratively generates new slicing criteria with respect to which intra-procedural slices are computed. For each procedure $A$ called by procedure $P$ where there is the original slicing criterion $C$, the new criterion in $A$ are generated by: (i) the statement number is the final statement in $A$; (ii) actual parameters in the call are substituted with formal parameters in the procedure $A$. In the paper, the algorithm is referred to as $\text{DOWN}(C)$. For each procedure $B$ calling procedure $P$, the new criterion in $B$ are defined by: (i) the statement in $B$ is that of the call statement. (ii) the variables are in the initial statement in $P$, local variables are removed and formal parameters are substituted with actual parameters. The algorithm is referred to as $\text{UP}(C)$.

This process continues until there is no longer calling or called procedures. Thus, an inter-procedural slice is the union of the intra-procedural slices for each criterion.
Horwitz et al. point out that “the chief difficulty in inter-procedural slicing is correctly accounting for the ‘calling-context’ of a called procedure”. The System Dependence Graph (SDG) \cite{64} provides the solution with use of an attribute grammar to represent calling and parameter-linkage relationships among procedures. The SDG includes the data dependence edges that represent ‘summary dependencies’ due to procedure calls, and the edges for direct dependency in intra-procedures.

Figure 2.5 shows a program that exhibits the calling-context problem. For example, assume that a slice is to be computed using the criterion \((12, \text{total})\). Using summary information to approximate the effect of the calls, the initial approximation of the slice will consist of the entire main procedure except lines 4, 7 and 11. As the initial criterion is in the main program, we have that \(\text{UP}(\{(12, \text{total})\}) = \emptyset\), and that \(\text{Down}(\{(12, \text{total})\})\) contains the criteria \((22, \{P, Q\})\) will be the inclusion of these procedures in their entirety. Note that the calls to \textit{Append} at lines 19 and 20 cause the generation of a new criterion \((14, \{M, N\})\) and thus re-slicing of procedure \textit{Append}. It can now be seen that the example program exhibits the ‘calling context’ problem: since line 20 is in the slice, new criteria are generated for all calls to \textit{Append}. These calls include the (already included) calls at lines 9, 19, and 20, but also the call \textit{Append}(\textit{sumN}, I) at line 7. The new criterion \((7, \{\text{sumN}, I\})\) that is generated will cause the inclusion of lines 7 and 4 in the slice. Consequently, the slice consists of the entire program except for the statement 11.

\subsection{The Program/System Dependence Graph’s algorithm}

\textbf{Ottenstein and Ottenstein’s algorithm—intra-procedural}

It is essential to generate a program representation in order to capture data and control dependence information. An extended program dependence graph \cite{103} represents control dependence as well as data dependence, but can only represent ‘struc-
program main()
input(N);
I := 1;
sumN := 0;
total := 1;
for (; I <= N; I + +) do
    Append(sumN, I);
    Square(total, I);
    Append(I, 1);
end
output(sumN);
output(total);

procedure Append(M; N)
M := M + N;

procedure Square(P; Q)
I := 1;
J := 0;
while I <= P do
    Append(J, P);
    Append(I, N);
end
P := J;

Figure 2.5: Weiser’s inter-procedural algorithm
tured’ programs.

Data dependence and control dependence are defined in terms of the Control Flow Graph (CFG) of a program. A CFG contains a node for each statement and control predicate in a program; an edge from node $i$ to node $j$ indicates the possible flow of control from the former to the latter.

Figures 2.1 and 2.2 give an example and the corresponding CFG, respectively. Figure 2.3 gives the dependence graph for the same program. ‘Entry’ is the condition for execution of the program, and is a distinguished node in every PDG. The solid edges represent control dependence and dashed edges represent data dependence.

The computation of slices is based on data dependence and control dependence. Originally, Ferrante et al. [47] introduced an approach going through the graph backwards, but forwards. However, computing program slicing can reverse the direction of the pointers in a PDG.

The paper [47] was the first of many to define slicing as a reach-ability problem in a dependence graph representation of a program. They used the PDG for static slicing of intra-procedure programs. In dependence graph based approaches, a slicing criterion is identified with a vertex $v$ in the PDG. In Weiser’s terminology, this corresponds to a criterion.

**Horwitz, Reps, and Binkley’s algorithm—inter-procedural**

**System Dependence Graph (SDG)** An SDG contains a program dependence graph for the main program, and a procedure dependence graph for each procedure [63]. The vertices of the program dependence graph for programs represent assignments and control predicates that occur in the program. Moreover, there is a distinguished vertex called the entry vertex representing the program entry. Furthermore, the PDG defined by Horwitz, Reps, and Binkley include two other categories of vertices [63]: 
1. “For each variable $x$ for which there is a path in the standard control-flow graph for $P$ on which $x$ is used before being defined, there is a vertex called the initial definition of $x$. This vertex represents an assignment to $x$ from the initial state. The vertex is labeled $X := \text{InitialState}(x)$.

2. For each variable $x$ named in $P$’s end statement, there is a vertex called the final use of $X$. It represents an access to the final value of $x$ computed by $P$, and is labeled $\text{FinalUse}(x)$.”

For example, Figure 2.7 shows a PDG corresponding to a program in Figure 2.6.

```plaintext
program sumcomputing(SumN, I : integer)
1  SumN := 2;
2  I := 0;
3  while $I < 100$ do
4      SumN := SumN * SumN;
5      I := I + 1;
6  end
```

Figure 2.6: An example of PDG extension definition

There are several types of vertices and edges in SDGs that do not occur in PDGs defined in [63].

1. There is a call-site vertex in the SDG for each call statement,

2. Actual-in and actual-out vertices that model the copying of actual parameters to/from temporary variables.

3. Each procedure dependence graph has an entry vertex, and formal-in and formal-out vertices to model copying of formal parameters to/from temporary variables.

Actual-in and actual-out vertices are control dependent on the call-site vertex;
formal-in and formal-out vertices are control dependent on the procedures entry vertex.

34
In addition to these intra-procedural dependence edges, an SDG contains the following inter-procedural dependence edges:

1. A control dependence edge between a call-site vertex and the entry vertex of the corresponding procedure dependence graph.

2. A parameter-in edge between corresponding actual-in and formal-in vertices.

3. A parameter out edge between corresponding formal-out and actual-out vertices.

4. Summary edges that represent transitive inter-procedural data dependences.

Figure 2.9 shows the SDG for the program of Figure 2.8. In the figure, solid arrows represent control dependency, and dashed arrows represent inter-procedural flow dependency and intra-procedural data dependency. A system dependence graph is formed by connecting procedure dependence graphs, that is, involving the addition of four new kinds of edges:
Program Main

1. Program Main
2. SumN := 0;
3. I := 0;
4. while I < 100 do
5.    call Foo1(SumN, I);
6. end
7. output(SumN, I);

Procedure Foo1(a, b)
8. call Foo2(a, b);
9. call Foo3(b);

Procedure Foo2(c, d)
10. c := c * d;

Procedure Foo3(e)
11. call Foo1(e, 1);

Figure 2.8: An example for the SDG. The example originally derives from the program used by Horwitz et al. [63].

1. A call edge is added from each call-site vertex to the related procedure-entry vertex.

2. A parameter-in edge is added from each actual-in vertex at a call site to the related formal-in vertex in the called procedure.

3. A parameter-out edge is added from each formal-out vertex in the called procedure to the related actual-out vertex at the call site.

4. A summary edge is added due to procedure calls.

inter-procedural algorithm  Horwitz, Reps, and Binkley [63] presented an algorithm for computing precise inter-procedural static slices, which consists of the following three components:

1. The System Dependence Graph (SDG), a graph representation for multi-procedure programs.
Figure 2.9: SDG for the program in Figure 2.8
2. The computation of inter-procedural summary information. This forms precise
dependence relations between the input and output parameters of each proce-
dure call, and is represented in the SDG in the form of summary edges.

3. A two-pass algorithm for extracting inter-procedural slices from an SDG.

For the original inter-procedural algorithm, Weiser’s method does not produce as
precise a slice as possible because the ‘transitive-closure’ operation fails to account for
the ‘calling context’ of a called procedure originally posed by Horwitz et al. For this
problem, Horwitz et al. introduced a two-pass algorithm with system dependence
graphs including some extra data dependence edges that represent transitive depen-
dences due to the effects of procedure calls. These edges are constructed with the aid
of an auxiliary structure that represents calling and parameter-linkage relationships.

Their algorithm introduced how procedure calls and procedure entry are repre-
sented in procedure dependence graphs and how edges representing dependences be-
tween a call site and the called procedure are added to connect these graphs together.
Transitive dependences due to procedure calls are computed using the linkage gram-
mar and are added as the final step of building a System Dependence Graph (SDG).
The inter-procedural algorithm is summarized by Horwitz et al. as follows [63]:

**Phase 1** identifies vertices that can reach the statement $s$, and are either in $P$ itself or in
a procedure that calls $P$ (either directly or transitively). Because parameter-out
edges are not followed, the traversal in Phase 1 does not descend into procedures
called by $P$. The effects of such procedures are not ignored, however; the pres-
ence of transitive flow dependence edges from actual-in to actual-out vertices
(subordinate-characteristic-graph edges) permits the discovery of vertices that
can reach $s$ only through a procedure call, although the graph traversal does
not actually descend into the called procedure.

**Phase 2** identifies vertices that can reach the statement $s$ from procedures (transitively)
called by \( P \) or from procedures called by procedures that (transitively) call \( P \). Because call edges and parameter-in edges are not followed, the traversal in Phase 2 does not ascend into calling procedures; the transitive flow dependence edges from actual-in to actual-out vertices make such ascents unnecessary.”

Due to the algorithm that does not indeed ascend/descend to calling/called procedures, the transitive-closure operation is avoided that leads to fail the consideration of call-context problems

### 2.3 Dynamic Slicing Algorithm

In dynamic slicing, there are also two types of methodology for intra-procedures and inter-procedures. This section will simply introduce the concept.

Korel and Laski described algorithms of dynamic slices as a trace consisting of a sequence of occurrences of statements and control predicates \([76]\). A dynamic slicing criterion is specified as a triple \((x, I^q, V)\) in where \(x\) denotes the specific input of the program, occurrence \(I^q\) is the \(q^{th}\) position of the trace of statement \(I\), and \(V\) is a subset of variables of the program.

Korel and Laski introduced three dynamic flow concepts formalising the dependency between occurrences of statements in a trace in order to compute dynamic slices. The Definition-Use \(DU\) relation represents a use of a variable with its last modification. The Test-Control \(TC\) relation represents the most recent occurrence of a control predicate with the statement occurrences in the trace that are control dependent on it. Occurrences of the same statement are relevant in the symmetric Identity Relation \(IR\).

For a slicing criterion \((x, I^q, V)\), the initial approximation \(S^0\) contains the last definitions of the variables in \(V\) in the trace before statement instance \(I^q\). Approximation \(S^{i+1}\) is defined as follows:
1 read(sum);
2 I := 1;
3 while I < N do
4   if I < 10 then
5     sum := sum + 1;
6   else sum := sum * 2;
7     I++;
8 end
9 write(sum);

Figure 2.10: An example of dynamic slicing: original program

$\text{1}$ read(sum);
$\text{2}$ I := 1;
$\text{3}$ 1 < 3;
$\text{4}$ if 1 < 10;
\hspace{1em}$\text{5}$ sum := sum + 1;
\hspace{1em}$\text{6}$ I++;
$\text{7}$ 2 < 3;
$\text{8}$ if 2 < 10;
\hspace{1em}$\text{9}$ sum := sum + 1;
\hspace{1em}$\text{10}$ I++;
$\text{11}$ 3 < 3;
$\text{12}$ write(x);

Figure 2.11: An example for dynamic slicing: the tracing of the program in Figure 2.10
\[ S^{i+1} = S^i \cup A^{i+1} \], where \( A^{i+1} \) is defined:
\[
A^{i+1} = \{ X^p | X^p \notin S^i, (X^p, Y^t) \in (DU \cup TC \cup IR), Y^t \in S^i, p, t < q \}
\]

Therefore, the dynamic slice is easily obtained from the \( S(C) \) of this process: any statement \( X \) for which an instance \( X^p \) occurs in trace will be in the slice.

For example, consider the program in Figure 2.10 and criterion \( (N = 3, 9^{12}, \text{sum}) \). Figure 2.11 shows the trace of the original program. On the basis of the definition, we can gain: \( DU = \{(1^1, 5^5), (2^2, 3^3), (2^4, 4^4), (2^2, 7^6), (5^5, 5^9), (7^6, 3^7), (7^6, 4^8), (7^6, 7^{10}), (5^9, 9^{12}), (5^9, 3^{11})\} \), \( TC = \{(3^1, 4^4), (3^5, 5^5), (3^3, 7^6), (4^4, 5^5), (3^7, 4^8), (3^7, 5^9), (3^7, 7^{10}), (4^8, 5^9)\} \), \( IR = \{(3^3, 3^7), (3^7, 3^3), (3^3, 3^{11}), (3^{11}, 3^3), (3^7, 3^7), (3^{11}, 3^7), (4^4, 4^{10}), (4^8, 4^4), (7^6, 7^{10}), (7^{10}, 7^6)\} \); therefore, \( S^0 = \{5^9\} \), \( A^1 = \{3^7, 4^8, 5^5\} \), \( A^2 = \{1^1, 3^3, 4^4, 7^6, 7^{10}, 3^{11}, 4^{14}\} \) and \( A^3 = \{2^2\} \); consequently, \( S^C = \{1^1, 2^2, 3^3, 3^7, 3^{11}, 4^4, 4^8, 5^5, 7^6, 5^9, 7^{10}, 9^{12}\} \). The dynamic slice with respect to criterion \( \{N = 3, 9^{12}, \text{sum}\} \) includes every statement except statement 6. Thus, this slice was shown in Figure 2.12.

1. \texttt{read(sum);}
2. \( I := 1; \)
3. \texttt{while I < N do}
4. \hspace{1em} \texttt{if I < 10 then}
5. \hspace{2em} \texttt{sum := sum + 1;}
6. \hspace{2em} /* sliced away */
7. \hspace{1em} \texttt{I++;}
8. \texttt{end}
9. \texttt{write(sum);}  

Figure 2.12: An example of dynamic slicing: \( \text{slice}(n = 3, 9^{12}, \text{sum}) \)

### 2.4 Conditioned Slicing Algorithm

“Conditioning, similar to program slicing, is a form of source code manipulation that allows a software engineer to extract an executable subprogram based upon a criterion of interest” [36]. In general, there are two steps within conditioned slicing: program
conditioning and static slicing. Program conditioning consists of removing a set of statements which cannot be executed when a condition holds at some point in a program.

Conditioned slicing forms a theoretical bridge between both static and dynamic slicing. It extends the traditional slicing criterion with a condition which captures a set of initial program conditions. The condition can reduce the program before applying a traditional static slicing algorithm. The simplification of processing specific conditions is called conditioning, which is achieved by deleting statements which do not contribute to the computation of the variables when the program is executed in an initial condition.

However, conditioned slicing approaches to program conditioning heavily rely upon both symbolic execution and reasoning about symbolic predicates. The reasoning can be performed by a ‘heavy duty’ theorem prover but this may impose unrealistic performance constraints. This is the main reason why conditioned slicing has not been developed very well, rather than static program slicing and dynamic slicing.

As to the research in this area, Danicic et al. [36] formulated a lightweight approach to theorem prover using the FermaT as a component to ConSUS, a program conditioning system for the Wide Spectrum Language WSL. The paper also provided empirical evidence that conditioning produces a significant reduction in program size. However, empirical results are still to be provided further since there has been no evidences that real programs can be ‘symbolically executed’ with the theorem proving.

As an illustrative example, also consider the program in Figure 2.10 and suppose that program conditioning be \( N < 10 \). When the program is run at statement 4, the process is that the algorithm inquires to theorem prover where the program should go based on the pre-conditioning (\( N < 10 \)). Thus, the whole conditioning process heavily depends on the performance of the theorem prover, and otherwise, ‘bad’ provers can
lead to imprecise results.

2.5 Applications of Program Slicing

2.5.1 Program integration

Due to program updating or maintenance, it is necessary to integrate several versions of a program into a common one. However, it is a complex and time consuming task to integrate programs manually. In order to apply the integration techniques, program difference analysis is needed which is the task of analysing an old and a new version of a program in order to determine the set of program components of the new version that represent syntactic and semantic changes [61]. On the basis of the program difference, program integration can be achieved by creating a new version that incorporates several of the changes and then merging them in a way that combines their separate features.

Horwitz, Prins, and Reps [62] discussed the program integration algorithm by comparing slices in order to detect equivalent behaviors. They used the static slicing algorithm for intra-procedural programs with Procedure Dependence Graph (PDG), which integrates changes in variants of a program. The algorithm consists of the 4 steps and refers to a program Base, and two variants A and B that have been derived from Base as the inputs of their algorithm [62]:

"Determining the Differences in Behavior of a Variant" : Characterize the difference between the behavior of Base and its variants A and B. The program dependence graphs are used for a representation from which to determine these changes with respect to the program Base.

Merging Program Dependence Graphs : Create the merged program dependence graph $G_M$. Graph $G_M$ is formed by taking the union of three sets of
slices; they represent the changed behaviors of A and B with respect to Base and the behavior of Base that is preserved in both A and B, respectively.

**Determining Whether Two Versions Interfere** Before using the algorithm to form the final integrated program, it is necessary to check whether A and B interfere with respect to the program Base. If a merged program dependence graph, $G_M$, that is created by the method described in the previous section can fail to reflect the changed behavior of the two variants A and B in two ways, it is claimed that interference occurs:

1. because the union of two feasible PDGs is not necessarily a feasible PDG, $G_M$ may not be a feasible PDG.
2. it is possible that $G_M$ will not preserve the differences in behavior of A or B with respect to Base.

**Reconstituting a Program From the Merged Program Dependence Graph:**

Horwitz et al. [62] provide function Reconstitute Program to create an appropriate program from $G_M$.

For an illustrative example to find the different from original and variational programs by using slicing shown in Figure 2.13–2.16, variation 1 and variation 2 add the conditional statement comparing to the Base program. From the difference of $\text{slice(write(sum))}$ of Base, variation 1 and 2, their algorithm can reconstruct the PDG by combining 3 PDGs based upon slices of variable $\text{sum}$. The integrated program is shown in Figure 2.16.

Reps and Yang [106] proved semantic justification of Horwitz, Prins, and Reps integration algorithm [62] by the mathematics-based slicing theorem and termination theorem. Furthermore, Yang, Horwitz and Reps improved the Horwitz, Prins, and Reps integration algorithm by using semantics-preserving transformations to loosen
1 \( a := 1; \)
2 \( x := 1; \)
3 \( sum := 0; \)
4 \textbf{while} \( x < 5 \) \textbf{do}
5 \( a := a/x; \)
6 \( x++; \)
7 \textbf{end}
8 \( sum := sum + a * x; \)
9 \( \text{write}(sum); \)

Figure 2.13: An example of program integrations: Base program

1 \( a := 1; \)
2 \( x := 1; \)
3 \( sum := 0; \)
4 \textbf{while} \( x < 5 \) \textbf{do}
5 \( a := a/x; \)
6 \( x++; \)
7 \textbf{if} \( a < 1 \) \textbf{then}
8 \( sum++; \)
9 \textbf{end}
10 \textbf{end}
11 \( sum := sum + a * x; \)
12 \( \text{write}(sum); \)

Figure 2.14: An example of program integrations: Variation 1

1 \( a := 1; \)
2 \( x := 1; \)
3 \( sum := 0; \)
4 \textbf{while} \( x < 5 \) \textbf{do}
5 \( a := a/x; \)
6 \( x++; \)
7 \textbf{if} \( a > 3 \) \textbf{then}
8 \( sum--; \)
9 \textbf{end}
10 \textbf{end}
11 \( sum := sum + a * x; \)
12 \( \text{write}(sum); \)

Figure 2.15: An example of program integrations: Variation 2
Search-Based Slicing

2.5.2 Testing

In general, there are several test methods in current testing techniques. Data flow testing is one to which slicing can be applied. A data flow test criterion refers to all definition-use pairs in a successful test suite. A program error may indicate the existence of some incorrect definition-use pairs which influence the incorrect output. This criterion is referred to as the ALL-DU.

Duesterwald, Gupta, and Soffa [41] introduced rigorous testing criterion, based on program slicing. That is, each def-use pair must be tested in a test-case; it must have an influence on at least one output value, called output-influencing. A def-use pair is defined as output-influencing if it occurs in an output slice. Thus, an automatic test-case generator can be used to construct enough test-cases such that all def-use pairs
are tested. Kamkar, Fritzson and Shahmehri [79] extend the ‘output-influencing’ def-use pair testing to inter-procedures by using inter-procedural dynamic dependence summary graph.

Gupta et al [56] formulated an approach to reduce the cost of regression testing. Traditional regression testing requires data flow history for re-computation of data flow for an entire program to detect affected definition-use associations and maintain a test suite [56]. Binkley [15] used forward slicing to find out affected points in programs and also defines program ‘difference’ representing the subprogram of modified programs. This paper argued that for the regression testing, just a program ‘difference’ needs to be re-run due to the maintenance of original programs.

Bates and Horwitz [10] applied program dependence graphs to the testing of modified programs. The difference from the methods of Gupta and Binkley is that Bates and Horwitz analyse dependence graphs for reusable test cases based on the different data testing adequacy criteria so that the algorithm can be easily extended to fit another adequacy criterion, i.e. the paper provided testing of PDG edge adequacy criterion, a variant of all-statements adequacy criterion.

This paper [10] presented an approach to selective regression testing using the concept of program slicing. That is, the technique uses both backward and forward slicing algorithms to determine directly and indirectly affected def-use associations. For direct influence, the backward algorithm moves backwards through the program, from the point of the edit, that searches for definitions related to the changed statement. For the indirect influence, the forward algorithm moves forwards from the point of the edit to detect uses, and subsequent definitions and uses. Through these two algorithms, the technique detects DU associations that are directly or indirectly affected because of program modifications.

As a simple illustration, consider Figure 2.14 which Figure 2.13 is modified based on. According to the backward and forward \textit{slice}(\textit{sum}, 12), it is just necessary that
Def-Use pairs of statement (3,8) and (8,11) needs be tested. Instead, testing cost is reduced due to avoiding to re-test the whole pairs of the modification.

2.5.3 Program metrics

Cohesion measurement

Ott and Thuss [101] viewed a module as a set of processing elements that act together to compute the outputs of a module. In general, high module cohesion is often suggested as a desirable property of program modularity. There are seven levels of modular cohesion identified by Constantine and Yourdon [34]. Ott [101] re-classified the cohesion class of a module into four levels by combining the former definition and formulating the relationship between slices according to different output variables and processing elements by defining the slice profile. The low cohesion class consists of coincidental and temporal modules corresponding to the lowest cohesion levels defined by Constantine and Yourdon in which a module is partitioned into disjoint sets of unrelated processing elements. Each element is involved in the computation of a different output value, and there is no intersection between the slices in terms of the output variables. There are no control or data relationships among the processing elements. The lack of processing element relationship is reflected by some non-intersecting slices in the profile for the module. For example, the program in Figure 2.17 suggests a low level since slice$(x) \cap$ slice$(a)$ is empty.

The second cohesion level is called control cohesion. It is associated with Constantine and Yourdons logical and procedural cohesion levels. Control cohesion is present when the only shared processing element is the intersection of the common control statement. Control cohesion consists of two or more sets of independent processing elements each of which depends on a common control statement. The intersection of slices will consist of control predicates. This is suggested in the slice profile by determining the overlap of all of the slices, which is the for, while or if statement.
If these statements are removed from each slice, the resulting set of slices will present low cohesion. For example, the program in Figure 2.18 suggests the control level since $\text{slice}(x) \cap \text{slice}(a)$ is just control statement `while` loop.

The third cohesion level is data cohesion corresponding to situations where data flows from one set of processing elements to another. There are two types data dependency: input-sided and output-sided communicational data flow. Slices have non-empty intersection and non-trivial differences. The data dependency is reflected in the slice profile by determining the intersection of all the slices at common statements. For example, the program in Figure 2.19 suggests the data level since $\text{slice}(x) \cap \text{slice}(a)$ is non-trivial empty, and both of $\text{slice}(x)$ and $\text{slice}(a)$ are included in $\text{slice(sum)}$.

The high cohesion level consists of the sequential and functional levels defined by Constantine and Yourdon [34], which is the data flow from a processing element flows to its successor. The processing elements are most highly interrelated due to the strong dependency of one on the other. The dependency is sequential since any assignments of one element are made after the other element has been determined. The most high processing element relationship is suggested in the profile by complete intersection between the slices. In particular, one slice $A$ is entirely contained within the other slice $B$. For example, the program in Figure 2.20 suggests the function level since $\text{slice}(x)$ includes $\text{slice}(a)$, which means the computation of former is dependent on that of latter.

```plaintext
1  a := 1;
2  x := 1;
3  while 1 < 5 do
4      a := a * 2;
5  end
6  while 1 < 5 do
7      x++;
8  end
9  write(a);
10 write(x);
```

Figure 2.17: An example of 4 types of cohesion: low level
Search-Based Slicing

Figure 2.18: An example of 4 types of cohesion: control level

```plaintext
1 a := 1;
2 x := 1;
3 while 1 < 5 do
4   a := a * 2;
5   x++; 
6 end
7 write(a);
8 write(x);
```

Figure 2.19: An example of 4 types of cohesion: data level

```plaintext
1 a := 1;
2 x := 1;
3 while 1 < 5 do
4   a := a * 2;
5   x = a * x;
6 end
7 sum := sum + a * x;
8 write(a);
9 write(x);
10 write(sum);
```

Figure 2.20: An example of 4 types of cohesion: function level

```plaintext
1 a := 1;
2 x := 1;
3 while 1 < 5 do
4   a := a * 2;
5   x = a * x;
6 end
7 write(a);
8 write(x);
```

The paper does not rely on any specific slicing method, and no quantitative measures are presented.

Measurement of dependences

Binkley and Harman [16] applied program slicing to dependence analysis and put forward the concepts ‘dependence clusters’ and ‘dependence pollution’, which can
contribute to the program maintenance and alleviate the complexity of the program.

A dependence cluster is a set of program statements all of which are mutually interdependent. The clusters can make problems for maintenance since a change to any statement in a cluster will have a potential impact on all statements in the cluster [16]. In some cases, the dependence binding the other statements in a dependence cluster is avoidable. This is called dependence pollution.

This paper [16] introduced a simple approach for finding dependence clusters in terms of slice sizes. A visualisation is introduced by defining the Monotonic Slice-size Graph (MSG). The paper points out that dependence clusters can be identified using program slicing because two nodes which depend upon each other must have the same slice and vice versa [16]. The approach claims that nodes of the dependence graph are in a dependence cluster if the size of their slice is identical, rather than the contents of slices is completely identical.

As an example illustrating the ‘dependence clusters’ and ‘dependence pollution’, Figure 2.21 shows the fragment in which 3 variables \((a, b \text{ and } c)\) are in the dependence cluster since they have the same slices. However, the clusters can be eliminated by transforming the original program into the one in Figure 2.22, which causes variables \(a, b \text{ and } c\) not to be in the same slice any more.

```
1  i := 0;
2  x := y + z;
3  y := x + z;
4  z := x + y;
5  while 2x < z - y + 10 do
6       a := a * a + (x - y);
7       b := b * b + (z - 2y - x);
8       c := c * c + (z - 2x - y);
9       if a = 10 or b = 100 or c = 50 then
10          break;
11  end
12 end
```

Figure 2.21: An example of dependence clusters and pollution: original program
1 \ i := 0;
2 \ x := y + z;
3 \ y := x + z;
4 \ z := x + y;
5 \textbf{while} 0 < 10 \textbf{ do}
6 \quad a := a \ast a;
7 \quad \textbf{if} \ a = 10 \textbf{ then}
8 \quad \textbf{break};
9 \quad \textbf{end}
10 \textbf{end}
11 \textbf{while} 0 < 10 \textbf{ do}
12 \quad b := b \ast b;
13 \quad \textbf{if} \ b = 100 \textbf{ then}
14 \quad \textbf{break};
15 \quad \textbf{end}
16 \textbf{end}
17 \textbf{while} 0 < 10 \textbf{ do}
18 \quad c := c \ast c;
19 \quad \textbf{if} \ c = 50 \textbf{ then}
20 \quad \textbf{break};
21 \quad \textbf{end}
22 \textbf{end}

Figure 2.22: An example of dependence clusters and pollution: transformed program
The paper [16] gave case study examples of what might be considered to be dependence pollution, illustrating the way in which dependence cluster analysis can be used to support and inform maintenance activities. Also, the paper pointed out there are two possible sources of dependence pollution are Mutually Recursive Clusters (MRCs) and Capillary Data Flows (CDFs). The paper argues: (i) “mutual recursion naturally leads to large dependence clusters because each function calls all the others, making the outcome of each function dependent upon the outcome of some call to each of the others;” (ii) “A Capillary Data Flow is a data flow which occurs between two large and otherwise unconnected clusters through a single variable. The variable acts as a small capillary vessel along which the dependence flows creating one large cluster from two or more, otherwise unconnected, sub-clusters.” On the basis of two types of pollution, the empirical results were given that dependence pollution may be eliminated by refactoring the program.

2.5.4 Program debugging

Program Debugging might be a tedious task especially when programmers develop large programs since few clues can be used to locate bugs in the programs. This is the main reason why program slicing is useful for debugging because it particularly allows program debuggers to skip lots of irrelevant statements in the process of looking for a bug [87]. To begin with, program slicing is posed by Weiser as an attractive application of program debugging. For instance, if a piece of code computes an improper value for a variable $n$, only the statements in the $\text{slice}(n)$ may possibly contribute to the computation of the value $n$. Therefore, programmers only need to focus on the slice instead of the whole program code in the way that the complexity of the task of program debugging can be eased.

such as ineffective statements, ineffective imported values and undefined variables and indicates that this code is apt to being the presence of bugs. Moreover, the authors especially analysed the repetitive statements while loop so as to find erroneous code.

Jayaraman et al. developed a java slicing tool, Indus [65] to help Eclipse users analyse the program. The tool also can provide every possible path of the computation of the variable of interest so as to locate the buggers due to the false output of variables. Slicing techniques have been used in some software analysis tools to assist program analysis, (e.g. ClearMaker combined with program slicing to determine the impact of program modifications which can easy the process of regression testing).

As originally introduced, static slicing involves all possible program executions. Although static slicing has many advantages in the process of program debugging, static slices still include large subprograms because of the imprecise computation of the slices. Dynamic slicing may more precisely identify the interesting parts concerned by programmers because they contribute to the computation of the incorrect output for a given program execution.

Several different techniques for computation of dynamic slices have been proposed [2, 53, 70, 78]. By considering a particular program execution, dynamic slicing may significantly reduce the size of the slice in comparison of static slicing. Therefore, dynamic slices are particularly useful in debugging, because they only reflect the actual dependences of that execution, hereby, smaller slices than static ones.

Agrawal et al. [1] introduced backtracking techniques which can restore previous states of statements and combined dynamic data and control slicing to trace back until localising the fault. During the process, local analysis and global analysis are used to guide the selection of error-like statements.

Choi et al. [31] designed Debugger PDD for sequential programs as well as parallel programs by flowback analysis. The authors split the process of debugging into 3 phases – preparatory, execution and debugging phases, and eventually generated a
‘back-traceable’ dynamic dependence graph. This is a semi-automated debugging process since user intervention is needed in order to obtain the dynamic graph during the last step.

Fritzson et al. [49] introduced a similar debugging algorithm to Choi et al. [31] including transformation tracing and debugging phases. However, the last step consists of the algorithmic debugging technique introduced by Shapiro [111] and test case partitioning with program slicing to reducing the irrelevant statements considered.

Korel and Rilling [77] defined several dynamic slicing features that may be useful in program debugging. They described the concepts of executable slices, influencing variables, contributing nodes, and partial dynamic slicing that are ‘extensions’ of executable dynamic slicing. The paper [77] also described the dynamic slicing tool that achieves these kind of dynamic slicing features.

2.5.5 Program maintenance

Schneidewind [108] posed that one of the reasons that maintenance is difficult is that it is hard to determine when a code change will affect some other piece of code. Thus, this work claims that software maintenance is more demanding than development. Another difficulty is due to large part to the semantic constraints that are placed on the maintainer. To begin with, “this problem has been addressed by attempting to eliminate these semantic constraints and then providing the maintainer with a tool that will pinpoint potential inconsistencies after changes have been implemented” [108].

Gallagher applied program slicing to the maintenance problem “by extending the notion of a program slice to a decomposition slice, which generally represents all computation on a given variable irrelevant to any statement number [50]. The paper demonstrated how to form a slice-based decomposition for programs. The paper is then able to delineate the effects of a proposed change by isolating those effects in
a single component of the decomposition. This gives maintainers a straightforward technique for determining those statements and variables that may be modified in a component and those that may not.

In order to eliminate the effect between the decomposition slicing and its complement (defined below), the paper put forward a set of definitions which can contribute to compute independent statements/variables.

Decomposition slices are independent if they have no intersection; one decomposition slice is dependent on the other one if one contains the other completely; an output-restricted slice is maximal if it is not dependent on any other slice; slice dependent statements are defined as intersection of two decomposition slices; independent statements are defined as non-intersection among all the decomposition slices; A variable that is the target of a dependent assignment statement is called a dependent variable. Alternatively and equivalently if all assignments to a variable are independent statements then the variable is called an independent variable.

Therefore, the decomposition principles can be stated as: Given a maximal output restricted decomposition slice of the program, delete the independent and output statements from the program and call it the complement of decomposition slice with respect to the program. With definition, some constraints in this paper are referred to as four maintenance rules:

1. Independent statements may be deleted from a decomposition slice.

2. Assignment statements that target independent variables may be added anywhere in a decomposition slice.

3. Logical expressions and output statements may be added anywhere in a decomposition slice;

4. New control statements that surround control any dependent statement will cause the complement to change [50].
procedure ProcessArray(integer N; integer X[ ]; integer sumX)

I: integer;
J: integer;
SumX = I;
for I = 1 to N do
    X[I] = 0;
    for J = 1 to I do
        X[I] = X[I] + J;
    end
end
for I = 1 to N do
    SumX = SumX + X[I];
end

Figure 2.23: An original program for the decomposition slicing example.

// The complement of the decomposition slice slice(SumX)
I: integer;
J: integer;
for I = 1 to N do
    X[I] = 0;
    for J = 1 to I do
        X[I] = X[I] + J;
    end
end

Figure 2.24: The complement of the decomposition slicing slice(SumX).
For example, consider the decomposition slice \( \text{slice}(\text{SumX}) \) of the program in Figure 2.23. The decomposition slice includes all the statements in the program, and thus, its complement of the slice is shown in Figure 2.24. In Figure 2.23, \( \text{SumX} \) in the statement 4 is independent which, thus, is available for the Rule 1 and 2 as well as the statements 11 and 12. Variables of statements 5, 6, 7 and 8 are dependent which, thus, is available for Rule 4. Moreover, the output statement for the variable \( \text{SumX} \) can be added to anywhere in the decomposition slice in Figure 2.24).

On the basis of decomposition slicing, the paper states that the maintainer can test the changes in the component with the assurance that there are no linkages into other components. The change is only needed to be tested in the decomposition slice. Thus, “decomposition slicing induces a new software maintenance process model that eliminates the need for regression testing” [50].

### 2.5.6 Refactoring of the program

Even if an original system is well designed, changes may have to be made by different people in order to be prepared for new requirements. However, the process of updating will get harder when maintaining the system due to the change made causing the quality decreasing of original structures.

The goal of software refactoring is to maintain a system by transforming the software from one representation to another without changing the semantics of the system. Changing the internal syntax of a program without changing its behavior is called refactoring.

**Tucking statements into functions**

Lakhotia and Deprez [82] investigated the problem of restructuring programs by breaking its large code fragments and tucking them into new functions. The paper presented a restructuring transformation tucking to decompose large, non-cohesive
code fragments into small, cohesive functions. ‘Tucking’ is a composition of three primitive transformations: Wedge, Split, and Fold [82]. The technique combines the program slicing and transformations.

Consider a function to be represented as a Control Flow Graph (CFG) and a program as a collection of CFGs. The input and output of the transformations are CFGs, its components, and some relations over these components. To tuck a program fragment, a programmer first slices the program by driving a wedge in the function, then splits the CFG of the program separated by the wedge, and then folds the split (a sub-CFG) into a new CFG [82]. The process is defined by Lakhotia and Deprez as follows:

**Wedge** Which takes the input which will be as a slicing criterion and returns those statements in the slice.

**Split transformation** Which, firstly, creates CFGs for the original program and a set of statements chosen at the wedge operation. Secondly, separate the CFG of the original program into the two sub-CFGs based upon the results of the wedge which divide the program into the slice part and the other.

**Fold transformation** Which is to create a new CFG by representing a new function and to substitute the subgraph in the original CFG with the node of a function call to the new sub-function.

As an illustrative example shown in Figure 2.23, **Wedge** is to do the slicing in terms of the criterion \((X[I], 8)\); **Split** is to divide the CFG of the original program into two sub-CFGs based upon the results wedged, in which some statements need to be kept in the remaining program in the condition of semantics unchanging (statement 2 in this situation); **Fold** is to replace the extracted procedure with a call. The final program is shown in Figure 2.25.
Untangling extraction

Ettinger et al. [43] examined how method extraction can be improved through program slicing. To implement the ‘Extract Slice as Method’ refactoring, the paper employed a variation of a method extraction approach by Lakhotia and Deprez [82]. Five sets of statements are defined in this paper in terms of a slicing criterion \(< s, v >\):

“**scope** : all statements that may be executed before \(s\);

**extracted** : \(\text{slice}(< s, v >)\);

**notExtracted** : \(\text{scope}/\text{extracted}\);

**kept** : \(\text{slice}(\text{notExtracted})\);

**deleted** : \(\text{scope}/\text{kept}\);”
As described in the paper, ‘double slicing’ is used to keep the some marked statements in the remaining program in context of preserving the intact semantics of the original program. The idea is to make sure that every statement that contributes to the computation of a kept statement must be kept as well. This is the reason for using slicing twice in the computation of kept.

For example of the program shown in the Figure 2.23, extracting the computation of \(< X[7], 8 >\) yields:

\[
\begin{align*}
\text{scope} & : \{2, \ldots, 8\} \\
\text{extracted} & : \{2, 3, 5, 6, 7, 8\} \\
\text{notextracted} & : \{4\} \\
\text{kept} & : \{2, 4\} \\
\text{deleted} & : \{3, 5, 6, 7, 8\}
\end{align*}
\]

The results of the method extraction is in the Figure 2.25 which is the same as the results by Lakhotia and Deprez [82].

**Transformation based procedure extraction**

Komondoor illustrated four transformation techniques in which the algorithm consists of statement reordering, predicate duplication, promotion and handling exiting jumps, which combine into the three steps of procedure extraction [74]:

1. The statements that are extracted are referred to as the ‘marked’ statements.

2. Semantics-preserving transformations are applied to make the marked statements a contiguous, well-structured block.

3. The marked statements are extracted into a new procedure, and replaced with a procedure call.
The focus of the steps is the second step of procedure extraction. The paper partitions the nodes in the region into three ‘buckets’: before, marked, and after by defining several relationships between them based on the analysis of data and control dependence. Consequently, “the output is created by stringing together the three hammocks with using the entry node of the marked hammock as the outside-exit node of the before hammock, and using the entry node of the after hammock as the outside-exit node of the marked hammock” [74]. Hammock refers to a subgraph of a CFG that has a single entry node, and from which control flows to a single outside-exit node; e-hammock is a subgraph of a CFG that has a single entry node – a hammock with exiting jumps.

In the paper, the authors defined a series of constraints to determine which nodes should fall into one of the buckets [74]:

“Data-dependence constraints : A data-dependence constraint $m \leq n$ is generated for each pair of nodes $m, n$ if there is $n$ is data dependence on $m$, such that $m$ must either go into a bucket that precedes $n$’s bucket, or must go into $n$’s bucket.

Loop-structure constraints : Assignment statements in e-hammock are never duplicated across multiple buckets, such that that $m$ and $n$ must be placed in the same bucket, which is referred to as $m = n$.

Control-dependence constraints : For each node $n$ in e-hammock and for each control ancestor $p$ of $n$ in e-hammock, the constraint $n \Rightarrow p$ is generated such that a copy of node $p$ must be included in the same bucket as node $n$.”

For example, consider the program in Figure 2.26. Suppose that one wants to extract computation of array $X$ and marked statements are selected in terms of a source code analysis tool as shown with bold font.
before bucket. \( \text{read}(X[0]) \leq i f(X[0] < 100) \) data-dependence constraints is generated since statement 5 is dependent on statement 4.

**promoted marked node:** \((X[I] = ABS(X[I])) = (for \ I \ = \ 1 \ to \ N \ do)\) loop-dependence constraints is generated since statement 8 and 9 are in the same loop. Similarly, the statements 10 and 11 are in the same bucket.

duplicate nodes. \( \text{Sumtotal} + = 1 \Rightarrow i f(X[0] < 10000) \) control-dependence constraints is generated since statement 6 is dependent on statement 5, such that \( i f(X[0] < 10000) \) is copied to before bucket (only if the copied statement is already in a bucket and otherwise, only move the statement into the corresponding bucket). Thus, results are shown in Figure 2.27 after extracting the marked nodes.

```
1 I: integer;
2 J: integer;
3 SumX = 0;
4 read(X);
5 if X[0] < 10000 then
6     Sumtotal+ = 1;
7 end
8 for I = 1 to N do
9     X[I] = ABS(X[I]);
10    for J = 1 to I do
11       X[I] = X[I] + J;
12 end
13 end
14 for I = 1 to N do
15     SumX = SumX + X[I];
16 end
```

Figure 2.26: An example of refactoring the function based upon transformation based procedure extraction: original program
Amorphous procedure extraction

Harman et al. [57] introduced a variety of procedure extraction [74] – amorphous procedure extraction. The paper claims that fewer marked statements can be achieved by relaxing the limitation of the syntax preservation while maintaining the semantics of the program. The authors argued that syntax preservation may cause imprecise promotion of unmarked statements.

For example, there is a fragment shown in Figure 2.28. If considering the computation of total of line 9, the line 5 and 7 are marked. Since line 7 is dependent on line 6, the line 6 also has to be marked, as results shown in Figure 2.29. However, if using pushing down transformation exchanges the position of line 6 and 7 shown in Figure 2.30, extraction results can be simpler, as shown in Figure 2.31.

Moreover, amorphous procedure extraction simply deals with the situation of side effects, which is any state change caused by the evaluation of an expression. Side effects increase the complexity of procedure extraction. It may become necessary to
1 a : integer;
2 b : integer;
3 c : integer;
4 while i++ < 10 do
5     a = a + 1;
6     c = a + 2;
7     b = c + 3;
8 end
9 total = a * b;
Figure 2.28: An example of refactoring the function with syntax-preserving extraction: original program

1 a : integer;
2 b : integer;
3 c : integer;
4 call abc;
5 total = a * b;
6 procedure abc;
7 while i++ < 10 do
8     a = a + 1;
9     c = a + 2;
10    b = c + 3;
11 end
Figure 2.29: An example of refactoring the function with syntax-preserving extraction: transformed programs

1 a : integer;
2 b : integer;
3 c : integer;
4 while i++ < 10 do
5     a = a + 1;
6     b = a + 2 + 3;
7     c = a + 2;
8 end
9 total = a * b;
Figure 2.30: An example of refactoring the function with amorphous extraction: original program
extract whole statements. Amorphous extraction can solve the problem increasing the unnecessary promotion. For example, consider the program fragment shown below:

```
a=a+(b++)+1;
  a=a+2+3;
  c=b+2;
```

If one extracts computing of the variable \( c \), the marked statement is referred to as bold font. Due to using the ‘++’, the first statement must be marked, rather than just include the expression \( b++ \). Applying the transformation can solve the problem and only extract useful expression, but the whole statement, as shown in below:

```
a=a+b+1;
  b++; 
  a=a+2+3;
  c=b+2;
```
2.6 Search Based Software Engineering and Applications

“Search based Software Engineering (SBSE) is a framework for considering the application of metaheuristic search techniques to software engineering problems. The SBSE framework allows search based techniques to be used to provide acceptable solutions in situations where perfect solutions are either theoretically impossible or practically infeasible” [60].

In order to apply the framework to a specific software engineering problem, it is necessary to reformulate the problem as a search problem [52, 118, 119], where natural representations, fitness functions and search based algorithms are necessary to define: a representation that is subject to symbolic manipulation of the problem; a fitness function which is defined according to the definition of the representation; a set of search-based operators which can calculate optimal fitness values [60].

Search based techniques have been applied to many software engineering activities from requirements engineering [6], project planning and cost estimation [3, 4, 5, 28, 39, 72] through testing [8, 9, 23, 26, 27, 55, 59, 85, 91, 114], to automated maintenance [24, 45, 58, 93, 94, 97, 109, 110], service-oriented software engineering [30], compiler optimization [33, 35] and quality assessment [25, 71]. Also there has been much previous work on SBSE [44, 59, 88, 94, 114].

As to the application of source code analyses, three papers related to the search based refactoring of the program were introduced, as described below.

2.6.1 Search based determination of refactoring

Seng et al. [110] argued that search based approaches are also applicable when reconstructing the class structure of a system. The paper formulated a methodology for object-oriented systems “that helps the user to determine refactoring to improve
the class structure of a system with respect to the values of several metrics and the number of violations of object oriented design principles” [110].

Seng et al. implemented the approach in detail by focusing on the setup of the model, the fitness function and crossover operations [110].

Source code model and model refactorings define the basic object oriented elements—classes, methods, attributes, parameters and local variables, and model refactorings—extract class, inline class, move attribute, push down attribute, pull up attribute, push down method, pull up method, extract superclass and collapse class hierarchy, which simulate the actual source code refactorings.

Combined fitness function is a sum of several metric values and is designed to be maximized. The paper captures 4 metrics including coupling, cohesion, complexity and stability, which are referred to as criteria to select the candidate solution during the operation of crossover and mutation.

Mutation extends the current chromosome by an additional model refactoring mentioned above.

Crossover combines two chromosomes by selecting the model refactorings from parent one and adding the model refactorings of parent two to the chromosome.

2.6.2 Search based software maintenance

O’Keeffe at el. [97] also formulated the task of refactoring as a search problem. They have constructed a search-based software maintenance tool based on the quality evaluation functions that are claimed to reflect refactoring quality.

Similarly to the methodology in [110], the paper defined the evaluation of measurement as the fitness function, model refactorings and search techniques.

The paper has chosen refactorings that operate at the method and field level of granularity, while [110] can operate on the level of the class. Also, the paper compared the performance of three search based algorithms—First-Ascent Hill Climbing
Search-Based Slicing

Literature Review

(HC1, Steepest-Ascent Hill Climbing (HC2) and Low-Temperature Simulated Annealing (SA)). Evaluation of fitness functions is based on the several metrics defined in the [11], including Design Size in Classes (DSC), Number Of Hierarchies (NOH), Average Number of Ancestors (ANA), Direct Class Coupling (DCC), Cohesion Among Methods of Class (CAM) and so on, which evaluate the software maintainability.

2.6.3 Search based amorphous slicing

Fatiregun et al. [45] applied the search techniques to generating the amorphous slicing comparing to the ‘traditional algorithms’. Amorphous slicing is a form of slicing, in which any transformations can be applied, rather than merely statement deletion which is used in syntax preservation slicing.

The amorphous slice produced can be a lot smaller than syntax-preserving slicing. Thus, for those applications which require slices to be as small as possible and where syntax preservation is unimportant, amorphous slicing is clearly attractive. However, since finding good general algorithms is hard, amorphous slicing technique are less well developed.

As an example of amorphous slicing, Figure 2.32 shows comparison between syntax-preserving slicing and amorphous slicing. As shown in the figure, amorphous form of slice(sum, 5) is simpler than the form of static slicing.

The paper [45] showed that amorphous slices can be computed using search techniques by reformulating amorphous slicing as a redundancy removal problem and then searching the optimum sequences of transformations. The paper applied 4 search approaches—Genetic Algorithm, Hill Climb, Systematic Algorithm and Random Algorithm. Empirical results argued that at least one of 4 algorithms is better compared to the analytical algorithm, which uses traditional syntax-preserving static slicer.
Search-Based Slicing

/* original fragment */
1 x = a + 1;
2 y = x * a + 2;
3 z = 2 * y;
4 temp = x * y * z;
5 sum = 2 * z;

/* syntax-preserving slice */
6 x = a + 1;
7 y = x * a + 2;
8 z = 2 * y;
9 sum = 2 * z;

/* amorphous slice */
10 sum = 2 * (a + 1) * a + 2;

Figure 2.32: An example of amorphous slicing

2.7 Summary

This chapter presented a review of the current program slicing techniques and their applications, and introduced the general framework of Search Based Software Engineering and its applications.

Weiser originally conceived of program slices as a model of the mental abstractions made by programmers when debugging a program, and advocated the use of slicing in debugging tools. Slicing has also proven to be useful for a variety of other applications including: program integration, testing, program maintenance, measurement, debugging and refactoring. At the end, the framework of Search Based Software Engineering was introduced as well as its applications.
Chapter 3

The Framework of Search Based Slicing

This chapter introduces the Search-Based Slicing (SBS) framework which is based on the Search-Based Software Engineering framework. The significance of the SBS framework is demonstrated through an illustrative example.

SBS lies within the general SBSE framework, that is, program slicing analysis using the theory of SBSE. SBS project belongs to the field of Software Engineering. The utility and power of program slicing stems from the ability to assist in tedious and error-prone tasks such program debugging, testing and re-engineering [18, 20, 112]. SBS seeks to use this power to locate program dependency structures of interest.

SBS will require work on search algorithms, representation of properties and visualization of results. There will also be an empirical element to the evaluation of the work, which will require detailed statistical analysis and experience with experimental and empirical software engineering.

The aim of SBS is to develop automated search techniques to explore the space of possible slice sets in order to identify properties and dependency structures of interest within the source program. In the first step, in order to validate that the framework
is applicable to program slicing, the thesis will consider the following questions on
the decomposition of the program into a slicing set:

“Based upon the SBSE framework combined with program slicing techniques,
  can we explore search algorithms to find out the decomposition that is op-
  timum or close to optimum within the large space of slicing sets?”

The analysis of empirical results studied proved that the SBSE framework is ap-
  plicable to applications of program slicing in analysis of dependency structures of the
  program. The feature of this framework is flexible since more possible applications
  could be found only based on the change of fitness function in terms of specific issues.

3.1 Search Space

The purpose of all search algorithms is to locate the best (or an acceptably good) so-
  lution among a number of possible solutions in a search space. The process of looking
  for a solution is equivalent to that of looking for some extreme value—minimum or
  maximum in the search space.

In the experiments reported here, the search space is all the possible sets of slices,
  that is the power set of all the possible sets of slices. Following Horwitz et al. [64], a
  ‘possible slicing criterion’ is taken to mean ‘any node of the System Dependence
  Graph (SDG) of the program’. Therefore, for a program with \( n \) nodes in the SDG,
  there will be \( n \) corresponding slicing criteria and, therefore, \( 2^n - 1 \) subsets of slices.
  These \( 2^n - 1 \) subsets of slicing criteria form all the possible sets of slices. Consequently,
  the search space is the power set of all the subsets—\( 2^n - 1 \). Clearly, enumeration will
  not be possible since \( n \) can be arbitrarily large. This observation motivates the search
  based software engineering approach advocated in this thesis.
3.2 Representation of Solutions & Slices

The representation of a candidate solution is critical to shaping the nature of a search problem. In the problem of program decomposition, the representation of final solutions is that a program can be divided into a number of slices. Let \( S[i] \) be a binary bit and \( i \) be a corresponding slicing criterion, so that \( S[i] = 1 \) if the solution include the corresponding slice based on the criterion \( i \) and \( S[i] = 0 \) if the solution does not include the corresponding slices based on the criterion \( i \).

In the solution set, each element consists of a set of slices which represents the whole program. For the convenience of computing the intersections of sets of slices, the definition of representation of slicing can be formulated as a simple 2-dimensional array: Let \( A[i, j] \) be a binary bit, \( i \) be a program point and \( j \) be a slicing criterion, so that \( A[i, j] = 1 \) if the slice based on criterion \( j \) includes the program point \( i \) and \( A[i, j] = 0 \) if the slice based on criterion \( j \) does not include the program point \( i \).

In this way, the array \( A \) denotes the set of slices of the program, with both array bounds determined by the number of program points (i.e. nodes of the SDG, see an illustrative example given in Table 4.1 in Section 4.1).

3.3 Fitness Function

Fitness function definition depends mainly on the specific problem. Consider splitting a procedure into several subfunctions (described in detail in Chapter 5). In this problem, the optimal solution is of the minimal overlaps between sets of slices. Thus, a mechanism needs to be introduced to measure the ‘minimum’ degree of overlap. There are two options to solve this problem. One is to define a threshold as the ‘minimum’ parameter. Assume that \( \{S_1, ..., S_n\} \) represents the set of sets of slices and \( S_i \) consists of a slicing set \( A \) (defined in section 3.2); let \( \cap(S_i, S_j) \) be intersection of two sets of slices, such that the optimum solution sought by search algorithms is
defined as follows: \( \forall i \text{ and } j \Rightarrow \cap(S_i, S_j) \leq \text{threshold} \). There is no ‘typical’ standard for the threshold, but the ratio of intersection of each pairwise to total program points in the function can be referred to as the threshold. The thesis did not investigate this empirical study, instead, the thesis used a greedy-based algorithm.

Alternatively, the greedy algorithm can be applied to computing the value of the intersection of each pairwise such that definition of threshold can be avoided. In the greedy algorithm, the slice will be chosen by comparing all the slices and checking which one will have the smallest overlap with the current set of slices. In this way, there is the guarantee that, in every time, the chosen slice is the best choice added to some set of slices.

### 3.4 Significance of the Framework

The power of the framework is combining SBSE with the program slicing technique. Program slicing is a good tool which can assist in program debugging, testing, parallelization, integration, maintenance, software metrics and so on. Even though the slicing technique can help solving tedious and error-prone tasks, this technique does not tackle NP-Hard problems. NP-Hard is referred to as problems of non-deterministic polynomial time hard, that is, there is no polynomial time algorithm for solving the problems. Informally speaking, the solution of NP-Hard problems can not be found in a reasonable time.

Consequently, the main idea for the search based slicing framework is to apply search-based approaches to all the possible sets of slices in order to find the optimal solution.

As an illustrative example, consider the following *splitting procedure* situation: procedures can be split into smaller sub-procedures if the procedures have multi-tasks. Each sub-procedure represents a semantically independent subunit. Here, there are
essentially two processes. The first step is to judge whether or not a procedure can be split; and then, if it can, the second step is to split the procedure in terms of the dependency relationship between statements.

To illustrate the idea of splitability I had in mind, Figures 3.1 and 3.2 show procedures that ‘can be split’ and those that are ‘not splittable’, respectively. The procedure of Figure 3.1 counts keyboard input characters, including letters, space and digits while the procedure in Figure 3.2 calculates the biggest common divisor and the smallest common multiple of two integer parameters.

In Figure 3.1, lines 1 – 12 denote a procedure which includes three components computing three output variables letters, space and digit. It is noticed that the three components are independent except that they are all dependent on statements 1 and 4. Thus, each component completes an independent functionality implying that the procedure can be split into three sub-procedures corresponding to lines 13 – 20, 21 – 28 and lines 29 – 36.

In Figure 3.2, the procedure consists of two components that compute two output integer variables n and p. Since the computation of p is totally dependent on the computation of n, this procedure cannot be split. However, the sub-procedure of computing the variable n can be extracted from the original procedure. Lines 17 – 29 represent the extracted sub-procedure.

Of course, these examples are merely illustrative toy examples. For more realistic procedures, the situation can be more complex than the examples considered above. That is, a procedure could include multiple components and even different components could have very tight data and control dependency relationships. Therefore, it is infeasible and error-prone to recognise different procedure components and split them manually. This thesis introduces an approach using optimisation based slicing [103, 117] to recognise procedure components and split them into executable sub-procedures.
Thus, the thesis transforms the splitting procedure problem into a search-based optimization problem as follows:

“Seek to find a set of sets of slices which compute the formals/outputs in a procedure to make sure the overlap of the sets of slices is as small as possible, such that the procedure can be divided into several higher cohesion level functions. Moreover, the process of splitting a procedure can be continued until each new sub-procedure is ‘atomic’ one.”

In summary, according to the SBS framework, only a different fitness function needs to be defined in order to interpret the feature of each specific problem.
// Original procedure
1  c : char;
2  letters, space, digit : integer;
3  letters = 0; space = 0; digit = 0;
4  while (c = getchar())! = end of line do
5    if c >= a and c <= z or c >= A and c <= Z then
6      letters ++;
7    if c == blank space then
8      space ++;
9    if c >= 0 and c <= 9 then
10      digit ++;
11  end
12  write(letters, space, digit);

// Sub-procedure1
13  c : char;
14  letters : integer;
15  letters = 0;
16  while (c = getchar())! = end of line do
17    if c >= a and c <= z || c >= A and c <= Z then
18      letters ++;
19  end
20  write(letters);

// Sub-procedure2
21  c : char;
22  space : integer;
23  space = 0;
24  while (c = getchar())! = end of line do
25    if c == blank space then
26      space ++;
27  end
28  write(space);

// Sub-procedure3
29  c : char;
30  digit : integer;
31  digit = 0;
32  while (c = getchar())! = end of line do
33    if c >= 0 and c <= 9 then
34      digit ++;
35  end
36  write(digit);

Figure 3.1: An example of splitable procedures
Search-Based Slicing

The Framework of Search Based Slicing

// Original procedure
1 p, r, n, m, temp : integer;
2 read(n, m);
3 if n < m then
4    temp = n;
5    n = m;
6    m = temp;
7 end
8 p = n * m;
9 while m! = 0 do
10   r = n%m;
11   n = m;
12   m = r;
13 end
14 p = p/n;

// The biggest common divisor:
15 write(n);

// The smallest common multiple:
16 write(p);

// Extracted procedure
17 r, n, m, temp : integer;
18 read(n, m);
19 if n < m then
20    temp = n;
21    n = m;
22    m = temp;
23 end
24 while m! = 0 do
25   r = n%m;
26   n = m;
27   m = r;
28 end
29 write(n);

Figure 3.2: An example of non-splitable procedures
Chapter 4

Decomposition of Programs *

This chapter introduces the first application of the SBS framework, that is, defines the fitness function of ‘program decomposition’. The approach is to decompose a program into a set of covering slices that minimise inter-slice overlap. The chapter also reports the result of an empirical study of algorithm performance and result-similarity for hill climbing, genetic, random search and greedy algorithms applied to a set of 12 C programs.

4.1 Problem Description

The goal is to identify dependency structures by searching the space of all subsets of program slices. In this experiment, static backward slicing is used, but the approach is not confined merely to static backward slicing; it can be used with any analysis that returns a set of program points (thereby including all forms of program slicing).

As an illustrative example, consider a program that has only 8 program points. Table 4.1 gives all the slices of this hypothetical example in terms of each program point as slicing criteria.

---

*This chapter is based on the work published on Journal of Information and Software Technology (IST) [66]
4.1.1 Fitness Function for the Decomposition

The choice of a fitness function depends upon the properties of the set of slices for which the search algorithm will optimize. This choice is a parameter to the overall approach to search based slicing. In order to illustrate the search based slicing approach, this section introduces several metrics that will be used as fitness functions to decompose a program into a set of slices that collectively cover the entire program, while minimizing the degree of overlap between the slices.

These metrics are inspired by previous work on sliced-based metrics by Bieman, Ott and Weiser [14, 86, 92, 98, 99, 100, 101, 102, 115, 116]. The following notation will be used.

<table>
<thead>
<tr>
<th>Program Slicing</th>
<th>Program point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 1 1 1 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>1 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>4</td>
<td>0 1 1 1 1 0 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1 1 0 0 0</td>
</tr>
<tr>
<td>6</td>
<td>1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>7</td>
<td>0 0 1 0 0 1 1 1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0 1 1 1 1</td>
</tr>
</tbody>
</table>

Table 4.1: An example of looking for optimal properties in program slicing sets.

The table represents the value of each slice. In this table, a 1 represents a program point that is included in the slice, while a 0 represents a program point that is not included in the slice. In this situation, a good decomposition would be the set \( \{1, 5, 7\} \), rather than \( \{1, 2, 7\}, \{6\} \) or any other subsets. The solution \( \{1, 5, 7\} \) is preferable, even though \( \{1, 2, 7\} \) has the same coverage as \( \{1, 5, 7\} \), because \( \{1, 2, 7\} \) has more overlap than \( \{1, 5, 7\} \); even though \( \{6\} \) has the same overlap as \( \{1, 5, 7\} \), because \( \{6\} \) has less coverage than \( \{1, 5, 7\} \). The other subsets have the same situation as the set \( \{1, 2, 7\} \) and/or \( \{6\} \).
Let $M$ be the number of program points of the program, $P$ be the number of program points of the optimal slicing set, $\cap(S_1, ..., S_i)$ be the intersection of $i$ slices, $\cup(S_1, ..., S_i)$ be the union of $i$ slices and $Max(S_1, ..., S_i)$ be the largest slice selected from $i$ slices. All the metrics defined below are normalized. Normalization allows for comparison of metrics from differently sized programs, while the expression as a percentage is merely a convenience: the metrics are so-defined that 100% denotes the maximum possible value. The metrics used are as follows:

**Coverage.** This measures how much the program points in a slicing set cover the program points of the whole program. This metric was introduced by Weiser [115].

$$100 \cdot \frac{\cup(S_1, ..., S_P)}{\cup(S_1, ..., S_M)} \quad 1 < P < M$$

**Overlap.** This evaluates the number of program points of the intersection within a slicing set. It can be defined in many ways; this study considers two possibilities:

**Average** For each pair of slices in the set, evaluate the percentage of program points that are in both. The average value is evaluated based on all such pairwise comparisons.

$$100 \cdot \left(\sum_{i=1}^{P-1} \sum_{j=i+1}^{P} \frac{\cap(S_i, S_j)}{Max(S_i, S_j)}\right) \quad 0 < i < P$$

**Maximum** For each pair of slices in the set, evaluate the percentage of program points that are in both. The maximum value is the largest value among all pairwise comparisons.

$$100 \cdot Max\left(\frac{\cap(S_i, S_j)}{Max(S_i, S_j)}\right) \quad 0 < i \neq j < P$$

With any definition of properties of interest, a mechanism is needed to map properties onto overall fitness values. For multiple objective problems, one simple technique
for combining \( n \) values for a fitness value: \( \text{Property}_1, \ldots, \text{Property}_n \) is to combine them into a single fitness value using corresponding ‘weights’ \( K_1, \ldots, K_n \).

In the experiments reported upon here, two fitness functions are defined, implemented and experimented with (corresponding to the two choices for measurement of overlap):

\[
\text{Coverage} \cdot 0.5 + (100 - \text{Average}) \cdot 0.5 \quad (4.1)
\]

\[
\text{Coverage} \cdot 0.5 + (100 - \text{Maximum}) \cdot 0.5 \quad (4.2)
\]

In both cases the weights are set to 0.5 so that each of the two objectives of the two fitness functions is considered equal. Nevertheless, decision of the weights is optional, different weights for the coverage and overlap could be considered in terms of the specific needs. As an illustrative example of fitness here, equal weights are considered since there are no other evidence that the coverage is more dominant to the overlap and vice versa. Both formulations of fitness attempt to capture the decomposition of the program that maximises coverage while minimizing overlap. Future work will consider the variation of these weights and the exploration of the Pareto front of optimal solutions.

### 4.2 Search Algorithms

This section describes the 4 types of search algorithms used in the experiments reported upon in the study. The detailed description of these is given in algorithmic pseudo code in Figures 4.1, 4.2, 4.3 and 4.4.
4.2.1 Genetic Algorithm

A Genetic Algorithm (GA) [118] begins with a set of solutions (represented by chromosomes) called a population. Solutions from one population are used to form a new population. This is motivated by a hope that the new population will be better (according to the fitness function) than the old one. Solutions are selected to form new solutions (offspring) according to their fitness; the more suitable they are, the more chance they have to reproduce. This process is repeated over a series of ‘generations’ until some termination condition is satisfied. In the GA, the primary operations and parameters are as follows:

**Selection** Selection determines the chromosomes that are selected from the population to be parents for crossover, based on their fitness. There are many methods for selecting the best chromosomes such as roulette wheel, Boltzmann, tournament, rank and steady state [118]. The experiments reported upon in this experiment use the Elitism and Rank Selection method. The Elitism keep the best chromosome in the previous population. Rank Selection firstly ranks


**Hill Climbing Algorithm:**
Parameters: Max: Population × Generation (referring to parameters of GA); Sum: the calculation times of the fitness; S: the current solution; N: the neighbour of the current solution.

Begin
    Sum ← 0
    while (Sum <= Max) do
        begin
            initiate S randomly
            if (fitness of S < fitness of HC(S))
                S ← HC(S)
            end
        end
    End

HC(S)

i ← 0
while (true) do
    begin
        look for N(i)
        while (true) do
            begin
                if (Sum++ > Max) break ‘outside-while’ loop
                if (fitness of S < fitness of N(i))
                    S ← N(i)
                    i ← 0
                    break
                else if (i++ < the number of all the slices)
                    look for next neighbour of the current solution.
                    break ‘outside-while’ loop
                end
                i ← i + 1
            end
        end
    return current S

Figure 4.2: The Hill Climbing Algorithm Used in the Study
Greedy Algorithm:
Parameters: Initial Solution Set: \{0,...,0\}; Candidate Set: the set of all the slices of the program.

Begin

evaluate each slice of candidate set
while(not solution)
begin
select the slice
end
End

Figure 4.3: The Greedy Algorithm Used in the Study

Random Algorithm:
Parameters: Generation(G): 100; Individuals(I) (corresponding to the population in GA): 50.

Begin

\[ i \leftarrow 0 \]
while\(i < G\) do
begin
\[ t \leftarrow 0 \]
while\(t < I\) do
begin
initiate \(I(t)\) randomly
evaluate \(F(t)\)
\[ t \leftarrow t + 1 \]
end
\[ i \leftarrow i + 1 \]
end
End

Figure 4.4: The Random Algorithm Used in the Study
the population and then every chromosome receives a value from this ranking; The best has the value n (the number of chromosomes in population), second best n-1 etc. and the worst has the value 1; and then, put each chromosome on the wheel based on their values such that the bigger the value is, the more chance the chromosome will be chosen.

**Crossover and Crossover Probability** Crossover operates on selected genes (elements of chromosomes) from parent chromosomes to create new offspring. The likelihood that crossover will be performed is called Crossover Probability [118]. The experiments reported upon in this experiment use the method of multi-point crossover with a Crossover Probability of 0.8.

**Mutation and Mutation Probability** Mutation randomly changes the offspring resulting from crossover. The likelihood of mutation is called the Mutation Probability [118]. The experiments reported upon in this experiment use random bit flip with a Mutation Probability of 0.01.

### 4.2.2 Hill Climbing

A Hill-Climbing (HC) algorithm looks for the neighbour of a current solution and if the neighbor is better, this neighbour replaces the current solution. The operation will be repeated until no better neighbour can be found. The definition of neighbours is referred to as flipping an element of $S$ (defined in Section 3.2). In order to ensure fairness of comparison, the HC algorithm has the same budget of computation time as GA’s. That is, the experiments use multiple restart Hill-Climbing and allow the same number of fitness evaluations in total (over all hill climbs) as are allowed to other algorithms.
4.2.3 Greedy Algorithm

In general, a greedy algorithm consists of two sets and three main functions [95]:

**Solution Set** From which a solution is created.

**Candidate Set** Which represents all the possible elements that might compose the solution.

**Selection Function** Which chooses the most promising candidate to be added to the solution.

**Value-Computing Function** Which gives the value of a solution.

**Solution Function** Which checks whether a final solution has been reached.

In the experiments, the greedy algorithm can be described as follows:

- The initial solution set is a binary string with each bit set to 0 and all the slices make up the candidate set;

- The candidate set is all the slices based on each SDG node in the program.

- The value-computing function evaluates the number of program points of the current solution set;

- The selection function chooses the slice that has the best contribution to the coverage value of solution and the smallest overlap value, that is, the bigger the ratio of increment of coverage and increment of overlap, the more chance the slice is chosen;

- The solution function checks whether coverage value of current solution has covered the whole program points in the program. The greedy algorithm is a
heuristic algorithm and not a search algorithm, but its results can be compared to the others using the same fitness function.

4.2.4 Random Algorithm

The Random Algorithm (RA) generates the individuals (solutions) randomly. The purpose of using RA is to measure the performance of the other algorithms. Since a random search is unguided and therefore ‘unintelligent’, it would be hoped that the guided search approaches and the other algorithms would outperform it. The random algorithm is therefore included to provide a base line, below which performance of the other algorithms should not fall.

4.3 Empirical Study

The slicing data used in the empirical study was collected by constructing the set of possible backward slices (with Codesurfer) of each program’s System Dependence Graph (SDG) [64]. Slice size is measured by counting vertices of the dependence graph, rather than lines of code. The study concerns source code of six open source programs, written in C. The program sizes range from 34 to 1,008 program points. At first sight, the size of each program may seem relatively small. However, the problem complexity is determined by the number of sets of slices which ranges from $2^{34}$ to $2^{1008}$ which is a very large search space. Summary of information concerning the programs studied can be formed in Table 4.2.

4.3.1 Which algorithm is the best?

Each non-greedy algorithm was executed 100 times with randomly chosen initial values (thus effectively sampling the space of possible start points). This produces a set of 100 results, one for each execution. The results obtained for some particular
Figure 4.5: Box plot of results for backward slicing in term of Fitness Function 1 defined in Section 4.1.1. The results show that the greedy algorithm performs the best. The low variance for the Hill Climbing algorithm (HC) suggests either a low order of modality in the landscape or a multi-modal landscape with similar valued peaks.
Figure 4.6: Box plot results for backward slicing in term of Fitness Function 2 defined in section 4.1.1. The results confirm the result from Fitness Function 1 (presented in Figure 4.5) that the greedy algorithm performs the best. The programs are presented in increasing order of size (top-to-bottom, left-to-right), providing evidence that the gap in performance between the greedy algorithm and the others increases with program size. The low variance for the Hill Climbing algorithm (HC) also replicates the finding for Fitness Function 1.
execution is determined by the random seed. The population from which this sample of 100 execution comes, is thus the population of random seeds. For the greedy algorithm, the execution results are the same every time since the results are gained with ‘Greedy Strategy’, rather than the random initial population. Boxplots is used to show the distribution of the results. Boxplots can be useful to display differences between populations without making any assumptions of the underlying statistical distribution; graphically depicting groups of numerical data through their five-number summaries (the smallest observation, lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation) as shown in Figure 4.5 and Figure 4.6.

Under the first fitness function (results presented in Figure 4.5), the performance of the greedy algorithm is the best except for the smallest program sum. Moreover, it is observed that for the smaller programs (e.g. (a) and (b)) HC performs better than either GA or Random. As program size increases (e.g. (c)-(f)) the GA performs better, beating HC and Random. HC performs worse on the larger programs suggesting that the HC landscape is too flat to easily find maxima.

To determine the relative performance of these three algorithms (non greedy algorithms), the Mann-Whitney and Wilcoxon test is applied to every program. As mentioned above, 100 sample results from execution are gathered for each program. For each program, the difference between the set of samples for each algorithm is

Table 4.2: The subject programs studied.

<table>
<thead>
<tr>
<th>Programs</th>
<th>LoC</th>
<th>Program Points</th>
<th>Size of Search Space</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>20</td>
<td>34</td>
<td>$1.37 \times 10^{11}$</td>
<td>Numerical value calculation</td>
</tr>
<tr>
<td>Hello</td>
<td>43</td>
<td>76</td>
<td>$7.55 \times 10^{22}$</td>
<td>Simple program, but more complex than ‘hello world’</td>
</tr>
<tr>
<td>Informationflow</td>
<td>109</td>
<td>176</td>
<td>$9.57 \times 10^{52}$</td>
<td>Example of simple information processing</td>
</tr>
<tr>
<td>Acct</td>
<td>681</td>
<td>546</td>
<td>$2.30 \times 10^{164}$</td>
<td>Accounting package</td>
</tr>
<tr>
<td>Newton</td>
<td>819</td>
<td>998</td>
<td>$2.67 \times 10^{300}$</td>
<td>Interpolated polynomial that uses Newton’s method</td>
</tr>
<tr>
<td>Tss</td>
<td>896</td>
<td>1008</td>
<td>$2.74 \times 10^{303}$</td>
<td>Three kinds of mathematical interpolation function</td>
</tr>
<tr>
<td>Total</td>
<td>2,568</td>
<td>2,838</td>
<td>$2.74 \times 10^{303}$</td>
<td></td>
</tr>
</tbody>
</table>
significant due to Asymp. Sig. (2-tailed)=.000 and Exact Sig.(2-tailed)=.000) at 
\( p = .05 \). The \( p \) value represents the probability of obtaining a result at least as 
extreme as the one that is actually observed. Informally speaking, the \( p \) value is a 
probability with a value ranging from 0 to 1. In other words, it is the answer to 
a question – if the populations have the same mean, what is the probability that 
random sampling would lead to a difference? As for values Asymp. Sig. (2-tailed) 
and Exact Sig.(2-tailed), when both are less than .01, the difference between two 
populations is considered as statistical significance.

Under the second fitness function (results presented in Figure 4.6), similar char-
acteristics can be observed to the first fitness function. The greedy algorithm outper-
forms the others except for the smallest program \textit{sum}. The GA performs the best 
of the non-greedy algorithms and the HC algorithm does not improve on Random 
except in the smallest programs (a) and (b). In the same way, the Mann-Whitney 
and Wilcoxon test at \( p = .05 \) applied in each program finds that Asymp. Sig. (2-
tailed)=.000 and Exact Sig.(2-tailed)=.000 represent that the difference of values of 
GA, Random and HC is statistically significant in each program.

In summary, the GA performs better as program size increases with the HC algo-
rithm having the opposite characteristic. Random is beaten by GA in all programs 
but by the HC algorithm only in small programs. The greedy algorithm beats the 
other 3 algorithms in the most situations. Furthermore, Table 4.3 shows the execution 
time of each algorithm for each program, which suggests that the greedy algorithm 
also has the best performance among 4 algorithms.

### 4.3.2 How similar are the results for each algorithm?

This section presents two approaches to compare the results produced by each algo-

rithm for similarity. The first is a purely visual representation, used to provide visual 
evidence for similarity. The second is a quantitative assessment of the similarity of
Table 4.3: Execution time of each algorithm for each program in condition of the machine—Ram 512M; Pentium4 3.2GHz. F1 and F2 represent the fitness function (1) and (2) defined in section 4.1.1, respectively; the measurement is based upon the milli second.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Greedy</th>
<th>GA</th>
<th>HC</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F1</td>
<td>F2</td>
<td>F1</td>
<td>F2</td>
</tr>
<tr>
<td>Sum</td>
<td>11</td>
<td>11</td>
<td>15121</td>
<td>15640</td>
</tr>
<tr>
<td>Hello</td>
<td>15</td>
<td>15</td>
<td>15611</td>
<td>16471</td>
</tr>
<tr>
<td>Information-</td>
<td>20</td>
<td>20</td>
<td>16611</td>
<td>17019</td>
</tr>
<tr>
<td>Acct</td>
<td>46</td>
<td>46</td>
<td>30676</td>
<td>42007</td>
</tr>
<tr>
<td>Newton</td>
<td>139</td>
<td>139</td>
<td>109122</td>
<td>166521</td>
</tr>
<tr>
<td>Tss</td>
<td>140</td>
<td>140</td>
<td>119124</td>
<td>183091</td>
</tr>
</tbody>
</table>

The findings suggest that the algorithms find similar (though not identical) solutions. This level of agreement, coupled with the low variance in hill climbing results provides evidence that the landscape is either uni-modal or multi-modal but with many local optima of similar value to each other, with the result that it is possible for search algorithms to find solutions of reasonable quality.

**Qualitative Similarity Analysis**

Figure 4.7 to Figure 4.12 provide a visualization of the results of the search. The goal of visualizing the optimal slicing set is to display the result obtained. Thus, the figures show optimal slicing set with search algorithms, rather than the entire set of slices for all the program points.

The X axis represents the slicing criteria, ordered by their relative location in the source code: earlier source code locations appear closer to the left side. The Y axis represents the program points belonging to the corresponding slice. As can be seen from these figures, the results produced by each algorithm are strikingly similar (Especially for the `newton` and `tss`, the image appears almost the same) but not identical.

In the Figures 4.11 and 4.12, it can be seen that there is a greater degree of similarity in the three heuristic methods (greedy, GA and HC), while the random
algorithm appears to produce a rather less ‘coherent image’. On the other hand, the greedy algorithm is apt to finding the optimal slicing set which has less slices than the GA, HC and random algorithms (except for the sum due to small program points in the program – too tiny). That is, the greedy algorithm can find optimum solutions which have smallest slices in the slicing set, such that the decomposition of program has the simplest form.

Moreover, GA and HC perform better than the random algorithm, which can be observed from Figures 4.5 and 4.6 (also from the empirical study in section 4.3.1), since GA and HC always try to cover with the program points in the program as many as possible, whereas the random algorithm has the form of less coverage–less coherent image. However, GA sometimes finds the slicing set which has more overlap than three others. Of course, these observations are qualitative and of illustrative value only. The next subsection provides a quantitative similarity analysis.

**Quantitative Similarity Analysis**

Table 4.4 presents results concerning the quantity of agreement between each pair of results for each algorithm. The calculation used for this is: $100 \cdot \frac{\cap(A,B)}{\min(A,B)}$. This represents, as a percentage, the degree of agreement between two sets $A$ and $B$. If the sets are identical then agreement is 100%; if there is no intersection then agreement is 0%. The percentage agreement measures the size of the intersection between the two sets relative to the size of the smaller of the two. Therefore, it is a measure of the degree to which the maximum possible intersection size is achieved.

By comparing how many identical slices there are in each pair of slicing sets, it is possible to measure result similarity. That is, Table 4.4 indicates the level of agreement the different search techniques share as to their choices of optimal solution.
Figure 4.7: Visualized results for backward slicing based on Fitness Function 1 with the program \textit{sum}. The GA produces smallest results for this very small program.
Figure 4.8: Visualized results for backward slicing based on Fitness Function 1 with the program \textit{hello}. 

(a) Greedy (Fitness Value = 92) 

(b) GA (Fitness Value = 90) 

(c) HC (Fitness Value = 92) 

(d) Random (Fitness Value = 91)
Figure 4.9: Visualized results for backward slicing based on Fitness Function 1 with the program informationflow. Note that the greedy algorithm produces the best results and also achieves this with the fewest slices.
Figure 4.10: Visualized results for backward slicing based on Fitness Function 1 with the program acct. The greedy algorithm produces the best results with the fewest slices.
Figure 4.11: Visualized results for backward slicing based on Fitness Function 1 with the program newton. Note the image for the random search appears to be a ‘grainy’ version of that for the others and that the greedy algorithm result contains fewer slices. There are more similarity in the intelligent searches; random produces a ‘poor imitation’.
Figure 4.12: Visualized results for backward slicing based on Fitness Function 1 with the program \textit{tss}. Note the image for the random search appears to be a ‘grainy’ version of that for the others and that the greedy algorithm result contains fewer slices.
Table 4.4: Comparison of slicing sets between the programs of the source code, $100 \cdot \frac{\cap(A,B)}{\min(A,B)}$.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Greedy</th>
<th>Genetic</th>
<th>H-Climbing</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Program1 sum.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>N/A</td>
<td>40</td>
<td>60</td>
<td>71</td>
</tr>
<tr>
<td>Genetic</td>
<td>40</td>
<td>N/A</td>
<td>50</td>
<td>29</td>
</tr>
<tr>
<td>H-Climbing</td>
<td>60</td>
<td>50</td>
<td>N/A</td>
<td>57</td>
</tr>
<tr>
<td>Random</td>
<td>71</td>
<td>29</td>
<td>57</td>
<td>N/A</td>
</tr>
<tr>
<td>(b) Program2 hello.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>N/A</td>
<td>71</td>
<td>91</td>
<td>80</td>
</tr>
<tr>
<td>Genetic</td>
<td>71</td>
<td>N/A</td>
<td>79</td>
<td>71</td>
</tr>
<tr>
<td>H-Climbing</td>
<td>91</td>
<td>79</td>
<td>N/A</td>
<td>82</td>
</tr>
<tr>
<td>Random</td>
<td>80</td>
<td>71</td>
<td>82</td>
<td>N/A</td>
</tr>
<tr>
<td>(c) Program3 informationflow.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>N/A</td>
<td>32</td>
<td>41</td>
<td>38</td>
</tr>
<tr>
<td>Genetic</td>
<td>32</td>
<td>N/A</td>
<td>48</td>
<td>61</td>
</tr>
<tr>
<td>H-Climbing</td>
<td>41</td>
<td>48</td>
<td>N/A</td>
<td>59</td>
</tr>
<tr>
<td>Random</td>
<td>38</td>
<td>61</td>
<td>59</td>
<td>N/A</td>
</tr>
<tr>
<td>(d) Program4 acct.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>N/A</td>
<td>28</td>
<td>34</td>
<td>32</td>
</tr>
<tr>
<td>Genetic</td>
<td>28</td>
<td>N/A</td>
<td>53</td>
<td>55</td>
</tr>
<tr>
<td>H-Climbing</td>
<td>34</td>
<td>53</td>
<td>N/A</td>
<td>56</td>
</tr>
<tr>
<td>Random</td>
<td>32</td>
<td>55</td>
<td>56</td>
<td>N/A</td>
</tr>
<tr>
<td>(e) Program5 newton.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>N/A</td>
<td>54</td>
<td>66</td>
<td>57</td>
</tr>
<tr>
<td>Genetic</td>
<td>54</td>
<td>N/A</td>
<td>58</td>
<td>61</td>
</tr>
<tr>
<td>H-Climbing</td>
<td>56</td>
<td>58</td>
<td>N/A</td>
<td>59</td>
</tr>
<tr>
<td>Random</td>
<td>57</td>
<td>61</td>
<td>59</td>
<td>N/A</td>
</tr>
<tr>
<td>(f) Program6 tss.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>N/A</td>
<td>38</td>
<td>36</td>
<td>32</td>
</tr>
<tr>
<td>Genetic</td>
<td>38</td>
<td>N/A</td>
<td>49</td>
<td>57</td>
</tr>
<tr>
<td>H-Climbing</td>
<td>36</td>
<td>49</td>
<td>N/A</td>
<td>51</td>
</tr>
<tr>
<td>Random</td>
<td>32</td>
<td>57</td>
<td>51</td>
<td>N/A</td>
</tr>
</tbody>
</table>
4.3.3 Visual Evidence for the Presence for Clones

Clone detection is referred to as techniques to detect duplicated code in programs. Clone detection techniques have been widely investigated and can be roughly classified into three categories: string-based [40, 68], token-based [7, 69], parse-tree based [12, 75, 90], which have different performance with refactoring tools to remove duplicated code [107]. Moreover, Komondoor [73] and Krinke [80] use dependence to identify clone code.

The visualisation of results yields an unexpected but interesting finding related to the presence of clones. Notice the repeated patterns in Figures 4.14, 4.13, 4.16 and 4.15. There are two kinds of repeated patterns. The first kind are examples of sharing the same program points. In these patterns, the same vertical image is replicated across the X axis, for example, the middle section of Figure 4.10 (b) where the number of program points is between 200 and 300. This is an example of a situation where a whole series of slices share the same subset of nodes in their slices.

However, there are also some potentially more interesting repeated images. Those are not dependence clusters, because they do not share a set of y axis points. For example, consider the four blocks in Figures 4.13 and 4.15 (A, B, C and D). These images denote patterns of dependence that are repeated in different sections of the code. For instance, if one scans the program newton, the code related to similar blocks in Figure 4.13 is shown in the Figure 4.14. The corresponding four blocks of the code compute the four interpolated coefficients with the different inputs. For the code of tss as shown in Figure 4.16, the corresponding blocks A, B, C and D in Figure 4.15 represent 4 functions which compute three kinds of mathematical interpolation in different parameters, respectively. In each group of 4 blocks of code, a similar functionality emerges.

Inspection of the code quickly reveals that the four chunks of code are clones. However, they are not identical. Nonetheless, they have a similar dependency structure
which shows up in the visualisation. Because the search seeks to cover the program, these similar figures occur at different program points they tend to show up.

When these two groups of duplicate code are mapped to the visualisation of slicing sets shown in Figures 4.13 and 4.15 respectively, the corresponding blocks are denoted by a similar shape, which suggests the presence of clones. Consider the program Newton as an illustrative example. The program computes the outputs of four interpolated coefficients with the different inputs, and the computation of each coefficient is dependent on a corresponding block of code. The code for each of the four is very similar. The information can be captured with visualisation of slices of the set of some program points contributing to computation of the coefficients.

This is interesting and may suggest applications for search based slicing in clone detection. However, this remains a topic for future work, as clone detection is not the focus of the present experiment.
Figure 4.14: Clones present in the program newton
4.3.4 Flexibility of the Framework

This chapter introduces the general framework in which applying the Search Based Software Engineering theory to program slicing looks for dependency structures of interest in the source code. The thesis defined the fitness function that can decompose a program into a slicing set in which the overlaps might be minimum. This is only an illustrative example to demonstrate the possible application of the framework. In fact, search based slicing could have some other potential applications in source code analyses. The following section will introduce three feasible applications with this framework and researchers might define different fitness functions according to the specific purposes for different problems.
Search-Based Slicing

Decomposition of Programs

```
void TSS1(float arr_low1[5], float arr_up1[5], float d1[6], float arr_res1[6])
{
    int k;
    int arr_diag[6] = {2,2,2,2,2,2};
    float arr_q[6];
    float arr_y[6];
    float arr_p[6];
    arr_q[0] = arr_diag[0];
    arr_y[0] = d1[0];
    for(k = 2; k <= 6; k++) {
        arr_p[k-2] = arr_low1[k-2]/arr_q[k-2];
        arr_q[k-1] = arr_diag[k-1] - arr_p[k-2]*arr_up1[k-2];
        arr_y[k-1] = d1[k-1] - arr_p[k-2]*arr_y[k-2];
    }
    for(k = 6 - 2; k >= 0; k--) {
        arr_res1[k] = (arr_y[k] - arr_up1[k]*arr_res1[k+1]) / arr_q[k];
    }
    return;
}
```

```
void TSS2(float arr_low2[10], float arr_up2[10], float d2[10], float arr_res2[10])
{
    int k;
    int arr_diag[11] = {2,2,2,2,2,2,2,2,2,2,2};
    float arr_q[11];
    float arr_y[11];
    float arr_p[10];
    arr_q[0] = arr_diag[0];
    arr_y[0] = d2[0];
    for(k = 2; k <= 11; k++) {
        arr_p[k-2] = arr_low2[k-2]/arr_q[k-2];
        arr_q[k-1] = arr_diag[k-1] - arr_p[k-2]*arr_up2[k-2];
        arr_y[k-1] = d2[k-1] - arr_p[k-2]*arr_y[k-2];
    }
    for(k = 11 - 2; k >= 0; k--) {
        arr_res2[k] = (arr_y[k] - arr_up2[k]*arr_res2[k+1]) / arr_q[k];
    }
    return;
}
```

```
void TSS3(float arr_low3[15], float arr_up3[15], float d3[16], float arr_res3[16])
{
    int k;
    int arr_diag[16] = {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
    float arr_q[16];
    float arr_y[16];
    float arr_p[15];
    arr_q[0] = arr_diag[0];
    arr_y[0] = d3[0];
    for(k = 2; k <= 16; k++) {
        arr_p[k-2] = arr_low3[k-2]/arr_q[k-2];
        arr_q[k-1] = arr_diag[k-1] - arr_p[k-2]*arr_up3[k-2];
        arr_y[k-1] = d3[k-1] - arr_p[k-2]*arr_y[k-2];
    }
    for(k = 16 - 2; k >= 0; k--) {
        arr_res3[k] = (arr_y[k] - arr_up3[k]*arr_res3[k+1]) / arr_q[k];
    }
    return;
}
```

```
void TSS4(float arr_low4[20], float arr_up4[20], float d4[21], float arr_res4[21])
{
    int k;
    int arr_diag[21] = {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
    float arr_q[21];
    float arr_y[21];
    float arr_p[20];
    arr_q[0] = arr_diag[0];
    arr_y[0] = d4[0];
    for(k = 2; k <= 21; k++) {
        arr_p[k-2] = arr_low4[k-2]/arr_q[k-2];
        arr_q[k-1] = arr_diag[k-1] - arr_p[k-2]*arr_up4[k-2];
        arr_y[k-1] = d4[k-1] - arr_p[k-2]*arr_y[k-2];
    }
    for(k = 21 - 2; k >= 0; k--) {
        arr_res4[k] = (arr_y[k] - arr_up4[k]*arr_res4[k+1]) / arr_q[k];
    }
    return;
}
```

(a) Block A

(b) Block B

(c) Block C

(d) Block D

Figure 4.16: Clone code in tss
Figure 4.17: Four classifications of function cohesion. The A, B and C represent the processing elements; A1, A2 and A3 represent the processing elements are all in the same control block such as if, for or while.

Splitting/Refactoring Functions/Procedures to Improve Cohesion

In general, a function (or procedure) in a program independently computes one or many results and returns outputs by defining some processing elements [101]. Functions have different cohesion levels, which determine the readability, testability, and maintainability of software in terms of the relationship between these processing elements. High cohesion is usually considered to be desirable [21, 22]. According to the definition of Ott and Thuss [101], cohesion levels can be divided into 4 classifications: low, control, data and function as depicted in Figure 4.17.

The goal of splitting a function is to reconstruct the original function which has the lower cohesion into the set of subfunctions which all have higher cohesion, without changing the original semantics. The hope is that each smaller function is more reusable and robust.

The low level suggests several distinct unrelated processing elements; the control level case is similar to the low level except that processing elements is all dependent on some control statements. In this situation, search approaches can find several sets of slices, each of which represents a processing element, thereby in condition of no overlap or minimum overlap (or only control statements) allowing the function to be split into subfunctions, hopefully with higher cohesion levels. The results of splitting the function are shown in Figure 4.18 (a) and (b). For the data level, there are two cases. Figure 4.17 (c) suggests that the computation of the processing elements B
Figure 4.18: The results of splitting or refactoring the functions. En and Re represent subfunction entry and return value, respectively; Fa and Fb represent function calling sites to subfunctions of processing element A and B, respectively. Note that the new subfunctions in (b) all include control statements in which block A1, A2 and A3 belong.

and C depend upon the results of the element A. This function also can be divided into two subfunctions by putting the element A into each subfunction computing B and C in the Figure 4.18 (c).

On the other hand, The Figure 4.17 (d) and (e) show 'non-split-function' cases. However, search approaches can look for several sets of slices corresponding to the processing elements A, B in the subfigure (d) and A in the subfigure (e). In this situation, a set/sets of slices representing the processing element A and/or B can be extracted from the functions into a new subfunction and position of the A and/or B can be replaced by a function call. The results of refactoring the function are shown in Figure 4.18 (d) and (e).

**Parallelism Computation**

“Parallelizability can be measured as the number of slices which have a pair-wise overlap less than a certain threshold. A high degree of parallelizability would suggest that assigning a processor to execute each slice in parallel could give a significant program speed-up” [117].

Search approaches can seek to find specific combinations of slices which can reach such a threshold. Therefore, the fitness function can be described as:

“Seek to search for a set of slices $\text{Slice}_1, ..., \text{Slice}_n$, in which the overlap of
each pairwise is all less than the parameter of parallelism (to be defined in terms of the specific problem), such that the influences among the slices are least when each slice is given a separate processor.”

‘The Chain of Slices’ for Program Comprehension

Normally, some slices in a program have overlaps, other than complete 'independence'. Especially, some slices could completely include the others or some slices are identical, for example, in the same dependence cluster [16]. Therefore, search approaches could look for several sets of slices, in each of which bigger slices can cover the smaller one, which is like a set of a chain of slices.

Therefore, the fitness function can be described as:

“Seek to search for a set of sets of slices \{S_1, ..., S_n\} containing the whole program, in which \(S_1, ..., S_n\) represents the set of the slices \(Slice_1, ..., Slice_m\) which have this kind of relationship: \(Slice_i \subset Slice_{i+1}\). That is:
\[
\forall x \in Slice_i \Rightarrow x \in Slice_{i+1} \quad (0 < i < m)
\]

This idea is to find out some inclusive relationship amongst slices such that it can contribute to comprehension for understanding the program. When maintainers want to understand the program developed by other programmers, ‘the chain’ is helpful for comprehending the program structure step by step by understanding a program from the smallest slice to the biggest one.

4.4 Empirical Study with Greedy Algorithm

In the previous experiments in Section 4.3, 6 programs selected are relatively small since the purpose is to demonstrate that the SBSE framework can be applied to the program slicing technique to locate dependency structures of interest in source code. At the same time, it was found that the greedy algorithm is the best in the 4
algorithms in terms of fitness function, execution time and the number of the chosen slices in the solution found.

The further empirical study applies the greedy algorithm to the 6 larger C programs shown in Table 4.5, which are all open source taken from the ‘Gnu’ website (ftp://ftp.gnu.org/gnu). The number of vertices are all the program points in the SDG [63] and the number of slices are based upon contributions only from source code since, for the decomposition problem, the real concern is focused on the program exclusive of lib files and other program points from Codesurfer representations. The purpose is to find out how efficiently the greedy algorithm can decompose the program into a set of slices. In Table 4.6, ‘Percentage’ shows the percentage of the local optimal set of slices to the whole set of slices, and ‘Execution time’ shows running time to decompose the whole program. On the other hand, for each program, each function is decomposed into a set of slices with the greedy algorithm in order to explore the performance of the greedy algorithm on decomposing the functions. Figure 4.19 shows the frequency distributions of the percentage of the functions decomposed in each program.

As shown in the Figure 4.19, the majority of the decomposed functions falls into the range where the percentage is lower than 25%, and many of them are in between 6% and 20% with few beyond 50%. This suggests that less than one fifth of a program points can usually be used to decompose the whole program/function. Moreover, in Table 4.6, percentage for the whole program also suggests that less than 20% of the program points can decompose the whole program.

<table>
<thead>
<tr>
<th>Programs</th>
<th>termutils2.0</th>
<th>acct6.3</th>
<th>space</th>
<th>oracolo2</th>
<th>byacc1.9</th>
<th>a2ps4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (Loc)</td>
<td>6697</td>
<td>9536</td>
<td>9126</td>
<td>14326</td>
<td>6337</td>
<td>42600</td>
</tr>
<tr>
<td>Number of vertices</td>
<td>11037</td>
<td>21382</td>
<td>20556</td>
<td>20551</td>
<td>33022</td>
<td>43141</td>
</tr>
<tr>
<td>Number of slices</td>
<td>2952</td>
<td>5305</td>
<td>9887</td>
<td>8776</td>
<td>8046</td>
<td>17226</td>
</tr>
<tr>
<td>Number of functions</td>
<td>56</td>
<td>88</td>
<td>137</td>
<td>135</td>
<td>178</td>
<td>248</td>
</tr>
</tbody>
</table>

Table 4.5: Program descriptions.
Figure 4.19: Frequency distributions of percentage of decomposed functions. The X axis is the percentage of the number of the optimal set of slices to all the slices in a function; the Y axis is the number of decomposed functions which ‘drop in’ the corresponding percentage.
Table 4.6: Percentage and execution time. Percentage is the ratio between the optimal set of slices to the set of all the slices of the whole program; execution time measured in seconds was obtained from execution of decomposing all the functions on a machine, Pentium4 3.2GHz with Ram 512M.

In this empirical study, non-greedy algorithms are not considered since it was found earlier that the greedy algorithm outperforms the others. On the other hand, there are scalability issues for the GA, HC and random algorithms with large programs. For instance, the experiments implemented the 4 algorithms with the program `space`, which is very popular source code and has 9887 program points corresponding to the source code. Except for the greedy algorithm which can calculate in 126.4 seconds, the running time with the others roughly takes 1 minute for computing a fitness function value. That is, for both the population and generation equal to 100, a running time will take about 10,000 minutes, which, thus, causes the difficulty in a reasonable time to get boxplots to analyse the distribution of the results.

In fact, there may exist some redundancy, that is, some slices are as same as others or completely included with others. In this situation, search algorithms will take time to ‘filter’ the slices which will not contribute the fitness. The redundant issue can be addressed in the following situations.

**with the greedy:** Firstly, as for slices included by others, the algorithm select a slice, and then it would not select the other slices which are included by this slice. Secondly, with some slices which have a lot of overlaps with chosen slices, rather than the above situation, the greedy algorithm will determine whether to select some slice or not, based upon the ‘greedy strategy’. Therefore, every slice chosen by the algorithm is one that will in the most extent contribute to
Increasing of the value of fitness. That is, ‘greedy strategy’ will not select the slices which are redundant to skew the results.

**with the GA and HC:** Even though there are many same or ‘similar’ slices which are selected as the first initial generation, through the crossover and mutation, or looking for the neighbours, fitness function value will be improved. That is, the redundancy will be ‘filtered’ if the algorithms can make sure that enough generations are run.

**with the random:** Redundance situation could affect the algorithm performance. However, the random is only considered as a standard to check the performance of the other algorithms, rather than a ‘real intelligent’ search approach.

Thus, this situation will not affect the value of fitness function with the greedy algorithm, GA and HC, but will affect calculation time of fitness function, especially when there are a lot of redundancy slices in the program.

Therefore, the execution time of algorithms could be sped up by reducing the search space before applying search algorithms. Table 4.7 lists the percentage of slices redundancy of 12 programs in the experiments. Redundancy is defined as the ratio of the number of slices which are included in others to the number of all the slices of the program with coefficient 100. As shown in the figure, redundancy in the 12 programs is universal. It is suggested that execution time could be improved by reducing the redundancy in the source code. This is referred to as the further work.

<table>
<thead>
<tr>
<th>Programs</th>
<th>sum</th>
<th>hello</th>
<th>informationflow</th>
<th>acct</th>
<th>newton</th>
<th>tss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage</td>
<td>3.57</td>
<td>2.17</td>
<td>21.54</td>
<td>26.52</td>
<td>41.98</td>
<td>42.76</td>
</tr>
<tr>
<td>Programs</td>
<td>termutils-2.0</td>
<td>acct-6.3</td>
<td>space</td>
<td>oracolo2</td>
<td>byacc1.9</td>
<td>a2ps-4.1</td>
</tr>
<tr>
<td>Percentage</td>
<td>43.1</td>
<td>45.2</td>
<td>40.5</td>
<td>40.9</td>
<td>44.9</td>
<td>48.7</td>
</tr>
</tbody>
</table>

Table 4.7: Percentage of redundancy slices.
4.5 Summary

This study has introduced a general framework for search based slicing, in which the principles of Search Based Software Engineering are used to formulate a problem of locating dependency structures as a search problem.

The study presented results from an instantiation of this general framework of Search Based Slicing, for the problem of program decomposition, presenting the results of a case study that evaluated the application of greedy, hill climbing, genetic and random algorithms for both performance and similarity of results. Based upon the greedy algorithm, the best of 4 algorithms, further empirical study was formed to explore how efficiently large programs and single functions can be decomposed.

The results indicated that the algorithms produce relatively consistent results and that the greedy algorithm outperforms its rivals. The results also provided evidence that the landscape for the problem is either of low modality or is multi-modal, but with similarly valued peaks. The results are encouraging, because they suggest that it is possible to formulate dependency analysis problems as search problems and to find good solutions in reasonable time using this approach.
Chapter 5

Analysis of Procedure Splitability *

This chapter introduces a greedy dependence-based procedure splitting algorithm that provides automated support for analysis and intervention where procedures show signs of poor structure and over large size. The chapter also reports on the algorithm, implementation and empirical evaluation of procedure splitability. The study reveals a surprising prevalence of splitable procedures and a strong correlation between procedure size and splitability.

5.1 Problem Description

There is evidence that as programs evolve their structure deteriorates [83, 84]. One way that this program degradation manifests itself is that programs become ‘bloated’. That is, systems have a tendency to accrue additional functionality and the code that goes with it. In this chapter the author explores this question of bloating at the procedure level. The primary question that the thesis addresses is: is there a link between procedure size and splitability?

*This chapter is based on the work published on 15th Working Conference on Reverse Engineering (WCRE 2008) [67]
some work on extracting a single sub-procedure [82], this question has not previously been addressed. Clearly, more splitable procedures are candidates for re-engineering and so a positive answer to the question, though perhaps a further ‘cloud’ over the structural integrity of large-procedure systems, may contain a ‘silver lining’ of good news for the reverse and re-engineering community. That is, for the re-engineering community a positive answer to the central question would suggest that re-engineering becomes increasingly effective as procedure size increases.

Previous work on this problem has focussed on cohesion improvement and procedure extraction. For example, tucking [82] extracts a part of a procedure to make a new sub-procedure with the aim of improving cohesion, while procedure extraction, extracts marked code into a procedure, potentially improving the procedural abstraction of un-modularised code blocks [57, 74].

This chapter adopts a similar approach to tucking, except that a procedure may be split into several sub-procedures, rather than merely splitting in two, which tucking does. The approach also uses a greedy optimization algorithm to find the optimal or near optimal points for a split, whereas tucking simply follows a set of pre-determined rules to locate split (or wedge) points.

Following an approach inspired by Bieman and Ott’s slice based measurement of cohesion [14], the thesis defines measurements of splitability; the degree to which a procedure may be split without replication of any of the code in its body. The thesis then uses an implementation of the greedy splitting algorithm to empirically investigate splitability and its correlation to procedure size.

The thesis addresses three research questions RQ1-RQ3 to evaluate both the splitting technique and the splitability of code:

**RQ1:** How frequently are there splitable procedures in the programs studied?

**RQ2:** How much repeated code will need to be generated as a result of splitting. A perfect split requires no repetition. The more repetition is required, the less
splitable a procedure is.

**RQ3:** What is the relationship between splitability and the size of a procedure?

The central finding lies in the answer to the last question. The thesis demonstrates that there is, indeed, a statistically significant correlation between increasing procedure size and increasing splitability. This is good news for re-engineers who may naturally consider large procedures as starting candidates for re-engineering activities.

To answer these questions, the thesis presents three related empirical investigations. The investigations are performed on six different open source C programs (See Table 5.2). The primary contributions of the chapter are:

1. An algorithm based on greedy application of dependency analysis is introduced to identify split points in procedures.

2. The empirical study shows that a surprisingly large proportion of the procedures considered are splitable.

3. The thesis provides evidence for a strong statistical correlation between procedure size and splitability in the programs studied.

The next section, Section 5.2, defines splitable procedures and introduces the splitting algorithm. Section 5.3 presents the empirical study and answers the research questions posed in Section 5.1. Section 5.4 depicts thread to validity whilst Section 5.5 draws conclusions.

### 5.2 The Splitting Algorithm

Whether or not a procedure is apt to being split depends on the structure of the procedure. The possibility of splitting is increased if components of the procedure
have little in common. The attribute of splitting (or splitability) is closely related to that of cohesion. Procedures are more inclined to split the lower their cohesion level is. This observation is inspired by Bieman and Ott [14] who used slicing as a mechanism to evaluate cohesion.

The Codesurfer scripting language based on Scheme-STK [54] is used to process program code represented as a System Dependence Graph SDG [63] and to calculate the slices. Slicing is performed on every SDG node, and, therefore, for a procedure of \( n \) nodes, the search space for procedure splitting consists of \( 2^n - 1 \) subsets of slices.

Next, the concept of splitable procedures is introduced:

**Definition 1** \( \text{maximal}(s) \)

For a procedure \( P \), let \( \text{Slices}(P) \) be all the backward slices of \( P \) based on each SDG node as a slicing criterion. A slice \( s \) which is not included in any other slices is called a maximal slice. Thus, \( s \) is \( \text{maximal}(s) \) \( \iff \forall s', s' \neq s \Rightarrow s \nsubseteq s' \)

**Definition 2** Splitable Procedure

A procedure \( P \) is splitable if and only if there exist at least \( s_1, s_2 \in \text{Slices}(P) \land s_1 \neq s_2 \land \text{maximal}(s_1) \land \text{maximal}(s_2) \)

The purpose of applying the optimisation-based slicing approach advocated here is to identify split points automatically. This chapter uses a greedy algorithm [95] to tackle this problem. The algorithm searches for a set of slices representing procedure components that make the overlap of these slices as small as possible.

The purpose of the splitting algorithm is to locate the best (or an acceptably good) solution among a number of possible solutions in the search space. The process of looking for a solution is equivalent to that of looking for a set of slices in the search space with minimal overlap among the slices. Clearly, enumeration will not be possible since the search space grows exponentially in \( n \), the size of the program.
Though this is an exponentially large search space, the underlying optimisation problem is a set cover problem. Set cover problems submit to optimisation using greedy-based algorithms. That is, the greedy algorithm is known to produce solutions within a log of the global optimum for set cover [46, 104].

In employing the greedy algorithm presented in Figure 5.1, a slice is represented as a sequence of binary digits in a two-dimensional matrix, where columns correspond to SDG nodes and rows the slices. Here, a “1” and “0” correspond to whether a SDG node belongs or does not belong to the slice, respectively. In the same figure, \textit{groups}[N] represents a set of slices corresponding to a sub-procedure.

The splitting algorithm proceeds as follows:

1. Construct all the static backward slices in a procedure by slicing with respect to all the SDG nodes in the corresponding SDG.

2. Find sets of slices representing procedure components that make the overlap of these slices as small as possible. A greedy algorithm is used to construct such a minimal overlap set is described in Figure 5.1.

3. Recover slice statements from the corresponding SDG nodes by combining nodes that belong to a single statement. An example illustrating such combination is shown in Figure 5.3 where the SDG nodes of a statement and their dependency relationships are depicted.

4. Make the sub-procedures obtained executable.

One can always split a procedure into a set of slices where there are ‘useless’ slices that are totally included within others. Thus, to obtain useful and independent parts, \textit{maximal}(s) is defined and the greedy dependence-based splitting algorithm is introduced to split procedures according to Definition 2.
// The greedy algorithm
1 slices[N]; Array of Array of [0,1] sequence, slices[i] is a sequence of [0,1] and corresponds to node i as a slicing criterion.
2 groups[N]; Set of slices, a groups[i] represents Sub-procedure i.
3 count = 1; Integer, represents the number of sub-procedures.
4 sort slices according to their sizes in descending order;
5 groups[1] = {slices[1]};
6 for I := 2 to N do
7    if slices[I] ∈ groups[count] then continue;
8    else
9       count++;
10      groups[count] = slices[I];
11 end
12 end

Figure 5.1: Procedure splitting algorithm

Sum_and_Product ((1) int N, (2) int Sum, (3) int Prod){
    (4) int I;
    (5) Sum=0;
    (6) Prod=1;
    (7) for(I = 1; I <= N; I ++){
        (8) Sum=Sum+I;
        (9) Prod=Prod*I;}
}

Figure 5.2: An example for the splitting algorithm. The digits represent the SDG nodes of the procedure.

gamma = asin(expression)

Figure 5.3: An example of recovering the statement gamma = asin(sin(alpha) * b) from the corresponding six SDG nodes. Solid lines and dashed lines represent control and data dependency, respectively. The SDG node type (call-site, actual-in and expression) is shown under the SDG nodes.

120
In an illustrative example presented in Figure 5.2, Procedure *Sum_and_Product* includes two computations for the variables *Sum* and *Prod*. For simplicity, the example of Figure 5.2 only includes statements that correspond to single SDG nodes. In general, a statement may have several SDG nodes as explained in Figure 5.3. Table 5.1 shows all the backward slices for each SDG node of the procedure. The splitting algorithm is to find a set of maximal slices, each of which represents an independent sub-procedure. In terms of Definition 1, *slice8* and *slice9* are two maximal slices and the rest are not, due to the fact that *slice2* and *slice5* are included in *slice8*, *slice3* and *slice6* are included in *slice9* and *slice1*, *slice4* and *slice7* are included in both. In fact, *maximal(s)* represents useful independent computation in a procedure.

In the procedure shown in Table 5.1 and the splitting algorithm in Figure 5.1, *slices[N]* represents 9 slices of the procedure; *groups[N]* represents a set of maximal slices whilst *count* represents the number of maximal slices with initial value being 1. After sorting, the first maximal slice is *slice8* assigned to *groups[1]*. The *for* loop is to find out more maximal slices if they exist. In this case, *slice9* is another maximal slice.

<table>
<thead>
<tr>
<th>Program slices</th>
<th>SDG nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>slice1</td>
<td>1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>slice2</td>
<td>0 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>slice3</td>
<td>0 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>slice4</td>
<td>0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>slice5</td>
<td>0 1 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>slice6</td>
<td>0 0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>slice7</td>
<td>1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>slice8</td>
<td>1 1 0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>slice9</td>
<td>1 0 1 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

Table 5.1: The backward slices of the splitable procedure in Figure 5.2. A 1 represents a SDG node that is included in the slice, while a 0 represents a SDG node that is not included in the slice.
Semantic Constraints

In the section above, the greedy-based splitting algorithm recognises the splitting points in terms of the dependency relationship. In this way, the results will obtain a set of sub-procedures which may compromise the semantic purposes of programmers. For example, one defines a ‘struct’ data structure in a procedure and computes each element in the ‘struct’ with a set of statements. Essentially, each set of statements represents a slice (where the slicing criterion is the variable of each element in the ‘struct’). The splitting algorithm may split the procedure into a set of slices each of which is a sub-procedure. However, this will cause the sub-procedures to be less meaningful since all the elements together constitute the entire ‘struct’.

```c
int grphaexc(struct charac * p1, struct charac ** pp2, struct GrPhaExc ** grphaexc_ptr)
{
    Variable Declerations and Initializations
    1 *curr_ptr = p1;
    2 error = 0;
    3 error = (GetKeyword(Keywords[43], curr_ptr));
    4 if (error != 0) {
        5 *grphaexc_ptr = NULL;
        6 return 1;
    }
    7 error = unifpha(*curr_ptr, curr_ptr, unif_val_ptr, phase_unit_ptr);
    8 if (error == 0) {
        9 *grphaexc_ptr = (struct GrPhaExc*) malloc(sizeof(struct GrPhaExc));
        10 if (*grphaexc_ptr == NULL) {
            11 parserro(*curr_ptr, 55, “ ”);
            12 interror("grphaexc()");
        }
        13 (*grphaexc_ptr)->TYPE = UNIFORM_LAW;
        14 (*grphaexc_ptr)->UNIF_VAL = unif_val;
        15 (*grphaexc_ptr)->PHASE_UNIT = phase_unit;
        16 (*grphaexc_ptr)->PQEXP_PTR = NULL;
        17 *pp2 = *curr_ptr;
        18 return 0;
    }
```

Figure 5.4: An example of the ‘struct’ semantic constraint. The source code is from the procedure grphaexc in the project space at the ‘GNU’ website.

Figure 5.4 shows an illustrative example where there are the computations of
a ‘struct’ data structure *grphaexc.ptr. Lines 13-16 are the statements which assign the final values for each element in *grphaexc.ptr. The algorithm will consider the computation of each element variable – TYPE, UNIF.VAL, PHASE_UNIT, PQEXP_PTR as a separate sub-procedure without the semantic constraint.

In the implementation, the thesis defines two semantic constraints to make sub-procedures more meaningful. Of course, these two constraints do not exhaust all the situations which can completely make the sub-procedures perfectly meaningful. However, these sub-procedures may be a set of candidates for reusability or refactoring. Formally, the two constraints’ definitions are defined as follows:

**Definition 3 Struct Constraint**

Given a ‘struct’ data structure, S and the elements, \{e_1, ..., e_n\}, if and only if slice(e_i) (i ≤ n) and slice(e_j) (j ≤ n) are maximal slices, both of these slices are combined into a single sub-procedure. slice(e_i) represents a slice which is sliced based on the variable e_i in a statement.

**Definition 4 Condition-Control Constraint**

Given two slices, slice_i and slice_j, if and only if both slices are maximal slices and control dependent on a predicate in the if or switch control statement, both of these slices are combined into a single sub-procedure.

Figure 5.4 also shows the Condition-Control Constraint. Since slice(*pp2) in line 17 and slice(TYPE) in line 13 are both controlled by the control variable error in line 8, these two slices are merged into one sub-procedure. The reason for this constraint is because the thesis considers that this kind of control variable makes ‘inside-control-block’ slices more ‘cohesive’. Again, the thesis needs to point out that this kind of semantic constraint is flexible and can be defined in different ways according to the specific issues.
5.3 Empirical Study

The algorithm explores procedure structures automatically and provides candidates for splitting. In this section, procedure structures and relationships between procedure components of six open source systems are analysed. The structures and relationships are evaluated in terms of code overlap, splitability and the correlation between procedure size and splitability.

5.3.1 RQ1: How frequently are there splits?

It is not always the case that a procedure can be split into two or more sub-procedures. This section explores the frequency of occurrence of splitable and non-splitable procedures.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Description</th>
<th>Size</th>
<th>No of Procs</th>
<th>Split Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Loc</td>
<td>Ver</td>
<td>Count</td>
</tr>
<tr>
<td>termutils2.0</td>
<td>terminal control utilities</td>
<td>4,334</td>
<td>2,952</td>
<td>56</td>
</tr>
<tr>
<td>acct6.3</td>
<td>accounting utilities package</td>
<td>6,178</td>
<td>4,305</td>
<td>88</td>
</tr>
<tr>
<td>space</td>
<td>ESA ADL interpreter</td>
<td>9,106</td>
<td>9,887</td>
<td>137</td>
</tr>
<tr>
<td>oracolo2</td>
<td>an array processor</td>
<td>14,326</td>
<td>8,776</td>
<td>135</td>
</tr>
<tr>
<td>byacc1.9</td>
<td>LALR parser generator</td>
<td>6,420</td>
<td>8,046</td>
<td>178</td>
</tr>
<tr>
<td>a2ps4.1</td>
<td>Text to postscript converter</td>
<td>20,407</td>
<td>17,226</td>
<td>248</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>60,771</td>
<td>51,192</td>
<td>842</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>10,129</td>
<td>8,532</td>
<td>141</td>
</tr>
</tbody>
</table>

Table 5.2: The set of programs studied. Entry ‘Loc’ represents the lines of code; ‘Descriptions’ for the functionality of programs; ‘Ver’ for the number of SDG nodes corresponding to source code; ‘No of Procs’ for the total number of procedures in the entire program; ‘Split pros’ represents how many procedures in the program can be split into two or more; ‘Count’ for the number of procedures which can be split and ‘Perc’ for the percentage of ‘Count’ to ‘No of Procs’.

Table 5.2 provides information about how many procedures in the program can be split and the percentage of this to the total number of procedures. The results indicate that more than 20% of procedures (average 23.6%) in each of the six programs include independent procedure components contributing to the whole functionality.
Figure 5.5: Distribution of splitable procedures. A column at the horizontal position $x$ of the height $y$ means that $y/N\%$ of procedures are split into $x$ sub-procedures, where $N$ is the total number of procedures.

Figure 5.5 shows the percentage of frequency distributions of procedures in terms of the number of sub-procedures obtained by splitting. Each column in the figure represents the percentage of the number of procedures that can be split to the total number of procedures in the entire program. Here, the $x$-axis entries represent the number of sub-procedures obtained as a result of splitting, where 1 represents non-splitable procedures. All figures show a maximum split level of 4. However, (b), (c), (e) and (f) have a very few exceptional procedures that can be split into 9, 9, 6 and 13 sub-procedures respectively. Fewer than 1% of the procedures can be split into more than 4 sub-procedures so, for clarity, they are not shown.

The results depicted in Figure 5.5 show that the majority of procedures falls into the first column indicating that most procedures are not splitable. As far as procedure cohesion is concerned, it is preferable that a procedure only completes one single task, thus making programs more efficient and maintainable [22, 34, 101]. However, in
some cases, procedures can be split into two or more sub-procedures. This suggests possible ‘granularity of modularisation’ issues; perhaps procedures should be split to aid on-going maintainability.

5.3.2 RQ2: Splitability of Procedures

This section analyses the dependency amongst sub-procedures of each splitable procedure and explores the overlap and splitability distributions. To this end, code overlap and splitability measures are defined in terms of the sub-procedures’ sizes.

Since, in general, different procedure components are not completely independent, sub-procedures might share some common SDG nodes whose computation contributes to all the sub-procedures. In this case, there is extra repeated code generated and shared between different sub-procedures. As can be seen from Figure 5.5, the majority of splitable procedures are split into two or three sub-procedures, to be referred to as 2-way and 3-way splitable procedures, respectively. In what follows, the chapter concentrates on this group of splitable procedures, disregarding those that could potentially be split into more than three sub-procedures.

Depending on the size of the overlap or repeated code common to all sub-procedures, different levels of splitability can be defined. Evaluating repeated code shared by all sub-procedures suggests a measure of dependency between sub-procedures. Splitability decreases as dependency between sub-procedures increases, because too many repeated SDG nodes imply high inter-dependency relatedness between different procedure components. The splitability measure expresses how tightly different procedure components are related. High splitability indicates that procedures tend to compute more than a single and independent functionality and vice versa.

In order to define procedure splitability as a quantifiable measure, the following definitions are required. These definitions capture different metrics for assessing splitability. Let $S_i$ and $S_j$ denote two sub-procedures of a splitable procedure,
Size($S_i$) and Size($S_j$) denote the sizes, i.e., the number of SDG nodes of $S_i$ and $S_j$, respectively. Further, let $Size(S_i \cap S_j)$ be the number of the SDG nodes shared by the two sub-procedures $S_i$ and $S_j$ and $\text{Max}(\text{Size}(S_i), \text{Size}(S_j))$ be the greater value of the two sizes.

**Definition 5** *Overlap between two sub-procedures of 2-way splittable procedures*

Overlap of 2-way splittable procedures, $\sigma^{12}$, is the ratio of the number of SDG nodes shared by the two sub-procedures to the number of SDG nodes of the sub-procedure of greater size,

$$\sigma^{12} = \frac{\text{Size}(S_1 \cap S_2)}{\text{Max}(\text{Size}(S_1), \text{Size}(S_2))} \quad (I)$$

**Definition 6** *Min-Overlap amongst three sub-procedures of 3-way splittable procedures*

Min-Overlap of 3-way splittable procedures, $\sigma_{\text{min}}^{123}$, is the ratio of the number of SDG nodes shared by the three sub-procedures to the number of SDG nodes of the sub-procedure of greatest size,

$$\sigma_{\text{min}}^{123} = \frac{\text{Size}(S_1 \cap S_2 \cap S_3)}{\text{Max}(\text{Size}(S_1), \text{Size}(S_2), \text{Size}(S_3))} \quad (II)$$

**Definition 7** *Max-Overlap amongst three sub-procedures of 3-way splittable procedures*

Max-Overlap of 3-way splittable procedures, $\sigma_{\text{max}}^{123}$, is the biggest Overlap of three pairs of sub-procedures. If

$$\sigma^{12} = \frac{\text{Size}(S_1 \cap S_2)}{\text{Max}(\text{Size}(S_1), \text{Size}(S_2))},$$

$$\sigma^{13} = \frac{\text{Size}(S_1 \cap S_3)}{\text{Max}(\text{Size}(S_1), \text{Size}(S_3))}, \text{ and}$$

$$\sigma^{23} = \frac{\text{Size}(S_2 \cap S_3)}{\text{Max}(\text{Size}(S_2), \text{Size}(S_3))}$$
then
\[ \sigma_{\text{max}}^{123} = \text{Max}(\sigma^{12}, \sigma^{13}, \sigma^{23}) \]  

Overlap represents the cohesive degree between sub-procedures, that is, the more overlap between sub-procedures is, the more cohesive a procedure is, and therefore, the less likely the procedure is splitable. Splitability is therefore measured as the opposite of overlap. Thus,

**Definition 8 Splitability between two sub-procedures of 2-way splitable procedures**

Splitability of a 2-way splitable procedure is defined as the opposite of Overlap of a procedure.

\[ S = 1 - \sigma^{12} \]  

**(IV)**

**Definition 9 Max-Splitability amongst three sub-procedures of 3-way splitable procedures**

Max-Splitability of a 3-way splitable procedure is defined as the opposite of Min-Overlap of a procedure.

\[ S_{\text{max}} = 1 - \sigma_{\text{min}}^{123} \]  

**(V)**

**Definition 10 Min-Splitability amongst three sub-procedures of 3-way splitable procedures**

Min-Splitability of a 3-way splitable procedure is defined as the opposite of Max-Overlap of a procedure.

\[ S_{\text{min}} = 1 - \sigma_{\text{max}}^{123} \]  

**(VI)**

Figure 5.6 shows overlap frequency distribution of splitable procedures that are split into two sub-procedures. The x-axis represents the range of the overlap as defined in (I). The y-axis represents the percentage of the number of the splitable procedures.
Figure 5.6: Overlap frequency distribution of splittable procedures that can be split into two sub-procedures. The $x$-axis measures overlap using $\sigma^{12}$ defined in (I). The $y$-axis represents the percentage of 2-way splittable procedures that lie in the corresponding overlap range.
which lie in the corresponding overlap range to the number of all the 2-way splitable procedures of each program.

The overlap distribution expresses dependency between two different procedure components. Locations closer to the origin indicate low inter-dependency implying high splitality. In this region, procedures can be split without generating a large amount of repeated code, i.e., their components share few common SDG nodes.

For locations away from the origin, splitality of procedures decreases since increasing overlap between two sub-procedures indicates high inter-dependency between sub-procedures. That is, the different components of a procedure share quite a few common SDG nodes. In this case, splitting generates a large amount of repeated code.

Turning to 3-way splitable procedures, there are two empirical investigations of procedure overlap corresponding to definitions (II) and (III). Figure 5.7 shows the overlap distribution for 3-way splitable procedures in terms of the overlap measure defined in (II). For low overlap ranges, splitable procedures consist of multi-functionality, where at least two sub-procedures are relatively independent and do not share many common SDG nodes. The distributions of the six programs considered in this study exhibit different behaviour.

Figure 5.8 shows the overlap distribution for 3-way splitable procedures corresponding to maximum overlap \( \sigma_{\text{max}}^{123} \) as defined in (III). This figure shows a different behaviour to that corresponding to minimum overlap of Figure 5.7. For low overlap ranges, splitable procedures consist of multi-functionality where all the three sub-procedures are relatively more independent of each other and all do not share many common SDG nodes.
Figure 5.7: Overlap frequency distribution of splitable procedures that can be split into three sub-procedures. The $x$-axis measures overlap using $\sigma_{123}^{\text{min}}$ defined in (II). The $y$-axis represents the percentage of 3-way splitable procedures that lie in the corresponding overlap range.

<table>
<thead>
<tr>
<th>Programs</th>
<th>termutils</th>
<th>acct</th>
<th>space</th>
<th>oracolo2</th>
<th>byacc</th>
<th>a2ps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spearman coefficient (Correlation)</td>
<td>0.81</td>
<td>0.73</td>
<td>0.82</td>
<td>0.73</td>
<td>0.83</td>
<td>0.92</td>
</tr>
<tr>
<td>Number of 2-way Splitable procedures (Ranks)</td>
<td>8</td>
<td>17</td>
<td>19</td>
<td>21</td>
<td>27</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 5.3: Correlation between procedure size and splitality of procedures where splitable procedures can be split into two sub-procedures in terms of $S$ defined in (IV).
Figure 5.8: Overlap frequency distribution of splitable procedures that can be split into three sub-procedures. The x-axis measures overlap using $\sigma_{123}^{\text{max}}$ defined in (III). The y-axis represents the percentage of 3-way procedures that lie in the corresponding overlap range.

Table 5.4: Correlation between procedure size and splitability of procedures where splitable procedures can be split into three sub-procedures in terms of $S_{\text{max}}$ defined in (V).
5.3.3 RQ3: Exploration of correlation between procedure size and splitability

This section explores the correlation between procedure size defined in terms of SDG nodes and splitability as defined in (IV), (V) and (VI). That is, how the size of a procedure is related to its splitability. Both 2-way splitable and 3-way splitable procedures are considered. Intuitively, procedures with relatively small size should turn out to be barely splitable; with increasing procedure size, the procedures would be expected to be more likely to split into more sub-procedures.

The Spearman correlation coefficient ($\rho$) is used here to investigate the relationship between a procedure’s size and its splitability.

Table 5.3 shows the first correlation between procedure size and splitability in terms of splitability definition (IV), where procedures are split into two sub-procedures. For all the six programs considered in the empirical study, except a2ps, $\rho$ lies between 0.7 and 0.9 indicating a strong correlation between procedure size and splitability. For a2ps, the $\rho$-value approaches 1 indicating a stronger correlation. This kind of strong or very strong correlation indicates that as procedure size increases, procedure’s splitability increases. The results studied show that, with increasing procedure size, there is a trend to 2-way splitability.

Table 5.4 shows the second correlation between procedure size and splitability in terms of splitability as defined in (V) where procedures are split into three sub-procedures. The $\rho$-values for termutils and a2ps lie between 0.0 and 0.2, which denotes very weak to negligible correlation. The $\rho$-value for space is between 0.2 and 0.4, which indicates weak or low correlation. In this case, there is no significant correlation between procedure size and splitability. However, the $\rho$-value for byacc lies between 0.9 and 1 indicating very strong correlation. For acct and oracolo2, since the number of ranks is too small, the $\rho$-value is not statistically significant.

Spearman Correlation data corresponding to Definition (VI) shows similar results.
to those of Table 5.4, i.e., there is no significant strong correlation between procedure size and splitability, except for byacc.

In summary, these results indicate that there is no consistent correlation between procedure size and procedure’s splitability into three sub-procedures, though there is for splitability into two sub-procedures.

5.4 Threats to validity

This section briefly outlines potential threats to the validity of the empirical study. In this study, the hypotheses of procedure splitability into two or more components together with the distribution of the resulting code overlap are studied. In addition, the relationship between procedure size and its splitability is investigated. Therefore, one issue to address is internal and external validity of the results presented, i.e., to check whether there has been a bias in the experimental design that could affect the collection of data and, consequently, the distributions and relationships under study.

One primary threat arises from faults of the tools used to collect data. Codesurfer scripting language was used for slicing. Since Codesurfer is a mature and widely used tool, this threat to internal validity is diluted.

Another source of bias comes from the selection of programs studied. That is, whether or not the collection of selected programs constitute a random sample. The selection has an impact on the extent to which it is possible to generalise from the results obtained. We note that, because of the rich and diverse nature of programs, it is impossible to sample a sufficiently large set of programs such that the full diversity of all possible programs could be captured. However, the study draws upon code from real world open source applications. Furthermore, the number of procedures considered is 842, providing a large pool of results from which to make observations.

With this study, the thesis does not claim that the results obtained show that the
hypotheses are valid for any set of programs. Caution is required before making claims as to whether these results would be observed on other programs, possibly from different sources and in different programming languages. As with all such experimental software engineering, further experiments are required in order to replicate the results contained here. The results do show, however, that there are cases where procedures can be split into two or more components with varying degrees of code overlap and that there is a correlation between procedure size and procedure splitability for the programs studied.

5.5 Summary

The research provides an approach to split procedures allowing the exploration of procedure structure. The approach provides executable sub-procedures as candidates, thus potentially contributing to program understanding, maintenance and procedure reuse.

An empirical study is performed using six real world open source C programs. The empirical investigation targeted, firstly, evaluating program structures in terms of splitable procedure distribution, secondly, defining procedure splitability for 2-way and 3-way splitable procedures, and analysing the splitability distribution for these two classes of splitable procedures, and thirdly exploring the correlation between procedure size and procedure splitability. The results show that there is a strong correlation between procedure size and procedure splitability in the case of 2-way splitable procedures. However, the results indicate no strong correlation in the case of 3-way splitable procedures for five out of six programs.
Chapter 6

Patterns of splitability of procedures

This chapter analyses the patterns of splitable procedures, that is, in what way a procedure can be split. There are two classifications of procedures, non splitable and splitable procedures. The chapter looks into the splitable procedures studied in the splitting algorithm and generalises from these 8 patterns for the 194 splitable procedures in terms of program schema-based and graphical representations. Also, the chapter provides the corresponding concrete procedures as illustrative examples.

The motivation for this work is derived from the validation of the splitting algorithm. Moreover, the thesis may explore the category of splitable procedures in terms of the greedy-based splitting approach by looking into the code and suggested split points. The goals of analysing splitable patterns are summarised as follows:

1. Verify the validity of the splitting algorithm by looking into the 194 splitable procedures and by generalising the categories of attributes that make procedures splitable.

2. Summarise 8 patterns among the 194 ‘typical’ splitable procedures. Each pattern represents a classification of procedures that can be split in a specific way.
In this chapter, the thesis also provides real procedures studied in the empirical investigation and each procedure corresponds to a schema-based pattern as an illustrative example. For some examples, the chapter only shows the main part of the procedures due to space limitation. However, all the procedures studied are open source code that can be found at the ‘GNU’ website (ftp://ftp.gnu.org/gnu).

### 6.1 The results for case studies of splitable procedure patterns

The thesis looked into all the splitable procedures studied with the greedy-based splitting algorithm in the previous chapter and generalised 8 splitable patterns. There are two tables that provide the distribution of all the patterns as follows.

<table>
<thead>
<tr>
<th>Patterns</th>
<th>1 (CIB)</th>
<th>2 (IBSB)</th>
<th>3 (ICB)</th>
<th>4 (PDCB)</th>
<th>5 (IC)</th>
<th>6 (PSC)</th>
<th>7 (ICSB)</th>
<th>8 (PDB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Number of Procedures</td>
<td>20</td>
<td>17</td>
<td>28</td>
<td>34</td>
<td>13</td>
<td>17</td>
<td>10</td>
<td>55</td>
</tr>
<tr>
<td>Percentage of Patterns</td>
<td>10.3</td>
<td>8.8</td>
<td>14.4</td>
<td>17.5</td>
<td>6.7</td>
<td>8.8</td>
<td>5.2</td>
<td>28.4</td>
</tr>
<tr>
<td>Total of splitable Procedures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>194</td>
</tr>
</tbody>
</table>

Table 6.1: Distribution of patterns with a set of splitable procedures. The definitions of the eight patterns are described in the following sections.

Table 6.1 shows the distribution of 8 patterns of splitable procedures. The majority of distributions lies in Pattern 8 (28.4%), 4 (17.5%) and 3 (14.4%). For these patterns, it is difficult to recognise the structure of procedures manually, especially for Patterns 8. This is partly because the subprocedures are not independent of each other totally (e.g. Pattern 8), or partly because the splitting points are hard to find out (e.g. Patterns 3 and 4). This situation similarly occurs in the rest of patterns. Thus, using the splitting algorithm can relatively ease the program structure analysis.
Table 6.2: Distribution of patterns with a set of splitable procedures. The entry ‘No. of sub’ represents the number of sub-procedures in the corresponding pattern; ‘Percentage’ represents the number that can be split into the number of corresponding sub-procedures to the total number of procedures in the pattern.

Table 6.2 provides more details of the distribution of splitable procedures for each pattern. As shown in the table, the majority lies in the procedures that are split into 2 sub-procedures. Some of the procedures can be split into 3 sub-procedures and less can be split into 4. The percentage will decrease while the number of sub-procedures increases.

### 6.2 Program Model and Terminology

In this chapter, the thesis gives graphical representations of the splitting patterns. In these graphs (referring to Figure 6.15 as an illustrative example), a node represents a collection of statements, a target variable, a condition variable or a condition block. An arrow from node $A$ to node $B$, $A \rightarrow B$ expresses computational dependency of $B$ on $A$. A dotted arrow $A \dashrightarrow B$ represents partial dependency. $B$ is partially dependent
on $A$ means that some statements within $A$ do contribute to the computation of variables in $B$. As well as the graphical representations, the chapter will also use schema-based representations to formalise the patterns found. Next, this section introduces the related definitions as follows:

**Definition 11 Linear Schema**

Program schemas define a class of programs, all of which have identical statement structure but whose expressions may differ. In linear schema, each function and predicate symbol occurs at most once.

A program schema depicts the structure of a program by replacing real expressions and predicates with the symbols of expression and predicate. A schema thus defines a whole class of programs, all of which have the same structure. One can obtain a real program from a program schema by substituting expression and predicate symbols with real meaningful expressions and predicates. As the example in Figure 6.1 shows, Functions $a$, $c$ and $d$ represent the symbols of expressions while Function $b$ represents the symbol of a predicate. Thus, a real program can be obtained by replacing the symbols of expressions and predicates with some concrete ones. As shown in Figure 6.2, Functions $a()$, $b(w)$, $c()$ and $d()$ can be replaced with expressions and predicate $10$, $w < 5$, $3$ and $5$.

```
u := a();
if b(w)
    then v := c();
else v := d();
```

Figure 6.1: An example of program schema.

This is the standard definition for linear program schema, which have a well developed theory. Danicic et al. [37] introduced linear schemas to represent classes of programs for dependence analysis. The ‘pattern analysis based’ schema notation
is derived from the standard notation of linear schema. The definition – Variant of Linear Schema, introduced below is a variant of linear schema.

**Definition 12 Variant of Linear Schema**

In program schemas, the symbol \( f(u) \) represents an expression or a predicate. Symbol \( v \leftarrow \{u\} \) represents a computation of the variable \( v \) which depends on the variable \( u \).

The concept of \( v \leftarrow \{u\} \) essentially corresponds to the backward slice of the variable \( v \) where \( v \) is dependent on \( u \).

**Definition 13 Condition Variable**

A condition variable is represented by a node with at least two successors. The successors are determined by the boolean value (T,F) of the condition predicate.

**Definition 14 Target Variable**

A variable computed and delivered (returned) by a procedure is referred to as a **target variable**.

**Definition 15 Collection**

A *collection* of statements that are logically related and not directly controlled by any condition variable is represented by a single node of a special kind. The relationship between the statements is determined by contributing to the computation of a target variable.

**Definition 16 Condition Block**

Another special kind of node, referred to as a condition block, represents all statements whose execution is controlled by a condition variable.
Definition 17 *Program Schema Expression*
Given a set of variables $\mathcal{V} = \{v_1, ..., v_n\}$, a program schema symbol $f(v_i, v_j)$ ($0 \leq i \neq j \leq n$) expresses a program expression that depends on Variable $v_i$, does not depend on Variable $v_j$ in Set $\mathcal{V}$, and possibly depends on the rest of $\mathcal{V}$. Furthermore, $f(All)$ specifically represents an expression that may possibly depend on any of all the variables in a procedure.

Definition 18 *Program Schema Computation*
Given a set of variables $\mathcal{V} = \{v_1, ..., v_n\}$ and a variable $w$, Symbol $w \leftarrow \{v_i, v_j\}$ expresses a computation of the defined variable $w$ where the computation depends on the variable $v_i$ and does not depend on the variable $v_j$. Similarly, $\{All\}$ specifically represents the computation that possibly depends on all the variables in a procedure.

In what follows, the thesis considers procedures computing single or a set of target variables. The computations may depend on condition variables. Both, target variables and condition variables can be global, local or formal parameters of arbitrary types.

6.3 Splitting Patterns

6.3.1 Pattern 1 : Completely Independent Blocks (CIB)
Given a procedure $\tau$ computing a target variable $v$, the pattern considered here deals with the case where the computation of $v$ depends on different assignments of a condition variable $u$. In this case, the computations of $v$, together with the corresponding conditions represent independent sub-procedures which can be split.

In the CIB pattern procedure code consists of different condition blocks, all of which depend on different instantiations of one variable. Depending on the value of this variable, a specific computation of the target variable is performed.
Symbolically, we can express the situation in the following way. $\tau(v, u, rest)$ expresses $\tau$-dependency on the set of variables $v$, $u$ and $rest$. $rest$ represents a set of auxiliary variables relevant to the computation of $u$ and $v$. For one value of $u$, a condition is satisfied and, as a result, a certain part of the program is executed delivering a value for $v$. Consequently, for each value of $u$, there corresponds a set of program statements computing $v$. The set of program statements computing $v$, which depends on the value of $u$ represents a splitable sub-procedure of $\tau$.

Assume $u$ takes the values $U_0, U_1, ..., U_n$, the corresponding variables satisfied during program execution are $CS_1, CS_2, ..., CS_n$, and the corresponding $v$ computations deliver $V_0, V_1, ..., V_n$. $\tau(v, u, rest)$ can be split into $\tau_0(V_0, U_0, CS_0)$, $\tau_1(V_1, U_1, CS_1)$, ..., $\tau_n(V_n, U_n, CS_n)$.

Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.3. In the figure, lines 1-5, 6-10 and 11-15 represent independent parts, respectively, such that this class of programs can be split into a set of independent sub-procedures. Figure 6.4 shows the corresponding schema-based subprocedures.

Figure 6.5 provides a graphical representation of the pattern for the case where a condition variable takes different values $U_0, U_1, ..., U_n$. Different sub-procedures are identified, each of which corresponds to a different value of the condition variable and delivers a different value of the target variable, $V_0, V_1, ..., V_n$. In this figure, the ‘Cases’ node represents the set of procedure conditions and ‘Finalise’ the set of statements required to finalise the computation of the target variable. The computations may include other intermediate variables on which the values $V_0, V_1, ..., V_n$ depend.

Figure 6.6 shows the source code of the procedure `adddef`, the splitable procedure corresponding to Pattern 1 (CIB). In the figure, there are 9 if condition blocks controlled by the condition variable `error` with the different values. The block, lines 2-4, is controlled by the value assigned in the line 1; similarly, the blocks, lines 6-9
// Definition of condition variable u
1 \text{if} \ g_0(u) \ \text{then}
2 \quad v \leftarrow \{\text{All}\};
3 \quad \text{return;}
4 \end
5

......

// Definition of condition variable u
6 \text{if} \ g_i(u) \ \text{then}
7 \quad v \leftarrow \{\text{All}\};
8 \quad \text{return;}
9 \end
10

......

// Definition of condition variable u
11 \text{if} \ g_n(u) \ \text{then}
12 \quad v \leftarrow \{\text{All}\};
13 \quad \text{return;}
14 \end
15

Figure 6.3: Program Schema for Pattern 1 (CIB)
// Sub-procedure 1
1 $u = f_0(All)$;
2 if $g_0(u)$ then
3  $v \leftarrow \{All\}$;
4  return;
5 end

// Sub-procedure i
6 $u = f_i(All)$;
7 if $g_i(u)$ then
8  $v \leftarrow \{All\}$;
9  return;
10 end

// Sub-procedure 3
11 $u = f_n(All)$;
12 if $g_n(u)$ then
13  $v \leftarrow \{All\}$;
14  return;
15 end

Figure 6.4: Sub-procedures with Program Schema for Pattern 1 (CIB)

Figure 6.5: Pattern 1 (CIB) of splitable procedures
int adddef(struct charac * p1, struct charac ** pp2, struct AddRem ** addrem_ptr)
{
    Variable Declarations and Initializations
    1. error = (GetKeyword(Keywords[79], curr_ptr));
    2. if (error != 0){
    3.     *addrem_ptr = NULL;
    4.     return 1;}
    5. erro = nodedef(*curr_ptr, curr_ptr, node_ptr);
    6. if (error == 17) {
    7.     *addrem_ptr = NULL;
    8.     parserro(*curr_ptr, 58, " ");
    9.     return 17;}
10. if (error == 0) {
11.     *addrem_ptr = malloc(sizeof(struct AddRem));
12.    if (*addrem_ptr == NULL){
13.        parserro(*curr_ptr, 55, " ");
14.        inerror("adddef()");}
15.    (*addrem_ptr)->ADDREM_TYPE = ADD_TYPE;
16.    (*addrem_ptr)->BLOCK_TYPE = NODE_BLOCK;
17.    (*addrem_ptr)->NODE_PTR = *node_ptr;
18.    (*addrem_ptr)->NEXT = NULL;
19.    *pp2 = *curr_ptr;
20.    return 0;}
Another four blocks controlled by condition variable error
.
.
.
21. error = hexdef(*curr_ptr, curr_ptr, node_ptr);
22. if (error == 17) {
23.     *addrem_ptr = NULL;
24.     parserro(*curr_ptr, 59, " ");
25.     return 17;}
26. if (error == 0) {
27.     *addrem_ptr = malloc(sizeof(struct AddRem));
28.    if (*addrem_ptr == NULL){
29.        parserro(*curr_ptr, 55, " ");
30.        inerror("adddef()");}
31.    (*addrem_ptr)->ADDREM_TYPE = ADD_TYPE;
32.    (*addrem_ptr)->BLOCK_TYPE = BLOCK_BLOCK;
33.    (*addrem_ptr)->NODE_PTR = *node_ptr;
34.    (*addrem_ptr)->NEXT = NULL;
35.    *pp2 = *curr_ptr;
36.    return 0;}

Figure 6.6: The procedure corresponding to Pattern 1 (CIB). The chosen example is the procedure adddef in space.c of the project space from the ‘Gnu’ website.
Patterns of splitability of procedures

and 10-20 are controlled by the value assigned in the line 5; the blocks, lines 22-25 and 26-36 by the value assigned in line 21. Totally, there are 9 blocks each of which represents a sub-procedure. The corresponding schema-based pattern is shown in Figures 6.3 and 6.4.

6.3.2 Pattern 2 : Independent Blocks with Shared Block (IBSB)

The pattern described in this section is an extension of Pattern 1 (CIB). Here, a procedure computes several variables and deals with several condition variables.

Given a procedure $\tau$ computing a set of target variables $\{v_0, v_1, \ldots\}$, the pattern considered deals with the case where these computations are independent of each other except that the computations of the target variables depend on a common condition block. In this case, the computations of the target variables represent independent sub-procedures that can be split. All sub-procedures share the common condition block on which all computations are dependent.

Symbolically, $\tau(v_0, v_1, \ldots, u_0, u_1, \ldots, rest)$ expresses $\tau$-dependency on a set of variables where each computation of the $i^{th}$ target variable $v_i$ depends on a condition variable $u_i$ and represents a splittable sub-procedure of $\tau$. In addition, all sub-procedures share a common condition block on which all the computations depend.

Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.7. In the figure, lines 1-5 are a shared part by lines 6-9, 10-13 and 14-17, such that each sub-procedure includes the share part. Figure 6.8 shows the corresponding schema-based sub-procedures.

Figure 6.9 provides a graphical representation of the pattern with $n + 1$ target variables and $n + 1$ condition variables. Here, ‘Block’ represents condition blocks on which the computation of all the target variables depend; ‘Cases’ represents conditions with different condition variables $\{u_0, u_1, \ldots, u_n\}$. The nodes labelled $v_0, v_1, \ldots, v_n$
// Computation of Block all the target variables depend on
1 if $d_0(All)$ then
2 \hspace{1em} a_0 \leftarrow \{All\};
3 \hspace{1em} a_1 \leftarrow \{All\};
4 \hspace{1em} a_2 \leftarrow \{All\};
5 end

// Definition of condition variable $u_0$
6 $u_0 = f_0(All)$;
7 if $g_0(u_0)$ then
8 \hspace{1em} v_0 \leftarrow \{a_0, a_1, a_2, v_1, \ldots, v_n\};
9 end

...  // Definition of condition variable $u_i$
10 $u_i = f_i(All)$;
11 if $g_i(u_i)$ then
12 \hspace{1em} v_i \leftarrow \{a_0, a_1, a_2, v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n\};
13 end

...  // Definition of condition variable $u_n$
14 $u_n = f_n(All)$;
15 if $g_n(u_n)$ then
16 \hspace{1em} v_2 \leftarrow \{a_0, a_1, a_2, v_0, \ldots, v_{n-1}\};
17 end

Figure 6.7: Program Schema for Pattern 2 (IBSB)
// Sub-procedure 0
1 if $d_0(All)$ then
2 $a_0 \leftarrow \{All\}$;
3 $a_1 \leftarrow \{All\}$;
4 $a_2 \leftarrow \{All\}$;
5 end
6 $u_0 = f_0(All)$;
7 if $g_0(u_0)$ then
8 $v_0 \leftarrow \{a_0, a_1, a_2, v_1, \ldots, v_n\}$;
9 end

// Sub-procedure i
10 if $d_0(All)$ then
11 $a_0 \leftarrow \{All\}$;
12 $a_1 \leftarrow \{All\}$;
13 $a_2 \leftarrow \{All\}$;
14 end
15 $u_i = f_i(All)$;
16 if $g_i(u_i)$ then
17 $v_i \leftarrow \{a_0, a_1, a_2, v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n\}$;
18 end

// Sub-procedure n
19 if $d_0(All)$ then
20 $a_0 \leftarrow \{All\}$;
21 $a_1 \leftarrow \{All\}$;
22 $a_2 \leftarrow \{All\}$;
23 end
24 $u_n = f_n(All)$;
25 if $g_n(u_n)$ then
26 $v_n \leftarrow \{a_0, a_1, a_2, v_0, \ldots, v_{n-1}\}$;
27 end

Figure 6.8: Sub-procedures with Program Schema for Pattern 2 (IBSB)
represent computations of the target variables. As before, the computations may include other intermediate variables.

Figure 6.10 shows the source code of the procedure create_file_names, the splitable procedure corresponding to the Pattern 2 (IBSB). In the figure, the condition blocks, lines 1-5 are the shared blocks by independent blocks, lines, 6-7, 8-9, 10-11, 15-21, 22-27 and 28-33. Each independent block is controlled by the different condition variables. The corresponding schema-based pattern is shown in Figures 6.7 and 6.8.

6.3.3 Pattern 3: Independent Collections & Blocks (ICB)

Given a procedure $\tau$ computing a set of target variables $v_0, v_1, ..., v_n$, the pattern considered here deals with the case where some of the target variables are computed within different condition blocks and others are computed with a set of statements independent of the condition blocks. That is, the set of target variables consists of two parts, $v_B$ and $v_S$ where the first represents those variables computed within the condition blocks and the latter those computed outside any condition blocks. In this
create_file_names() {
    Variable Declarations and Initializations
    1 tmpdir = getenv("TMPDIR");
    2 if (tmpdir == 0) tmpdir = "/tmp";
    3 len = strlen(tmpdir);
    4 i = len + 13;
    5 if (len && tmpdir[len-1] != '/') ++i;
    6 action_file_name = MALLOC(i);
    7 if (action_file_name == 0) no_space();
    8 text_file_name = MALLOC(i);
    9 if (text_file_name == 0) no_space();
    10 union_file_name = MALLOC(i);
    11 if (union_file_name == 0) no_space();
    Another two blocks dependent on the blocks, lines 1-5
   .
   .
   .
    12 strcpy(action_file_name+len, tmpdir);
    13 strcpy(text_file_name+len, tmpdir);
    14 strcpy(union_file_name+len, tmpdir);
    15 if (rflag) {
        16 code_file_name = MALLOC(len + 8);
        17 if (code_file_name == 0){
            18 no_space();
            19 strcpy(code_file_name, file_prefix);
            20 strcpy(code_file_name + len, CODE_SUFFIX);
        } else code_file_name = output_file_name;
    21 if (dflag){
        22 defines_file_name = MALLOC(len + 7);
        23 if (defines_file_name == 0)
            24 no_space();
            25 strcpy(defines_file_name, file_prefix);
            26 strcpy(defines_file_name + len, DEFINES_SUFFIX);
        } else defines_file_name = output_file_name;
    27 if (vflag){
        28 verbose_file_name = MALLOC(len + 8);
        29 if (verbose_file_name == 0)
            30 no_space();
            31 strcpy(verbose_file_name, file_prefix);
            32 strcpy(verbose_file_name + len, VERBOSE_SUFFIX);
        } else verbose_file_name = output_file_name;
    }
}

Figure 6.10: The procedure corresponding to Pattern 2 (IBSB). The chosen example is the procedure create_file_names in main.c of the project byacc from the ‘Gnu’ website.
case, each block or each collection of statements contributing to the computation of a target variable represents an independent sub-procedure.

Procedure code consists of a set of independent condition blocks computing a subset of the target variables and a collection of statements computing the rest outside these blocks. The computations could include other intermediate variables on which target variables depend.

Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.11. In the figure, from lines 1-4 to 5-8 represent the independent ‘Condition Block’ parts and line 9 represents the independent ‘Collection’ part such that each part represents an independent sub-procedure. Figure 6.12 shows the corresponding schema-based sub-procedures.

```plaintext
// Definition of condition variable u0
1 u0 = f0(All);
2 if g0(u0) then
3   v_{B0} ← \{v_{B1}, \cdots, v_{Bn}, v_{S0}\};
4 end
...

// Definition of condition variable u_n
5 u_n = f_n(All);
6 if g_n(u_n) then
7   v_{Bn} ← \{v_{B0}, \cdots, v_{B(n-1)}, v_{S0}\};
8 end

// Computation of the target variable outside any blocks
9 v_{S0} ← \{v_{B0}, \cdots, v_{Bn}\};
```

Figure 6.11: Program Schema for Pattern 3 (ICB)

Figure 6.13 shows the graphical representation of the pattern where Block0,..., Blockn represent independent condition blocks computing target variables, ‘Collection’ represents the computation of another target variable v. In this example, n + 2 sub-procedures arise as a result of splitting the procedure code.

Figure 6.14 shows the source code of the procedure `prepare_termcap`, the splitable procedure corresponding to Pattern 3 (ICB). In the figure, there are four indepen-
// Sub-procedure 0
1 \( u_0 = f_0(All) \);
2 if \( g_0(u_0) \) then
3 \( v_{B_0} \leftarrow \{ v_{B_1}, \ldots, v_{B_n}, v_{S_0} \} \);
4 end

// Sub-procedure n
5 \( u_n = f_n(All) \);
6 if \( g_n(u_n) \) then
7 \( v_{B_n} \leftarrow \{ v_{B_1}, \ldots, v_{B(n-1)}, v_{S_0} \} \);
8 end

// Sub-procedure n+1
9 \( v_{S_0} \leftarrow \{ v_{B_0}, \ldots, v_{B_n} \} \);

Figure 6.12: Sub-procedures with Program Schema for Pattern 3 (ICB)

Figure 6.13: Pattern 3 (ICB) of splitable procedures
static void prepare_termcap (term) {
    Variable Declerations and Initializations
    1 if (term == NULL)
    2    error (1, 0, “No value for TERM and no T specified”);
    3 switch (tgetent (0, term)){
    4    case 0:
    5        error (1, 0, “Unknown terminal type %s”, term);
    6    case (-1):
    7        error (1, 0, “No termcap database”);
    8        tc_pc=tgetstr(“pc”, 0);
    9        PC = tc_pc ? *tc_pc : 0;
    10    term_width = tgetnum (“co”);
    11    if (term_width <= 0)
    12    term_width = 80;}

Figure 6.14: The procedure corresponding to Pattern 3 (ICB). The chosen example is the procedure prepare_termcap in tabs.c of the project termutils from the ‘Gnu’ website.

ident parts: the condition blocks, lines 1-2, lines 3-7 and lines 10-12 and the collection, lines 8-9. Each independent block is controlled by the different condition variables and the collection part computes the variable PC. The corresponding schema-based pattern is shown in Figures 6.11 and 6.12.

6.3.4 Pattern 4 : Partially Dependent Collection & Block (PDCB)

This pattern is a generalisation of Pattern 3 (ICB) where a collection/block may partially contribute to the computation of each other.

Figure 6.15 shows the graphical form of the pattern. The partial dependency between the variable v and Block is represented by a dotted arrow. Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.16. In the figure, line 1 is an independent part, and lines 7-8 and ‘Block’, lines 2-6 are possibly partially dependent on each other. Figure 6.17 shows the corresponding schema-based sub-procedures.
Figure 6.15: **Pattern 4 (PDCB)** of splitable procedures

```
// Computation of variables v_{Si}
1 v_{Si} ← \{v_{Ba}, v_{Bb}, v_{Sa}, v_{Sb}\};
  ···
// Definition of condition variable u
2 u = f(All);
3 if g(u) then
  4 v_{Ba} ← \{v_{Sa}, v_{Sb}, v_{Si}\};
  5 v_{Bb} ← \{v_{Sa}, v_{Sb}, v_{Si}\};
6 end
  ···
// Computation of the target variable outside any block
7 v_{Sa} ← \{v_{Ba}, v_{Si}\};
8 v_{Sb} ← \{v_{Bb}, v_{Sa}, v_{Si}\};
```

Figure 6.16: Program Schema for Pattern 4 (PDCB)

```
// Sub-procedure i
1 v_{Si} ← \{v_{Ba}, v_{Bb}, v_{Sa}, v_{Sb}\};
// Sub-procedure i+1
2 u = f(All);
3 if g(u) then
  4 v_{Ba} ← \{v_{Sa}, v_{Sb}, v_{Si}\};
  5 v_{Bb} ← \{v_{Sa}, v_{Sb}, v_{Si}\};
6 end
  7 v_{Sa} ← \{v_{Ba}, v_{Si}\};
// Sub-procedure i+2
  8 u = f(All);
  9 if g(u) then
    10 v_{Bb} ← \{v_{Sa}, v_{Sb}, v_{Si}\};
  11 end
    12 v_{Sa} ← \{v_{Ba}, v_{Si}\};
    13 v_{Sb} ← \{v_{Bb}, v_{Sa}, v_{Si}\};
```

Figure 6.17: Sub-procedures with Program Schema for Pattern 4 (PDCB)
static void yy_include_push (char *file) {
    Variable CRUD: and Initializations
    1 if (include_stack.ptr >= MAX_INCLUDE_DEPTH)
    2 + error (1, 0, (“too many includes”));
    3 include_stack[include_stack.ptr] = YY_CURRENT_BUFFER;
    4 filename_stack[include_stack.ptr++] = file;
    5 message (msg_file, (stderr, “%s:%d: includes %s”, yyfilename, yylineno, file));
    6 yyin = xrfopen (yyfilename);
    7 yy_switch_to_buffer (yy_create_buffer (yyin, YY_BUF_SIZE));
}  

Figure 6.18: The procedure corresponding to Pattern 4 (PDCB). The chosen example is the procedure yy_include_push in sheets-map.c of the project a2ps from the ‘Gnu’ website.

Figure 6.18 shows the source code of the procedure yy_include_push, the splitable procedure corresponding to the Pattern 4 (PDCB). In the figure, there are two sub-procedures – the block, lines 1-2 and the condition, lines 3-7. The computation of the block part shares the statements 3, 4, 6 and 7 in the collection part. These kinds of dependences are caused by function call such that the statement 2 depends on the statements after its position. This situation frequently occurs in inter-procedural slices due to global variables and function calls. Thus, tracing this dependency relationship is very different and error-prone since programmers must trace back each predecessor from the variable of interest step by step until covering all the related dependency. However, the splitting algorithm may automate this process to suggest the possible sub-procedure in terms of finding splitting points in the procedure. The corresponding schema-based pattern is shown in Figures 6.16 and 6.17.

6.3.5 Pattern 5: Independent Collections (IC)

Given a procedure \( \tau \) consisting of a set of ‘Collection’s computing a set of target variables, the pattern considered here deals with the case where all the variables are not computed in any ‘condition blocks’. In this case, the computations of variables
can be split into a set of independent ‘Collection’ s. Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.19. In the figure, lines 1, 2 and 3 are independent ‘Collection’ parts, respectively, such that this class of programs can be split into a set of independent sub-procedures. Figure 6.20 shows the corresponding schema-based sub-procedures and Figure 6.21 shows the corresponding graphic form of the pattern.

```
// Computation of the target variable v_0 outside any block
1 \[ v_{S0} \leftarrow \{v_{S1}, \cdots, v_{Sn}\}; \]

// Computation of the target variable v_i outside any block
2 \[ v_{Si} \leftarrow \{v_{S0}, \cdots, v_{Si-1}, v_{Si+1}, \cdots, v_{Sn}\}; \]

// Computation of the target variable v_n outside any block
3 \[ v_{Sn} \leftarrow \{v_{S0}, \cdots, v_{Sn-1}\}; \]
```

Figure 6.19: Program Schema for Pattern 5 (IC)

```
// Sub-procedure 0
1 \[ v_{S0} \leftarrow \{v_{S1}, \cdots, v_{Sn}\}; \]

// Sub-procedure i
2 \[ v_{Si} \leftarrow \{v_{S0}, \cdots, v_{Si-1}, v_{Si+1}, \cdots, v_{Sn}\}; \]

// Sub-procedure n
3 \[ v_{Sn} \leftarrow \{v_{S0}, \cdots, v_{Sn-1}\}; \]
```

Figure 6.20: Sub-procedures with Program Schema for Pattern 5 (IC)

Figure 6.22 shows the source code of the procedure version, the splitable procedure corresponding to Pattern 5 (IC). This example is the simplest independent collection situation where there are only two statements. Interestingly, each statement represents an independent functionality. In this case, most of situation reflects the specific purpose by developer. For example, one may just simple put a set of ‘printing task’ calls together, which completes a batch of printing tasks. The corresponding
Search-Based Slicing

Patterns of splitability of procedures

![Diagram](a) Original Procedure

![Diagram](b) A Set of Sub-Procedures

Figure 6.21: Pattern 5 (IC) of splitable procedures

```c
void version () {
    printf ("GNU tabs version %s", version_string);
    exit (0);
}
```

Figure 6.22: The procedure corresponding to Pattern 5 (IC). The chosen example is the procedure `version` in `tabs.c` of the project `termutils` from the ‘Gnu’ website.

schema-based pattern is shown in Figure 6.19 and 6.20.

6.3.6 Pattern 6: Partially Shared Collection (PSC)

Given a procedure $\tau$ computing a set of variables. A subset of variables $V_B$ is computed in condition blocks and a subset of variables $V_S$ is computed with collections of statements outside any blocks. The pattern deals with the case where the computations of $V_B$ all may partially depend on the computations of $V_S$ in `Collection`. In this case, the computations of $V_B$ can be split into a set of corresponding sub-procedures each of which includes part of `Collections`.

Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.23. In the figure, lines 1-3 are partially shared by independent block parts, lines 4-7, 8-11 and 12-15. Figure 6.24 shows the corresponding schema-based subprocedures.

This kind of procedure mainly consists of a set of blocks and includes a computa-
tion of variables independent of one another. As shown in Figure 6.25, all the blocks partially depend on the computation of the variables represented with ‘Collection’. All the blocks are independent of each other.

// Computation of variables $v_S$ the target variables $v_B$ depend on
1 $v_{S0} \leftarrow \{\text{All}\}$;
2 $v_{Si} \leftarrow \{\text{All}\}$;
3 $v_{Sn} \leftarrow \{\text{All}\}$;

// Definition of condition variable $u_0$
4 $u_0 = f_0(\text{All})$;
5 if $g_0(u_0)$ then
6 $v_{B0} \leftarrow \{v_{S0}, \overline{v_B1}, \ldots, \overline{v_Bn}\}$;
7 end

// Definition of condition variable $u_i$
8 $u_i = f_i(\text{All})$;
9 if $g_i(u_i)$ then
10 $v_{Bi} \leftarrow \{v_{Si}, \overline{v_B0}, \ldots, \overline{v_{Bi-1}}, \overline{v_{Bi+1}}, \ldots, \overline{v_Bn}\}$;
11 end

// Definition of condition variable $u_n$
12 $u_n = f_n(\text{All})$;
13 if $g_n(u_n)$ then
14 $v_{Bn} \leftarrow \{v_{Sn}, \overline{v_B0}, \ldots, \overline{v_{Bn-1}}\}$;
15 end

Figure 6.23: Program Schema for Pattern 6 (PSC)

Figure 6.26 shows the source code of the procedure buffer.save, the splitable procedure corresponding to Pattern 6 (PSC). This example consists of three sub-procedures, two blocks, lines 2-5 and 6-7, and one collection line 1 and 8. The computations of the first block depends on the statement 1 and the computation of the second block depends on the statements 8 and 1 in the the computation of the collection. The corresponding schema-based pattern is shown in Figure 6.23 and 6.24.
// Sub-procedure 0
1 \(v_{S0} \leftarrow \{\text{All}\};\)
2 \(u_0 = f_1(\text{All});\)
3 if \(g_0(u_0)\) then
4 \(v_{B0} \leftarrow \{v_{S0}, v_{B1}, \ldots, v_{Bn}\};\)
5 end

// Sub-procedure i
6 \(v_{Si} \leftarrow \{\text{All}\};\)
7 \(u_i = f_i(\text{All});\)
8 if \(g_i(u_i)\) then
9 \(v_{Bi} \leftarrow \{v_{Si}, v_{Bi+1}, \ldots, v_{Bn}\};\)
10 end

// Sub-procedure n
11 \(v_{Sn} \leftarrow \{\text{All}\};\)
12 \(u_2 = f_n(\text{All});\)
13 if \(g_n(u_n)\) then
14 \(v_{Bn} \leftarrow \{v_{Sn}, v_{B0}, \ldots, v_{Bn-1}\};\)
15 end

Figure 6.24: Sub-procedures with Program Schema for Pattern 6 (PSC)

(a) Original Procedure

(b) A Set of Sub-Procedures

Figure 6.25: Pattern 6 (PSC) of splitable procedures
void buffer_save (buffer_t * buffer, const char *filename){
    Variable Declarations and Initializations
1    FILE *out = xwfopen (filename);
2    if (buffer- >buf){
3        size_t cur;
4        for (cur = 0;cur < buffer- >bufsize;cur++)
5            putc (buffer- >buf[cur], out);}
6    if (buffer- >stream)
7        streams_copy (buffer- >stream,out);
8    fclose (out);
}

Figure 6.26: The procedure corresponding to Pattern 6 (PSC). The chosen example is the procedure buffer_save in buffer.c of the project a2ps from the ‘Gnu’ website.

6.3.7 Pattern 7 : Independent Collections with Shared Block (ICSB)

Given a procedure $\tau$ computing a set of variables, a subset of variables $V_B$ is computed in condition blocks and a subset of variables $V_S$ is computed with collections of statements outside any block. The pattern deals with the case where the computations of $V_S$ all may possibly partially depend on ‘Blocks’. In this case, the computations of $V_S$ can be split into a set of corresponding sub-procedures each of which includes part of the computation of $V_B$.

Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.27. In the figure, lines 1-5 are a partially shared part by lines 6, 7 and 8. Figure 6.28 shows the corresponding schema-based sub-procedures.

This kind of procedure mainly consists of a set of collections, and includes a computation of variables dependent on blocks. As shown in Figure 6.29, all the collections possibly partially depend on computation of variables represented within Blocks. All the collections are independent of each other.

Figure 6.30 shows the source code of the procedure output_trailing_text, the splitable procedure corresponding to Pattern 7 (ICSB). This example consists of
// Computation of Block all the target variables depend on
1 if $d_0(All)$ then
2 \hspace{1em} a_0 \leftarrow \{All\};
3 \hspace{1em} \vdots
4 \hspace{1em} a_i \leftarrow \{All\};
5 \hspace{1em} \vdots
6 \hspace{1em} a_n \leftarrow \{All\};
7 end

//
8 $v_{S0} \leftarrow \{a_0, v_{S1}, \ldots, v_{Sn}\}$;
9 \vdots

//
10 $v_{Si} \leftarrow \{a_i, v_{S0}, \ldots, v_{Si-1}, v_{Si+1}, \ldots, v_{Sn}\}$;
11 \vdots

//
12 $v_{Sn} \leftarrow \{a_n, v_{S0}, \ldots, v_{Sn-1}\}$;

Figure 6.27: Program Schema for Pattern 7 (ICSB)

// Sub-procedure 0
1 if $d_0(All)$ then
2 \hspace{1em} a_0 \leftarrow \{All\};
3 end
4 \hspace{1em} v_{S0} \leftarrow \{a_0, v_{S1}, \ldots, v_{Sn}\};
5 \hspace{1em} \vdots

// Sub-procedure i
6 if $d_0(All)$ then
7 \hspace{1em} a_i \leftarrow \{All\};
8 \hspace{1em} u_i = f_i(All);
9 \hspace{1em} v_{Si} \leftarrow \{a_i, v_{S0}, \ldots, v_{Si-1}, v_{Si+1}, \ldots, v_{Sn}\};
10 \hspace{1em} \vdots

// Sub-procedure n
11 if $d_0(All)$ then
12 \hspace{1em} a_n \leftarrow \{All\};
13 end
14 \hspace{1em} v_{Sn} \leftarrow \{a_n, v_{S0}, \ldots, v_{Sn-1}\};

Figure 6.28: Sub-procedures with Program Schema for Pattern 7 (ICSB)
void tputs (str, nlines, outfun) register char *str; int nlines;
register int (*outfun) {
  Variable Declearations and Initializations
  1 if (!str)
  2 return;
  3 while (*str >= 0 && *str <= 9) {
  4     padcount += *str++ - '0';
  5     padcount *= 10;
  6     if (*str == '.') {
  7         str++;
  8         padcount += *str++ - '0';
  9         if (*str == '*') {
 10             str++;
 11             padcount *= nlines;
 12         }
 13     }
 14     (*outfun) (*str++);
 15     padcount *= speed;
 16     padcount /= 1000;
 17     if (speed < 0)
 18         padcount -= padcount;
 19 else {
 20         padcount += 50;
 21     padcount /= 100;}
 22     while (padcount -- > 0)
 23     (*outfun) (PC);

Figure 6.30: The procedure corresponding to the Pattern 7 (ICSB). The chosen example is the procedure tputs in termcap.c of the project termutils from the ‘Gnu’ website.
three sub-procedures, the computations of lines 3-5, 12-13 and 22-23. The blocks, lines 1-2, 6-8, 9-11 and 17-21 are dependent parts which contribute the computation of the three `while` loops. For example, the first `while` depends on the first block; the second `while` loop depends on the first, the second and third blocks; the last `while` loop depends on the first and the last block. The corresponding schema-based pattern is shown in Figures 6.27 and 6.28.

### 6.3.8 Pattern 8 : Partially Dependent Blocks (PDB)

Given a procedure \( \tau \) consisting of a set of blocks. The blocks compute a set of the variables \( V_B \). The pattern considered here deals with the case where the computations of \( V_B \) may partially depend on each other. In this case, each condition block represents an independent sub-procedure. At the same time, all the sub-procedures may possibly copy part of statements from one another.

Employing program schema to describe this pattern, a class of programs can be identified as shown in Figure 6.31. In the figure, lines 1-4, 5-8 and 9-12 represent partially dependent parts each of which may possibly depend on one another, but it is not necessary that each block must depend on all the others. Figure 6.32 shows the corresponding schema-based sub-procedures.

This kind of procedure computes the target variables of \( V_B \) inside the corresponding block. The variables, besides depending on the specific block, partially depends on the statements from another blocks. As shown in Figure 6.33, Block 0, \( i \) and \( n \) represent the computations of \( V_B \) inside the blocks; the dash line represents the partial data/control dependency relationship between the blocks. Each computation of \( V_B \) may possibly include other statements outside its own block.

Symbolically, we can express the situation in the following way. \( \tau(V_B, V_{BD}, \text{rest}) \) expresses \( \tau \)–dependency on the set of Variables \( V_B \). Variables \( V_{BD} \) represent variables on which other variables are dependent. For each target variable in \( V_B \), the computa-
// Computation of Block some of the target variables depend on
1 if \( g_0(\text{All}) \) then
2 \( v_{B_0} \leftarrow \{v_{B_0}\} \);
3 \( v_{B_i} \leftarrow \{v_{B_i}\} \);
4 end
;

// Computation of Block some of the target variables depend on
5 if \( g_i(\text{All}) \) then
6 \( v_{B_i} \leftarrow \{v_{B_i}\} \);
7 \( v_{B_n} \leftarrow \{v_{B_0}\} \);
8 end
;

// Computation of Block some of the target variables depend on
9 if \( g_n(\text{All}) \) then
10 \( v_{B_n} \leftarrow \{v_{B_0}\} \);
11 \( v_{B_0} \leftarrow \{v_{B_0}\} \);
12 end

Figure 6.31: Program Schema for Pattern 8 (PDB)

... tion is performed partially in one block and partly preformed in another block. Consequently, Procedure \( \tau \) can be split into a set of sub-procedure \( \tau(V_{B_1}, V_{BD_1}, \text{rest}),..., \tau(V_{B_n}, V_{BD_n}, \text{rest}) \) representing the computations of \( V_B \).

Figure 6.34 shows the source code of the procedure \texttt{output_trailing_text}, the splitable procedure corresponding to Pattern 8 (PDB). This example consists of three sub-procedures – three blocks, lines 5-22, 23-29 and 30-31. The block, lines 1 is dependent part which is part of the computation of the block, lines 5-22. With dependence analysis, we can see that three sub-procedure blocks are intermingled – the computation of the block, lines 23-29 depends on the statements 5, 7, 8 and 9 in the block, lines 5-22 (including the whole block, line 1); the computation of the block, lines 30-31 depends on the statement 5, 7, 8, 9, 16, 17, 20 and 22 in the block, lines 5-22 and 24, 25, 28 in the block, lines 23-29 (including the whole block, line 1 as well). The corresponding schema-based pattern is shown in Figures 6.31 and 6.32.
// Sub-procedure 0
1 if $g_i(All)$ then
2 $v_{Bn} \leftarrow \{v_{B0}\}$;
3 end
4 if $g_0(All)$ then
5 $v_{B0} \leftarrow \{v_{B0}\}$;
6 $v_{Bi} \leftarrow \{v_{Bn}\}$;
7 end

// Sub-procedure i
8 if $g_0(All)$ then
9 $v_{Bi} \leftarrow \{v_{Bn}\}$;
10 end
11 if $g_i(All)$ then
12 $v_{Bi} \leftarrow \{v_{Bi}\}$;
13 $v_{Bn} \leftarrow \{v_{B0}\}$;
14 end

// Sub-procedure n
15 if $g_0(All)$ then
16 $v_{B0} \leftarrow \{v_{Bn}\}$;
17 end
18 if $g_n(All)$ then
19 $v_{Bn} \leftarrow \{v_{B0}\}$;
20 $v_{B0} \leftarrow \{v_{B1}\}$;
21 end

Figure 6.32: Sub-procedures with Program Schema for Pattern 8 (PDB)

Figure 6.33: Pattern 8 (PDB) of splitable procedures
output_trailing_text() {
    Variable Declarations and Initializations
    1    if (line == 0) return;
    2    in = input_file;
    3    out = code_file;
    4    c = *cptr;
    5    if (c == 'n') {
    6        ++lineno;
    7        if ((c = getc(in)) == EOF) return;
    8        if (!lflag){
    9            ++outline;
   10            fprintf(out, line_format, lineno, input_file_name);
   11            if (c == 'n') ++outline;
   12           putc(c, out);
   13            last = c;}
   14        else {
   15            if (!lflag){
   16                ++outline;
   17                fprintf(out, line_format, lineno, input_file_name);
   18                do { putc(c, out); } while ((c = *++cptr) != 'n');
   19                ++outline;
   20                putc('n', out);
   21                last = 'n';
   22            while ((c = getc(in)) != EOF){
   23                if (c == 'n') ++outline;
   24                putc('n', out);
   25                last = 'n';
   26            if (last != 'n'){
   27                ++outline;
   28                putc('n', out);
   29            if (!lflag)
   30                fprintf(out, line_format, ++outline + 1, code_file_name);}
    9        }
    8    }
    7}
    6}
    5}
    4}
    3}
    2}
    1}
}

Figure 6.34: The procedure corresponding to Pattern 8 (PDB). The chosen example is the procedure output_trailing_text in output.c of the project byacc from the ‘Gnu’ website.
6.4 Summary

This chapter looked into all the studied procedures and summarized 8 patterns for the splitable procedures. With the analysis of the patterns, the chapter validated the splitting algorithm that is consistent with the results from patterns analysis. On the other hand, the program schema used here classified the splitable procedures to represent different classes of procedures where in the same class procedures have the same structure and can be split in the particular way.
Chapter 7

Conclusions and Future Work

This chapter provides a summary of the achievements of the search-based slicing framework research outlined above. The motivation was to provide a fully automatic, flexible framework and to formulate program dependency structure problems as search-based problems.

The framework presented is a combination of program slicing and the SBSE framework. The Search-Based Slicing framework is a new methodology applied to source-code analysis. It is completely automated in contrast with previous work in this area. The innovative achievements of the work described in the thesis can be summarised in terms of different applications as follows:

7.1 Program Decomposition

The goals of this study were to introduce a novel framework, Search-Based Slicing, where the principles of SBSE are used to formulate any problem of locating dependency structures as a search problem.

Program decomposition presented the results of a case study that evaluated the application of greedy, hill climbing, random and genetic algorithms for both performance and similarity of results. Based upon the greedy algorithm, the best of the
four algorithms, a further empirical study was formed to explore how efficiently large programs and single functions can be decomposed.

The results indicated that the algorithms produced relatively consistent results and that the greedy algorithm outperformed its rivals. The results also provided evidence that the landscape for the problem is either of low modality or is multimodal, but with similarly-valued peaks. The results were encouraging, because they suggested that it is possible to formulate dependency analysis problems as search problems and to find good solutions in reasonable time spans using this approach.

The achievements can be summarized as follows:

1. An approach that identifies dependency structures was introduced as a search problem within the powerset of the set of all possible program slices, allowing search based algorithms to be used to search for dependency structures of interest.

2. A fitness function was introduced for guiding the search towards solutions that decomposed a program into a set of slices that collectively cover the whole program with minimal overlap. Four search algorithms were implemented in order to experiment with this fitness function.

3. The results of an empirical study were reported, showing that the greedy algorithm performed better than the hill climbing, random and genetic algorithm approaches to the problem. This is an attractive finding, since greedy algorithms are extremely simple and efficient.

4. A simple visualisation was introduced to explore the results and their similarity. This showed a higher degree of similarity in the results of the non-random search techniques than in those of the random search. This visual impression was augmented by computational analysis of results. The similarity of results for the non-random searches proved that the results are consistent and meaningful.
5. The visualisation also had an interesting side effect, which may be a useful spin-off: the presence of code clones became visually striking in some of the examples. However, clone detection was not the focus of this work.

6. The work also reported results on redundancy – that is, how often a slice is completely included in another one. The results suggested that the redundancy phenomenon is universal in twelve open source C programs. However, this redundancy does not affect the greedy algorithm advocated above.

7. Based upon the performance comparison of the four search algorithms, the greedy algorithm was further applied to six larger programs to decompose each procedure of each program. The results showed that the majority of procedures can be decomposed into sets of slices efficiently.

7.2 Splitable procedures and pattern analyses

This second study is another application of the SBS framework for dependency structure analysis. The study provided an approach to splitting procedures which allowed the exploration of procedure structures. The approach provided executable sub-procedures without changing the semantics, thus potentially contributing to program understanding, maintenance and procedure reuse.

An empirical study was performed using real world open source C programs. The empirical investigation targeted the evaluation of program structure in terms of splitable procedure distribution. The achievements are summarized below:

1. An approach was introduced to split a procedure into a set of sub-procedures.

   The approach provided a way to analyse and potentially to improve procedure structures. The approach also provided executable sub-procedures without changing semantics.
2. The empirical study showed that a surprisingly large proportion of the procedures considered are splitable.

3. The study provided evidence for a strong statistical correlation between procedure size and splitability.

At the same time, the thesis verified the validity of the splitting algorithm and classified the categories of splitable procedures with CFG-based representations and program schemas. The achievements are summarized below:

1. The validity of the splitting algorithm was verified by looking into the 194 splitable procedures and generalising the categories of attributes that make procedures splitable.

2. Eight patterns among the 194 ‘typical’ splitable procedures were summarised. Each pattern represents a classification of procedures that can be split in a specific way.

7.3 Future work

The flexibility of the SBS framework may open the way to new directions in source code analysis research. This section lists some potential future research directions.

7.3.1 Application to other programming languages

The thesis found the dependence between SDG nodes using a specific tool – Codesurfer. The tool applies to programs written in the C language. In order to identify generalized attributes, more languages with different features need to be explored. Certainly, this must depend on different tools supporting the work. For example, Indus [65] may slice Java programs.
7.3.2 Automation of the process of refactoring

The splitting algorithm explored the structure of procedures and provided executable sub-procedures. These sub-procedures were produced because of the ‘multi-task’ nature of procedures. As far as procedure refactoring is concerned, these sub-procedures might be considered to be candidates for refactoring elements.

However, the process of re-factoring is still not automated by the SBS framework. Thus one possible direction could be the automation of the refactoring of procedures, based on suggested split points from SBS.

7.3.3 Extension to multi-objective approaches

The purpose of SBS is to find an automatic approach to dependency structure analysis by simply defining fitness functions according to different problems. So far, the thesis has dealt with single-objective fitness functions which combined the sets of program metrics with coefficients.

The other alternative would be to have SBS keep each metrics separately in the search process. This ‘multi-objective’ approach or pareto frontier [96] will generate a pareto front line, rather than a fitness value. The SBS framework may take multiple objectives into account such that final optimal results can be chosen by programmers according to their particular aims.

In summary, the thesis presents a new framework with multiple applications and with plenty of scope for further development.
Bibliography


