Identifying High-Level Dependence Structures Using Slice-Based Dependence Analysis

by
Zheng Li

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Computing
King’s College London
University of London
Strand, London WC2R 2LS, UK

March 2009
© 2009 Zheng Li
Abstract

This thesis addresses dependence based concept assignment using slicing based dependence analysis on concept assignment.

Concepts are domain level entities concerned with the real world. Although concepts are defined at the domain level, the source code of a program must, nonetheless, express these concepts in program code denotations. Traditional concept assignment maps the low level of source code to the high level of the domain through the medium of associations. These associations typically take the form of comments and identifiers.

Previous work on the combination of dependence analysis with concept assignment gave the definition of either extending concepts to concept slices or abbreviating concepts to key statements. However, there are no empirical studies to show the feasibility of these two approaches.

In this thesis, we propose an improved key statement analysis algorithm and empirically study executable concept slicing and key statement analysis using ten subject programs with more than 600 concept bindings identified. We define dependence based metrics to evaluate both approaches and to provide evidence of both advantages and disadvantages. Based on these foundations, we propose a new combination technique – concept refinement, which adopts the advantages and overcome the disadvantages of previous two approaches.

The results for the same programs with the same concept bindings showed that concept refinement can produce dependence based concept bindings. The refined concept binding is smaller, but with similar impact and higher cohesion than the original HB-CA concept binding. Therefore, it becomes more precise after refinement; the refined concept binding is much smaller than the executable concept slice with respect to the original HB-CA concept binding. These findings suggest that the refined concept binding is more reasonable in size for program comprehension. Furthermore, the refined concept binding can be extracted from most of the original HB-CA concept bindings, thus concept refinement is more appropriate than the key statement analysis.
Acknowledgements

First of all, I would like to thank my supervisor Mark Harman for providing the support and motivation to do research without pressure. He has always been abundantly helpful, and has assisted me in numerous ways, including giving me the enthusiasm and inspiration. I would also like to thank my second supervisor Nicolas Gold for his contributions and suggestions throughout the duration of this work. I was funded by the EPSRC CONTRACTs and ASTReNet projects during the course of this research.

I would like to thank David Binkley for his very helpful communication and collaboration regarding dependence analysis.

Special thanks go to my colleague Kiarash Mahdavi for his valuable discussion during whole three years.

GrammaTech kindly provided the slicing tool CodeSurfer.

I cannot end without thanking my family. Without the love and support of them, this work would not have been possible. Many thanks to my loving wife and my patient parents, who have always supported me in my endeavour.
Declaration

The work presented in this thesis is original work undertaken between September 2004 and April 2008 at King’s College, University of London. Some of the work presented in this thesis has previously been published by the author in the following papers:


Note that author order is alphabetical. All of the work contained within this thesis represents the original contributions of the author.

Additionally, while undertaking this programme of research, the author also published the following papers and a research report to industry, which are related to the thesis, but from which no material is presented herein:


• Dave Binkley, Nicolas Gold, Mark Harman, Zheng Li Dependence Characteristics for DaimlerChrysler Programs Research Report for DaimlerChrysler, 2006
Notation

\( \mathbf{P} \) \hspace{1cm} \text{program} \\
\( s \) \hspace{1cm} \text{a statement in a program} \\
\( \mathbf{S} \) \hspace{1cm} \text{analysis scope (a set of contiguous statements)} \\
\( v \) \hspace{1cm} \text{a variable} \\
\( \mathbf{V} \) \hspace{1cm} \text{a set of variables} \\
\( \mathbf{PV} \) \hspace{1cm} \text{principal variable set} \\
\( \mathbf{PV}_O \) \hspace{1cm} \text{output principal variable set} \\
\( \mathbf{PV}_G \) \hspace{1cm} \text{global principal variable set} \\
\( \mathbf{PV}_{CBR} \) \hspace{1cm} \text{call-by-reference principal variable set} \\
\( \mathbf{KS} \) \hspace{1cm} \text{key statements set} \\
\( \mathbf{KS}_O \) \hspace{1cm} \text{key statements produced by } \mathbf{PV}_O \\
\( \mathbf{KS}_G \) \hspace{1cm} \text{key statements produced by } \mathbf{PV}_G \\
\( \mathbf{KS}_{Union} \) \hspace{1cm} \text{key statements produced by the union of } \mathbf{PV}_O \text{ and } \mathbf{PV}_G \\
\( \mathbf{S} \) \hspace{1cm} \text{static program slicing} \\
\( \overrightarrow{\mathbf{S}} \) \hspace{1cm} \text{forward slicing} \\
\( \overrightarrow{\mathbf{S}}(P, s, v) \) \hspace{1cm} \text{static forward slicing with respect to the variable } v \text{ at statement } s \text{ in the program } P \\
\( \overleftarrow{\mathbf{S}} \) \hspace{1cm} \text{backward slicing} \\
\( \overleftarrow{\mathbf{S}}(P, s, v) \) \hspace{1cm} \text{static backward slicing with respect to the variable } v \text{ at statement } s \text{ in the program } P
# Contents

Abstract

Acknowledgements

1 Introduction

1.1 Program Comprehension

1.1.1 Low-Level Analysis Based Techniques for Program Comprehension

1.1.2 High-Level Analysis Based Techniques for Program Comprehension

1.2 Motivation

1.2.1 High-Level Analysis Benefit from Low-Level Analysis

1.2.2 Low-Level Analysis Benefit from High-Level Analysis

1.3 Aims and Objectives of this Thesis

1.3.1 Concept Extension

1.3.2 Concept Abbreviation

1.3.3 Concept Refinement

1.4 Implementation

1.5 Contributions of the Thesis

1.6 Organisation of the Thesis

1

2

18

18

20

21

22

23

23

24

25

26

26

28

29

30
## Contents

### 2 Literature Survey

2.1 Introduction .................................................. 32

2.2 Concept Assignment .......................................... 32

2.2.1 Hypothesis-Based Concept Assignment (HB-CA) ........ 34

2.3 Dependence-Based Program Representation .................... 36

2.3.1 Control Flow Graph ....................................... 36

2.3.2 Data Flow Graph ......................................... 36

2.3.3 Program Dependence Graph ............................... 38

2.3.4 System Dependence Graph ................................. 39

2.4 Program Slicing .................................................. 41

2.4.1 The Concept of Program Slicing ............................ 41

2.4.1.1 Static Program Slicing ................................. 41

2.4.1.2 Dynamic Slicing ..................................... 42

2.4.1.3 Quasi static Slicing ................................... 44

2.4.1.4 Hybrid Slicing ........................................ 44

2.4.1.5 Simultaneous Slicing .................................. 46

2.4.1.6 Conditioned Slicing .................................. 50

2.4.1.7 Amorphous Slicing .................................... 51

2.4.1.8 Chopping .............................................. 52

2.4.2 Intraprocedural Program Slicing Algorithms .............. 54

2.4.2.1 Original Slicing Approach–dataflow Equations .... 54

2.4.2.2 Information-flow Relations ............................ 56

2.4.2.3 Graph Reachability .................................... 56

2.4.3 Interprocedural Program Slicing Algorithms .............. 58

2.4.3.1 Dataflow Equations .................................. 58

2.4.3.2 Information-flow Relations ............................ 59
CONTENTS

2.4.3.3 Graph Reachability ........................................ 59
2.4.4 Applications of Program Slicing .............................. 60
  2.4.4.1 Software Quality Assurance ............................... 60
  2.4.4.2 Software Maintenance and Re-engineering ............... 62
  2.4.4.3 Measurement .................................................. 63
2.5 Key Statement Analysis ........................................... 64
2.6 Dependence Clusters ............................................... 67
  2.6.1 Dependence Cluster Causes .................................. 67
  2.6.2 Dependence Clusters Identification ......................... 68
    2.6.2.1 Monotone Slice-size Graphs ............................ 69
    2.6.2.2 Characterising Dependence Cluster Size ............... 69

3 Experimental Design ................................................. 71
  3.1 Motivation ....................................................... 71
  3.2 Subject Programs ................................................ 72
  3.3 Analysis Tools ................................................... 74
  3.4 Statistical Analysis ........................................... 75
  3.5 Evaluation ....................................................... 75
  3.6 Threats to Validity ............................................. 76

4 Concept Extension Using Program Slicing ........................ 78
  4.1 Introduction ..................................................... 78
  4.2 Challenge ......................................................... 79
  4.3 Experimental Study .............................................. 80
    4.3.1 Motivation ................................................... 80
    4.3.2 Case Study Organisation .................................... 81
      4.3.2.1 Criteria Definitions ................................. 81
CONTENTS

4.3.2.2 Metrics for ECS ........................................ 84
4.3.3 Expected Results ........................................... 86
4.3.4 Method ..................................................... 87
4.4 Results and Discussion ....................................... 88
  4.4.1 Relationship between Criterion Coherence and Slice Coverage 88
  4.4.2 Relationship between Criterion Coherence and Slices Overlap 91
  4.4.3 Relationship between Criterion Size and Slice Size ........ 92
4.5 Summary ....................................................... 98

5 Concept Abbreviation Using Slice-based Dependence Analysis 100
  5.1 Introduction .................................................. 100
  5.2 Challenge ..................................................... 101
  5.3 The Framework of Parameterised KSA ......................... 102
    5.3.1 Extraction ............................................... 103
      5.3.1.1 Subroutine ........................................... 104
      5.3.1.2 High-Level Concepts ................................ 104
    5.3.2 Decomposition ............................................ 105
    5.3.3 Abbreviation .............................................. 106
    5.3.4 An Improved KSA_{BO} Algorithm ....................... 107
    5.3.5 Example .................................................. 109
  5.4 Key Statement Evaluation .................................... 110
    5.4.1 Metric 1: Impact .......................................... 110
    5.4.2 Metric 2: Cohesion ...................................... 111
  5.5 Experimental Study .......................................... 113
    5.5.1 Subject Programs ......................................... 113
    5.5.2 Analysis Tools ........................................... 114
    5.5.3 KSA Parameters Used in the Experiment ............... 114
## 5.6 Results and Discussion

### 5.6.1 Analysis Scope

- **5.6.1.1 Analysis Scopes Containing Principal Variable**
- **5.6.1.2 Analysis Scopes Containing Key Statement**

### 5.6.2 Principal Variable

- **5.6.2.1 Principal Variables in Functions**
- **5.6.2.2 Principal Variables in Concept Bindings**

### 5.6.3 Key Statements

- **5.6.3.1 Key Statement Size**
- **5.6.3.2 Impact**
- **5.6.3.3 Cohesion**

### 5.6.4 Key Statements and their Associated Analysis Scope

- **5.6.4.1 Impact**
- **5.6.4.2 Cohesion**

### 5.6.5 Summary of Results

### 5.7 Summary

## 6 Concept Refinement: Dependence Based Concept Assignment

### 6.1 Introduction

### 6.2 Challenge

### 6.3 Framework of Concept Refinement

### 6.4 Vertex Rank Model

- **6.4.1 Eigenvector Centrality**
- **6.4.2 Vertex Rank Model Algorithm**
- **6.4.3 Convergence of Computation**
- **6.4.4 Computational Complexity**

### 6.5 Source and Sink Identification Using the VRM
6.5.1 Concept Binding’s Source and Sink .............................. 164
6.5.2 Optimising Concept Binding’s Dependence Graph for the VRM 166
6.6 Evaluation .......................................................... 169
  6.6.1 Impact ............................................................ 169
  6.6.2 Cohesion .......................................................... 171
6.7 Experimental Study .................................................. 171
  6.7.1 Experimental Set-up .......................................... 171
  6.7.2 The VRM Implementation .................................... 172
  6.7.3 Results and Discussion ...................................... 172
    6.7.3.1 Size of Refined Concept Binding ....................... 173
    6.7.3.2 Impact ....................................................... 176
    6.7.3.3 Cohesion ................................................... 180
    6.7.3.4 Summary of Results ..................................... 181
6.8 Summary ................................................................ 183

7 Conclusions And Future Work ........................................ 184
  7.1 Summary of Thesis Achievements .............................. 184
  7.2 Conclusions .......................................................... 186
  7.3 Future Work .......................................................... 187
    7.3.1 Dependence Graph Refinement for VRM ................ 188
    7.3.2 Combination of HB-CA, Syntactic Analysis and Dependence Analysis ..................................................... 188
    7.3.3 Dependence Clusters ........................................... 188

Bibliography ................................................................. 189
List of Tables

2.1 Results of Weiser’s algorithm for the example program in Figure 2.2 with respect to slicing criterion (13, \{\text{sum}\}). ........................................ 55

3.1 Experimental Subjects used in the thesis. ............................... 72

4.1 Example Metrics Computations. ........................................... 85

4.2 The number of CA concepts for each experimental program. ...... 88

5.1 Experimental Programs. ..................................................... 113

5.2 The Summary of number of functions and concept bindings containing principal variables. ........................................... 118

5.3 The Summary of numbers of functions and concept bindings containing Key Statements. ........................................... 121

5.4 The mean and standard deviation of the number of principal variables in \( \mathbb{P}V_O \cup \mathbb{P}V_G \). ................................................... 126

5.5 The mean and standard deviation of the number of principal variables in \( \mathbb{P}V_O \cup \mathbb{P}V_G \). ................................................... 129

5.6 The Summary of average key statement size (as a percentage of scope size) for functions and concept bindings respectively. ........... 130

5.7 Wilcoxon Matched-Pairs Signed Ranks Test comparing Impact of key statements with their associated function and concept binding. ... 141

5.8 Wilcoxon Matched-Pairs Signed Ranks Test comparing Cohesion of key statements with their associated function and concept binding. .. 145

6.1 The corresponding SDG vertices for the area concept binding in Figure 6.2. ................................................................. 155
6.2 Wilcoxon Matched-Pairs Signed Ranks Test comparing Impact of HB-CA concept bindings with their refined concept bindings. . . . . . . . . 178

6.3 Wilcoxon Matched-Pairs Signed Ranks Test comparing Size of HB-CA concept bindings with their refined concept bindings. . . . . . . . . 179

6.4 Wilcoxon Matched-Pairs Signed Ranks Test comparing Impact of HB-CA concept bindings with their refined concept bindings. . . . . . . . . 180
List of Figures

2.1 A simple HB-CA example that includes three concepts related to sound volume. Sound pressure level (SPL) describes the relative intensity of a sound. .................................................. 35

2.2 An example of Control Flow Graph ......................................................... 37

2.3 An example of Program Dependence Graph .............................................. 40

2.4 A static backward slice example taken with respect to the variable \textit{sum} at the last statement. ................................................................. 42

2.5 A dynamic slice example taken with respect to the variable \textit{sum} at the last statement on the input 0 (\texttt{current}=0). .................................................. 43

2.6 A quasi-static backward slice example taken with respect to the variable \textit{result} at the last statement on any input stream of the form \texttt{<0...>}. ................................................................. 45

2.7 A simultaneous slice example taken with respect to the variable set \{i, sum\} at the last statement. ................................................................. 47

2.8 The Executable Concept Slicing Algorithm ................................................ 49

2.9 An example Conditioned slice taken with respect to the variable grade at the last statement on the condition $F_0 \equiv (mark \geq 35 \land mark < 40)$. 51

2.10 An example showing Amorphous Slicing producing thinner slices by removing syntactic restrictions. Slice computed w.r.t. variable \texttt{slice} at end. ................................................................. 52

2.11 A chopping example taken with respect to the boxed variables pair \{\texttt{current}, \texttt{sum}\}. ................................................................. 53

2.12 The intraprocedural program slicing algorithm ......................................... 57

2.13 The original flowchart for Key Statement Analysis .................................... 65
LIST OF FIGURES

2.14 Key Statement Analysis ‘Bieman-Ott’ Style[47] .......................... 65
2.15 Key Statement Analysis ‘Ball-Eick’ Style[47] .......................... 66
2.16 Sample of MSG ................................................................. 70

4.1 Example of the four types of criteria. ................................. 83
4.2 Average Coverage value comparison for CA and RN. .......... 89
4.3 Average Coverage for CA, CV, and CL .............................. 90
4.4 Box plot of sizes for each type of random criterion. ........... 91
4.5 Average Overlap value for each type of criterion. ............... 91
4.6 Two samples of scatter plots for slice size on CA criterion in Figures 4.7 and 4.8 .................................................. 94
4.7 Scatter plots to show the relationship between the size of criterion and the size of slices for all four types of criteria for the subjects without large dependence clusters. ................................. 95
4.8 Scatter plots to show the relationship between the size of criterion and the size of slices for all four types of criteria for the subjects with large dependence clusters. ................................. 96

5.1 The Framework of Parameterised Key Statement Analysis .... 103
5.2 KSA_{BO} Algorithm for identifying those statements that effect all of an analysis scope’s principal variables. .......................... 108
5.3 An example of key statement computed by KSA_{BO} algorithm for the concept binding. .................................................. 109
5.4 The value of parameters used in the experiment and their combinations .... 116
5.5 The number of functions with and without principal variables ..... 119
5.6 The number of concept bindings with and without principal variables 119
5.7 The number of concept bindings having principal variables and key statements in three pairs respectively for all programs ...... 123
5.8 The number of principal variables per function for each of the three kinds of principal variables. ................................. 125
5.9 The number of principal variables per concept binding for each of the three kinds of principal variables. .......................... 128
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.10</td>
<td>The average key statements size in functions for all programs</td>
<td>131</td>
</tr>
<tr>
<td>5.11</td>
<td>The average key statements size in concept bindings for all programs</td>
<td>131</td>
</tr>
<tr>
<td>5.12</td>
<td>Average Impact value for each type of key statement in functions for all programs</td>
<td>134</td>
</tr>
<tr>
<td>5.13</td>
<td>The boxplot of Impact in functions for all programs</td>
<td>134</td>
</tr>
<tr>
<td>5.14</td>
<td>Average Impact value for each type of key statement in concept bindings for all programs</td>
<td>135</td>
</tr>
<tr>
<td>5.15</td>
<td>The boxplot of Impact in concept bindings for all programs</td>
<td>135</td>
</tr>
<tr>
<td>5.16</td>
<td>Average Cohesion value for each type of key statement in functions for all programs</td>
<td>138</td>
</tr>
<tr>
<td>5.17</td>
<td>The boxplot of Cohesion in functions for all programs</td>
<td>138</td>
</tr>
<tr>
<td>5.18</td>
<td>Average Cohesion value for each type of key statement in concept bindings for all programs</td>
<td>139</td>
</tr>
<tr>
<td>5.19</td>
<td>The boxplot of Cohesion of key statements in concept bindings for all programs</td>
<td>139</td>
</tr>
<tr>
<td>5.20</td>
<td>The average Impact for KS\textsubscript{Union} and the associated functions. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)</td>
<td>142</td>
</tr>
<tr>
<td>5.21</td>
<td>The average Impact for KS\textsubscript{Union} and the associated concept bindings. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)</td>
<td>142</td>
</tr>
<tr>
<td>5.22</td>
<td>The average Cohesion for KS\textsubscript{Union} and the associated concept bindings. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)</td>
<td>144</td>
</tr>
<tr>
<td>5.23</td>
<td>The average Cohesion for KS\textsubscript{Union} and the associated concept bindings. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)</td>
<td>144</td>
</tr>
<tr>
<td>6.1</td>
<td>The Framework of Concept Refinement</td>
<td>151</td>
</tr>
<tr>
<td>6.2</td>
<td>An example of dependence based concept binding.</td>
<td>154</td>
</tr>
<tr>
<td>6.3</td>
<td>The dependence graphs for the area concept binding in Figure 6.2. The solid arrows represent data flow dependence edges and dashed arrows represent control dependence edges.</td>
<td>156</td>
</tr>
</tbody>
</table>
6.4 The VRM algorithm ......................................................... 160
6.5 An example of initial weights and stable weights assigned to vertices and edges ......................................................... 160
6.6 An example of initial dependence graph and amended dependence graph with distribution ratio matrix \((p = 0.85)\) ......................................................... 163
6.7 Examples tree-root-like and tree-branch-like control dependence graphs ......................................................... 168
6.8 The size of refined concept bindings over all concept bindings ......................................................... 173
6.9 The number of refined concept bindings that are same as the original HB-CA concept bindings by programs ......................................................... 174
6.10 The average size of refined concept bindings per program ......................................................... 175
6.11 Ratio of \(\text{Impact}(\text{RCB}, P)\) to \(\text{Impact}(\text{CB}, P)\) over all concept bindings ......................................................... 176
6.12 Average \(\text{Impact}\) value for HB-CA concept binding (CB) and refined dependence based concept binding (RCB) ......................................................... 177
6.13 Boxplot \(\text{Impact}\) value for HB-CA concept binding (CB) and refined dependence based concept binding (RCB) ......................................................... 178
6.14 Average \(\text{Cohesion}\) value for CB and RCB ......................................................... 180
6.15 Boxplot \(\text{Cohesion}\) value for CB and RCB ......................................................... 181
Chapter 1

Introduction

This thesis presents three approaches using slice-based dependence analysis to extend, abbreviate and refine the concept for program comprehension. In the thesis, a framework for the three approaches is described. The associated experimental studies are designed and implemented based on 10 C programs including industry code and open source. The evaluation augmented by a statistical analysis are presented to show the feasibility of the approaches.

This chapter presents some classical techniques for program comprehension and explains the need for the combination of approaches. The experimental set-up shows the programs used in the experimental studies and analysis tools employed. Lastly, this chapter presents an overview of the thesis structure.

1.1 Program Comprehension

Program comprehension plays an important role in nearly all software related tasks. It is vital to maintenance and reuse activities. Such activities are hard to perform
without a thorough and correct understanding of the component to be maintained or reused. Program comprehension is also indispensable for improving the quality of software development activities such as code reviews, debugging, and some testing approaches. All these development activities require programmers to read and understand programs. In software maintenance, it is estimated that up to 60% of the total time allotted is spent on program comprehension [32].

Several cognitive models have been suggested which attempt to explain how a software engineer understands code. These include:

1. Bottom-up model - group together small chunks of source code to build up higher levels of abstraction. These abstractions are then grouped together to eventually produce a high level design of a program [114, 115]. The bottom-up model is usually used when the code is new to the programmer.

2. Top-down model - a global hypothesis is defined, describing the program. The reader then tries to verify this hypothesis. This can cause further hypotheses to be created, building up a hierarchy of hypotheses to be verified. This continues until a level is reached where the hypothesis can be verified or proven to be false [28]. The top-down model is used when code under consideration is familiar.

3. Systematic model - a programmer systematically reads through the code in detail, looking at both the control-flow and data-flow abstractions [91].

4. As-needed model - a programmer only looks at the code related to a particular problem or task. Parts of the code are only looked at when the programmer needs to understand them [91].

5. Integrated model - combination of both top-down and bottom-up model. When
1.1. Program Comprehension

code is familiar, the integrated strategy uses the top-down model, and when the code is unknown, the bottom-up strategy is used [120].

Research has suggested that there is no single cognitive model that can explain the behaviour of all programmers, and that it is more likely that programmers, depending on the particular problem, will swap between models [117].

Due to the importance and high cost of program comprehension, a large amount of research has been carried out, in an attempt to guide and support software engineers in this process [28, 91, 112, 114, 120]. Within these efforts, substantial interest is usually directed towards the specific topic of automated assistance to software comprehension.

The main task in program comprehension is the identification of components and the reconstruction of the essential relations between them. A variety of analysis based techniques were proposed either at domain knowledge based high-level or at source code based low-level.

1.1.1 Low-Level Analysis Based Techniques for Program Comprehension

Low-level analysis represents the details of source code and analyses their relations, such as control and data flow analysis and dependence analysis [93].

Control and data flow analysis are based on the representation of Control Flow Graph (CFG). Control flow analysis is used to identify the basic blocks and loops in a procedure. Data flow analysis answers the questions that “Where is this variable defined?” or “where is this variable definition used?”, i.e., reaching definitions or use-def chaining.
Program dependency is a type of relationship among the operations in a program, including control dependence and data dependence. Program slicing is a dependence analysis based technique for extracting executable sub-programs based on criteria of interest (usually a set of statements and a set of variables) [121, 123]. The slice of a program for a set of variables at a particular line of the program is that part of the program responsible for giving a value to the variable at that location.

The simplification offered by program slicing has led to its application in program comprehension. De Lucia, Fasolino and Munro [36] used conditioned slicing to facilitate program comprehension. Harman et al. [61] claimed that the quasi-static slicing [119] appears to be most suited to program comprehension as it allows a program to be understood as a set of projections, each of which capture the effect of the program when executed in a set of possible initial states, allowing analysis by cases. Amorphous program slicing [59], which relaxes the syntactic constraint of traditional slicing, is also of much use in program comprehension [26, 58].

1.1.2 High-Level Analysis Based Techniques for Program Comprehension

High-level analysis retrieves a knowledge-based view that is more abstract than the source code, such as pattern recognition and concept assignment [93].

Pattern recognition [66, 79] attempts to detect the design patterns in the source code in order to understand the intended high-level connector.

Concept assignment [15] aims to allocate specific high-level domain concepts to sections of source code. There has been much work on extraction of the code that expresses domain level concepts [8, 15, 52, 96]. This work identifies the portions of code that correspond to the domain level concepts. It searches the source code for
the associations that express the domain level concept in a collection of source level statements. The concept extraction process is useful in source code comprehension, because comprehension tasks are often couched in terms of domain level concepts. Therefore, it can be helpful to identify the corresponding source code, particularly for code which may not be completely familiar to the programmer.

Gold [50] introduced the Hypothesis-Based Concept Assignment (HB-CA) to address the concept assignment problem. This method is computationally inexpensive, having linear computational growth in the length of the source code under analysis. It performs concept assignment, primarily using semantic information such as identifier names and comments, and contains an approach to analysing monolithic and poorly structured code using a self-organising neural network [80].

1.2 Motivation

As discussed in Section 1.1, both low-level analysis and high-level analysis techniques are efficient approaches of automated source code extraction for supporting program comprehension. However, each has a weakness which prevents wider application.

Low-level analysis based techniques usually make use of some kind of source code representation graph where the individual source code elements are interconnected with different dependencies. These graphs can be more or less precise depending on the accuracy of the low-level analyses. In some cases, however, for any non-trivial real-world program, size can make the generation and storage of source code representation graph an expensive process, and the low-level analysis is still much too detailed to provide the developer with any architectural insight [116].

High-level analyses based techniques usually deal with structural information, by
which optional structures that are likely to represent the actual architecture are provided. However, many of them are published and comparative studies between different techniques do not favour a specific one [83]. The situation in high-level analysis that the proposed reasonable options and final decisions are up to a systems engineer causes little support for the comparative assessment of the techniques.

Overall, each individual analysis technique at either low-level or high-level outperforms the others in certain situations. The lack of a general method triggers the demand for a reasonable combination of the single analyses between high-level and low-level which can benefit both by overcoming the weaknesses of each other.

1.2.1 High-Level Analysis Benefit from Low-Level Analysis

The accuracy of the high level analysis depends on the accuracy of the underlying low level analysis. For example for the current concept assignment techniques (e.g. HB-CA), concepts have to be retrieved with lexical and syntax analyses. Furthermore, for high-level analyses requiring behavioral information one would expect low level dependence analyses to be involved. Integrated dependence analyses which results in dependence based concept — if affordable in time and memory consumption — could also improve the quality of the high level analysis.

1.2.2 Low-Level Analysis Benefit from High-Level Analysis

The main limitation of the low level analyses are memory and time consumption on the scalability. However, for reasonable-size analysis scope, such as procedure and class, this is not a problem. Both flow analysis and dependence analysis are efficient on them [70, 107]. The identification of reasonable-size hot-spots (interesting parts) in the system is necessary, which, however, depends on the high-level analysis.
1.3 Aims and Objectives of this Thesis

Instead of inspecting the whole system on the first try, one could pick up some hotspots, such as concepts identified using concept assignment. This could cut down a system first and enable more advanced, low-level analyses to be applied to identified concepts.

1.3 Aims and Objectives of this Thesis

The aim of this thesis is to investigate the possibility of integration techniques between low-level analysis and high-level analysis for program comprehension. Program slicing based dependence analysis and concept assignment are employed as the representative of low-level and high-level analysis techniques.

Program slicing is a technique for extracting executable sub-programs based on criteria of interest (usually a set of statements and a set of variables) [121, 123]. The simplification offered by program slicing has led to its application in a number of software engineering tasks, for example, debugging [2, 73, 95, 124], restructuring [30], reuse [10, 34, 89] and program comprehension [36]. One drawback common to all slicing techniques is the low level at which a slicing criterion is specified. A software engineer requires significant internal knowledge of the code (in particular the program’s variables and the locations of their use) before slicing can be effectively employed.

Concept assignment aims to allocate specific high-level meaning to specific parts of a program. This technique can be used to identify program fragments associated with a particular concept in a program. For example, the code associated with computing the tax on a sale. However, the extracted code is not executable as a separate subprogram in its own right.
This thesis presents three types of combination approaches: concept extension, concept abbreviation and concept refinement. Essentially, these techniques use slice-based dependence analysis to extend, abbreviate and refine the concept bindings produced by concept assignment for program comprehension. The first two combinations were first proposed by Harman et al. [62]. However, these proposals have, hitherto, remained merely proposals; there are neither experimental studies to show the feasibility of the approach, nor the assessment of the extracted code.

1.3.1 Concept Extension

Extension: process or action of extending

Extend: make (something) longer or larger (in space or time)

Oxford Advanced Learner’s Dictionary

Concept extension uses program slicing to ‘extend’ a concept binding by tracing its dependencies. The slicing makes concept binding larger in size in order to make it executable, thus it is named concept extension.

Executable Concept Slicing (ECS) [47, 62] is a framework that unifies program slicing and concept assignment for higher-level executable source code extraction. However, the applicability of any code extraction technique for software engineering crucially depends upon the size of the subprogram extracted. From this point of view, one might suppose that ECS is simply impractical, because the union of several slices will tend to extract large amounts of code; perhaps the entire program in many cases. Therefore, experimental studies on the size of ECS are necessary to be implemented to examine if the ECS size is acceptable in practice.
1.3. Aims and Objectives of this Thesis

1.3.2 Concept Abbreviation

Abbreviation: abbreviating or being abbreviated
Abbreviate: shorten (a word, phrase, etc), esp. by omitting letters

Oxford Advanced Learner’s Dictionary

Concept abbreviation uses a slicing-based technique to shorten a concept binding without losing the computational impact of the concept. The resulting statements contribute more to the computation embodied by the binding than others, namely key statements. As the set of key statements is smaller in size than the original concept binding, this technique is named concept abbreviation.

The original notion of Key Statement Analysis (KSA) [47, 62] was to identify some important statements that contribute more to the computation embodied by the concept. There are no experimental studies to investigate feasibility and practicability of the key statement.

1.3.3 Concept Refinement

Refinement: refining or being refined
Refine: remove impurities from (something);
        improve (something) by removing defects and attending to detail

Oxford Advanced Learner’s Dictionary

Based on the experimental studies for ECS and key statements analysis, a new combined approach – concept refinement, is proposed, to refine the concept binding. It not only takes advantage but also overcomes disadvantage of both ECS and KSA.
Using slicing-based dependence analysis, this technique can remove non-concept-
dependent statements from a concept binding. As the remaining statements are
highly dependent without losing domain knowledge, this technique is named concept
refinement.

The three combination approaches have tackled the problem of program mainte-
nance. For example, an engineer can locate source code related to the target using
class concept assignment. If the concept binding is still quite large, Concept Abbreviation
that locates the key statements can help to focus attention more rapidly on the core
of a concept binding, and Concept Refinement can remove the non-concept related
statements within the binding. Once the code is changed as required by the task,
Concept Extension can build the executable slice with respect to the concept binding,
of which the test cost would be reduced compared to testing the whole program.

The overall aim encompasses the following general objectives:

1. To implement, compare and analyse the current combination approaches Ex-
ecutable Concept Slicing (ECS) and Key Statement Analysis [62, 47] in real
C programs.

2. To (re-)define metrics to evaluate and examine the feasibility of these combi-
nation approaches based on experimental studies.

3. To propose a new type of combination approach – dependence based concept
assignment (concept refinement).

4. To compare the new combination approach with previous ones.

5. To analyse the effects of dependence clusters on all three combinations.
1.4 Implementation

To implement the three types of combination approaches of concept assignment and slicing based dependence analysis, the data of concept bindings and slices need to be produced first. WeSCA is an HB-CA implementation tool that identify concept bindings in the source programs of C and COBOL [47]. There are corresponding slicers available for both program languages. CodeSurfer [53], a commercial tool developed by Grammatech for C is employed in the study. Therefore, the subjects used in experiments are limited to C programs. The subject programs need to cover various types of code, such as industry code and open source, without and with large dependence clusters etc. We choose the programs from the previous research on locating dependence clusters [21], which categorises the subjects as ‘with’ or ‘without’ large dependence clusters.

HB-CA needs a knowledge base for each program that involves human knowledge and work. In program maintenance, the knowledge base is usually built to only cover the concepts directly relevant to targets. In the experiments of this research, in order to eliminate the bias of specific concepts, it is necessary to build the whole knowledge base to cover various types of concepts for the subjects, which demands that the program size cannot be too large. In this thesis, the subject size ranges between 3KLoC and 29KLoc. However, HB-CA has a linear computation cost with the growth of program size, so the combination approaches presented in the thesis can be scaled to large programs.

Dependence analysis and slices are constructed using CodeSurfer. Three algorithms are designed for the three combination approaches. The most important benefit of a combination approach is that high level analysis can provide a reasonable size analysis scope for low level analysis. In this thesis, the analysis scope for three
approaches is HB-CA concept binding, of which the size is smaller than a procedure, so the computation cost of each algorithm would not be expensive. The algorithm complexity analysis accompanies with the algorithm description in each substantive chapter.

Informally, dependence clusters can be thought of as strongly connected statements in the dependence graph. From the standpoint of a dependence analysis, such as slicing, including any part of a dependence cluster causes the inclusion of the dependence cluster. Large dependence clusters have been defined as those that include more than 10% of the program [21]. Because they tend to affect any and all slicing based techniques, the effects of large dependence clusters need to be investigated for the combination techniques.

1.5 Contributions of the Thesis

The contributions of this thesis are as follows:

1. The proposal of a conceptual framework for three combination approaches between concept assignment and program slicing which extend, abbreviate and refine the concept binding using dependence analysis.

2. The proposal of a new approach for dependence based concept refinement: the Vertex Rank Model (VRM).

3. Experimental demonstration that dependence based concept refinement is better than the other two combination approaches.

4. The introduction of dependence based metrics that evaluate high-level source code extractions.
1.6 Organisation of the Thesis

The rest of the thesis is organised as follows:

- **Chapter 2 Literature Survey** reviews the concepts used throughout this work, namely concept assignment, program slicing, key statement analysis and dependence cluster analysis.

- **Chapter 3 Experimental Design** gives the details of selected subjects, analysis tools and statistical analysis approaches. Threats to validity are also discussed in this chapter.

- **Chapter 4 Concept Extension Using Program Slicing** presents the experimental study on concept slicing, which uses program slicing to extend concepts. The experimental study focuses on the size problem, and the results given provide the evidence that slice size with respect to concepts are smaller than those arbitrary criteria with similar size as concepts but with less conceptual cohesion.

- **Chapter 5 Concept Abbreviation Using Slice-based Dependence Analysis** presents a framework on three staged parameterised key statement analyse. This approach uses slicing based dependence analysis, to identify the key statement in an analysis scope. The possible parameter ‘values’ for each stage are discussed. The experimental results based on calculation of two metrics (impact and cohesion) investigate the feasibility of the approach.

- **Chapter 6 Concept Refinement: Dependence Based Concepts** describes a new framework on concept refinement using slice based dependence analysis that produce dependence based concepts. The Vertex Rank Model is presented to identify the source and sink for a concept; chopping is employed to identify
dependence based concepts from the original concepts. The implementation of experiments is presented and the result is analysed statistically.

- Chapter 7 **Conclusions and Future Work** concludes this Thesis. Here, the outcome of and answer to the research questions that this work sets out to investigate are summarised. Furthermore, the suggested areas of direction for future research are discussed.
Chapter 2

Literature Survey

2.1 Introduction

This chapter reviews work in the field of Concept Assignment and program slicing. It also reviews the concept of dependence clusters which is known to affect most slicing based techniques.

2.2 Concept Assignment

Concept assignment aims to allocate specific high-level meaning to specific parts of a program. This technique can be used to identify program fragments associated with a particular concept in a program. For example, the code associated with computing the tax on a sale could be a concept. The concept assignment problem was defined by Biggerstaff et al. as follows [15]:

Concept assignment is a process of recognising concepts within a computer program and building up an “understanding” of the program by
2.2. Concept Assignment

relating the recognised concepts to portions of the program, its operational context and to one another.

Intelligent agents (tools) that attempt to assign descriptions of computational intent to source code can be used for concept assignment. Biggerstaff et al. [15] claim that research on intelligent agents can be divided into 3 distinct approaches:

1. Highly domain specific, model driven, rule based question answering systems
   These systems depend on a manually populated data base describing the software system, typified by the Lassie system [39].

2. Plan driven, algorithmic program understanders or recognisers
   Two examples of this type are Programmer’s Apprentice [110] and GRASPR [125].

3. Model driven, plausible reasoning understanders
   DESIRE’S DM-TA0 subsystem [15, 16] and IRENE [75] are two examples.

Approaches 1 and 2 have the desirable characteristics that they are good at faithfully and completely deriving concepts within small-scale programs but suffer from the problem of not being able to deal readily with large-scale programs because the inference chains or parsing procedures tend to become computationally infeasible in the face of overwhelming numbers of details. Conversely, approach 3 can easily handle large-scale programs and their computational growth appears to be linear in the length of the program under analysis. They, of course, suffer the converse problem in that their results are approximate and imprecise.
2.2. Concept Assignment

2.2.1 Hypothesis-Based Concept Assignment (HB-CA)

Hypothesis-Based Concept Assignment (HB-CA) [52] is a plausible-reasoning approach that addresses the first part of Biggerstaff et al.’s concept assignment problem (recognising concepts and relating them to portions of the program). HB-CA is a three-stage process, Hypothesis Generation, Segmentation, and Concept Binding. The process requires a knowledge base of concepts to drive the analysis. This is a semantic network that contains two node types: the concepts in which the programmer may be interested and the indicators that may indicate the presence of a concept in the program (e.g., words that might be found in the source code). The knowledge base is constructed by a software engineer in advance of executing HB-CA (this is not a particularly difficult task since the relationships between concepts and indicators are quite simple). The concept may take the form of an Action or Object and the indicators can be possible string fragments of identifiers, keywords, or comments.

In the hypothesis-generation stage, the source code is taken as input and scanned for indicators of the various concepts in the knowledge base. A hypothesis for a concept is generated for each matching indicator. All hypotheses are sorted by their indicator position in the source code.

In the segmentation stage, the hypothesis list is analysed to group hypotheses into segments, initially using natural segment boundaries such as procedure boundaries. Following this, a self-organising map can be used to create non-overlapping partitions of high conceptual focus [52]. The output of this stage is a collection of segments, each containing a number of hypotheses.

In the concept-binding stage, the segments are scored in terms of concept occurrence frequency. Disambiguation rules select a concept where several different hypotheses
2.2. Concept Assignment

<table>
<thead>
<tr>
<th>Concept</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>(\text{sum} = 0;)</td>
</tr>
<tr>
<td>sum</td>
<td>(\text{for(i=0; i&lt;N; i++)})</td>
</tr>
<tr>
<td>sum</td>
<td>(\text{sum += Sound[i];})</td>
</tr>
<tr>
<td>spl</td>
<td>(\text{effective_SPL} = 20 \times \log(\sqrt{\text{sum}}))</td>
</tr>
<tr>
<td>average</td>
<td>(\text{average} = \text{sum} / N)</td>
</tr>
</tbody>
</table>

Figure 2.1: A simple HB-CA example that includes three concepts related to sound volume. Sound pressure level (SPL) describes the relative intensity of a sound.

Figure 2.1 provides a simple HB-CA example in which the domain model includes indicators for three concepts: \(\text{sum}\) indicates the computation of the \(\text{sum}\) concept (of a collection of sound frequencies), \(\text{spl, pressure, level}\) indicate the computation of a sounds effective sound pressure level (\(\text{spl}\)) concept, and \(\text{average}\) indicates the computation of the \(\text{average}\) concept (the average volume of the sound’s frequencies).

The first and third statements include an indicator for the \(\text{sum}\) concept. The penultimate statement includes an indicator for the \(\text{spl}\) concept, and the final statement includes an indicator for both the \(\text{average}\) concept and the \(\text{sum}\) concept. In this simplified example, Lines 1-3 are assigned to the \(\text{sum}\) concept, Line 4 to the \(\text{spl}\) concept and Line 5 to the \(\text{average}\) concept. Note that HB-CA generally assigns to regions rather than single lines and the results of HB-CA are not guaranteed to be executable.
2.3 Dependence-Based Program Representation

2.3.1 Control Flow Graph

A Control Flow Graph (CFG) [5] is a directed graph where the nodes represent basic blocks and directed edges represent possible immediate transfer of control from one basic block to another. A basic block is a linear sequence of program instructions having one entry point and one exit point. The CFG also includes two special blocks corresponding to the entry and exit of the flow graph.

A CFG is a static representation of the program, and represents all alternatives of control flow. CFG is extensively used for code optimisation and testing. It is also used to perform data flow analysis, e.g. reaching definitions or use-def chaining, in the program [6, 7]. Figure 2.2 shows a function written in C and its corresponding CFG.

2.3.2 Data Flow Graph

A data flow graph [104, 105] is a directed graph that represents the data dependence at operator level. Dennis’ work [38] opened up the area of data flow computation, in which two types of nodes are called links and actors. Actors describe operations, while links receive data from a single actor and transmit values to one or more actors by way of edges. Edges can be control or data edges.

Data flow graphs have been used successfully in the simulation of computer system. Because of the hierarchical nature and the modularity of data flow graphs, both software tasks and hardware units can be modelled in a uniform way using data flow graphs [76, 78].
void main(void) {
    int current;
    int sum = 0;
    int i = 0;
    do {
        printf("Enter an integer > ");
        scanf("%d", &current);
        if (current > 0) {
            sum = sum + current;
            i = i + 1;
        }
    } while (current > 0);
    printf("Sum of %d numbers is %d\n", i, sum);
}

Figure 2.2: An example of Control Flow Graph

A formal definition of data flow graph based on Dennis's model [38] was presented by Kavi et al. [77] as follows:

**Definition 2.1** A data flow graph is a bipartite labelled graph where the two types of nodes are called actors and links.

\[ G = \langle A \cup L, E \rangle \quad (2.1) \]

where

\[ A = a_1, a_2, \ldots, a_n \] is the set of actors

\[ L = l_1, l_2, \ldots, l_n \] is the set of links

\[ E \subseteq (A \times L) \cup (L \times A) \] is the set of edges.
2.3. Dependence-Based Program Representation

Actors represent functions and links are treated as place holders of data values (tokens) as they flow from actors to actors. Edges are the channels of communication (like arcs in Dennis’ model).

2.3.3 Program Dependence Graph

A Program Dependence Graph (PDG) [42, 69, 86] explicitly represents both the essential control and data dependencies in a program. A PDG consists of a control dependence sub-graph and data dependence sub-graph. The control dependence sub-graph represents essential control relations but excludes the fixed sequencing of operations of CFG. The data dependence sub-graph represents possible flow of data values from one statement to another.

The PDG is a directed graph whose nodes are connected by several kinds of edges. The nodes represent statements and predicate expressions and the edges incident to a node represent both control and data dependence. The PDG is mostly used for code optimisation, e.g. parallelism detection, loop fusion, clone detection etc. It is also used for performing slicing for maintenance and re-engineering purpose.

Horwitz et al. [70] used and redefined the PDG for computing program slices. Their definition shared the feature of explicitly representing both control and data dependences. Similarly, the vertices of PDG represent the assignment statements and control predicates that occur in program. In addition, three other categories of vertices are included, entry vertex, some initial definition of variable vertices and some final use of variable vertices. An edge represents either a control dependence or a data dependence. The control dependence edges must start from an entry vertex or a predicate vertex and are labelled either true or false. Two kinds of data dependence edges are represented, flow dependences and def-order dependences.
2.3. Dependence-Based Program Representation

There can be more than one data dependence edge between any two vertices.

Figure 2.3 shows a function \texttt{sum} written in C and its corresponding PDG.

2.3.4 System Dependence Graph

The System Dependence Graph (SDG) [70] is an extension to a PDG for programs with multiple procedures. The SDG is constructed by connecting the individual PDG of each procedure with additional vertices and edges.

For each call statement, there is a \texttt{call-site} vertex in the SDG as well as \texttt{actual-in} and \texttt{actual-out} vertices that model the copying of actual parameters to/from temporary variables. Each procedure (program) dependence graph has an entry vertex, and \texttt{formal-in} and \texttt{formal-out} vertices to model copying of formal parameters to/from temporary variables. \texttt{Actual-in} and \texttt{actual-out} vertices are control dependent on the \texttt{call-site} vertex; \texttt{formal-in} and \texttt{formal-out} vertices are control dependent on the procedure’s entry vertex.

These additional edges are of three sorts: (1) a \texttt{call} edge is added from each call-site vertex to the corresponding procedure entry vertex; (2) a \texttt{parameter-in} edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; (3) a \texttt{parameter-out} edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site. Call edges are a new kind of control dependence edge; parameter-in and parameter-out edges are new kinds of data dependence edges.
2.3. Dependence-Based Program Representation

int sum (int n)
{
    int sum=0;
    int i=0;
    while (i < n){
        sum = sum+i;
        i = i+1;
    }
    return sum;
}
2.4 Program Slicing

Program slicing, which was first introduced by Mark Weiser [121], is a decomposition technique that extracts program statements relevant to a particular computation from a program. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as slicing criterion. The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion are called the program slice. The process of computing program slices is called program slicing. Program slicing is found helpful and useful in program debugging, software testing, program comprehension, software maintenance, automatic parallelisation and program integration.

2.4.1 The Concept of Program Slicing

2.4.1.1 Static Program Slicing

The original form of program slicing defined by Mark Weiser is, in fact, a form of executable backward static slicing, which extracts from a program an executable subprogram composed of the statements relevant to the computation of particular variable $v$ at statement $s$ [121, 123]. The pair $< s, v >$ is referred to as a slicing criterion. “Executable” means that the slice can be compiled and run. Running the program and its slice produces the same sequence of values for $v$ at statement $s$. A backward slice (denoted by $\overleftarrow{S}$) consists all statements that the computation that the slicing criterion may depend on, while a forward slice (denoted by $\overrightarrow{S}$) includes all statements depending on the slicing criterion. “Static” means that only statically available information is used for computing slices, i.e. slices are computed as the solution to a static analysis programs.
2.4. Program Slicing

Figure 2.4 shows a simple program that computes the sum of input integers. The slice, highlighted in bold, is taken with respect to the final value of the variable \texttt{sum}.

```c
void main(void){
    int current;
    int sum = 0;
    int i = 0;
    do{
        printf("\nEnter an integer: ");
        scanf("%d", &current);
        if (current > 0){
            sum = sum + current;
            i = i + 1;
        }
    }while (current > 0);
    printf("\nThe sum of %d numbers is %d\n", i, sum);
}
```

Figure 2.4: A static backward slice example taken with respect to the variable \texttt{sum} at the last statement.

2.4.1.2 Dynamic Slicing

Korel and Laski introduced the notion of dynamic slicing [81]. A \textit{dynamic slice} preserves the program behavior for a specific program input, rather than that for the set of all inputs. A dynamic slice is an executable part of the program whose behavior is identical, for the same program input, to that of the original program with respect to a particular variable \(v\) at statement \(s\). Therefore, a dynamic slice is constructed with respect to three pieces of information, a variable \(v\) at statement \(s\), on an input \(x\). The tuple \(<s, v, x>\) is referred to as a \textit{dynamic slicing criterion}.

Due to the run-time handling of arrays and pointer variables, dynamic slicing algo-
2.4. Program Slicing

Algorithms can result in more precise slices than static ones. Due to taking a particular program execution into account, dynamic slicing may significantly reduce the size of the slice compared to static slicing. These are helpful in program debugging.

Figure 2.5 shows a dynamic slice for the program in Figure 2.4 and the slice highlighted in bold taken with respect to the final value of the variable \texttt{sum} on the input 0.

```c
void main(void){
    int current;
    \textbf{int sum = 0;}
    int i = 0;
    \textbf{do{\n        printf("\nEnter an integer: ");\n        scanf("%d", &current);\n        if (current > 0){\n            sum = sum + current;\n            i = i + 1;\n        }\n    }while (current > 0);\n    printf("\nThe sum of \%d numbers is \%d\n", i, sum);\n}
```

Figure 2.5: A dynamic slice example taken with respect to the variable \texttt{sum} at the last statement on the input 0 (\texttt{current=0}).

Agrawal et al. [4] extended dynamic slicing to relevant slicing. A relevant slice with respect to a variable contains not only the statements that have an influence on the variable but also those executed statements that did not affect the output, but could have affected it had they evaluated differently.

Hall [57] extended dynamic slicing and simultaneously applied it to a set of test
cases rather than just one test case. This slicing method is called *simultaneous
dynamic program slicing* which extracts a set of executable program subsets. The
basic approach is to apply any kind of dynamic slicing algorithm that meets certain
criteria (one of which is to be able to produce executable slices) and incrementally
builds the simultaneous slice using an iterative algorithm for all test cases.

### 2.4.1.3 Quasi static Slicing

Venkatesh introduced *Quasi static slicing* [119] which is the first attempt to define
a hybrid slicing method ranging between static and dynamic slicing. A quasi-static
slice is constructed with respect to an initial prefix of the input sequence to the
program. The quasi static slicing is used for the applications of analysing the be-
aviour of the program when the value of some input variables is fixed while other
input values vary. In the case all variables are unconstrained, the quasi static slice
coincides with a static slice, while when the values of all input variables are fixed,
the slice is a dynamic slice.

Figure 2.6 shows a quasi-static backward slice and the slice highlighted in bold taken
with respect to the final value of the variable *result* on any input stream of the
form \(<0\ldots>\).

### 2.4.1.4 Hybrid Slicing

As static slicing suffers from the problem of imprecision and dynamic slicing is
specific to only one execution, Gupta et al. [55, 56] presented a *hybrid slicing* tech-
nique for improving the precision and quality of static slices by integrating a limited
amount of dynamic information into a static slicing analysis. The hybrid slice is
computed both intra-procedurally and inter-procedurally by exploiting information
2.4. Program Slicing

```c
void main(void) {
    int current;
    int result;
    int flag;
    scanf("%d", &flag);
    if (flag)
        result = 0;
    else
        result = 1;
    do{
        scanf("%d", &current);
        if (current > 0) {
            if (flag)
                result = result + current;
            else
                result = result * current;
        }
    } while (current > 0);
    printf("The result is %d\n", result);
}
```

Figure 2.6: A quasi-static backward slice example taken with respect to the variable `result` at the last statement on any input stream of the form `<0 . . . >`. 
readily available during debugging. The hybrid slicing approach improves on the precision of static slicing by computing more accurate and therefore smaller slices. The hybrid slice decomposes the overall slice into subslices by using the dynamic information as reference points, thus dividing the execution path into intervals.

Within the idea of hybrid slicing, Schoenig and Ducassé [113] proposed a hybrid backward slicing algorithm for Prolog which can produce executable slices. Their algorithm is the first one proposed for Prolog.

Other similar approaches that use both static and dynamic information have been proposed in [33, 40, 74, 99]. These approaches use static information to improve the execution time performance of dynamic slicing while maintaining the precision of dynamic slicing. Rilling presented a hybrid slicing framework and introduced two general hybrid slicing algorithms in [111].

This thesis also presents a form of hybrid slice, but it contains two static information source – concept assignment and static slicing.

2.4.1.5 Simultaneous Slicing

Danicic and Harman [35] introduced simultaneous slicing as a generalisation of program slicing in which a set of slicing criteria is considered instead of only one in conventional slicing. The \{< s_1, v_1 >, \ldots, < s_n, v_n >\} is referred to as a simultaneous slicing criterion, where \(< s_i, v_i >, i \leq n,\) are static slicing criteria. Obviously, the slice can be computed from the union of slices with respect to each static criterion.

Figure 2.7 shows a simultaneous slice for the same program in Figure 2.4 and the slice highlighted in bold taken with respect to the final value of the variable set \{i, sum\}.
2.4. Program Slicing

```c
void main(void){
    int current;
    int sum = 0;
    int i = 0;
    do{
        printf("Enter an integer: ");
        scanf("%d", &current);
        if (current > 0){
            sum = sum + current;
            i = i + 1;
        }
    }while (current > 0);
    printf("The sum of %d numbers is %d\n", i, sum);
}
```

Figure 2.7: A simultaneous slice example taken with respect to the variable set \{i, sum\} at the last statement.
Decomposition slicing [45] was introduced in the context of software maintenance to capture all computation on a given variable. The slicing criterion is \{ <s_1, v >, \ldots, <s_n, v > \}. It is clear that decomposition slicing is a variety of simultaneous slicing where the set of slicing criteria is derived from the same variable in the whole program.

Gold et al. [47] introduced techniques for unification of program slicing and concept assignment. The resulting Concept Slices computed by the algorithm are subprograms extracted using high level criteria. Executable Concept Slicing (ECS) and Forward Concept Slicing (FCS) are proposed by using backward slicing and forward slicing with respect to the results of concept assignment, the concept binding. The Concept Slices formed are another variety of simultaneous slicing where the high level concepts are taken as slicing criterion.

Executable Concept Slicing (ECS) [47] is formed using slicing to extend the results of concept assignment by slicing with respect to the components associated with a given concept. This computation exploits the advantages of both techniques, while overcoming their individual weaknesses. Executable concept slicing takes advantage of the executability of programs produced by slicing and the high-level extraction criterion from concept assignment. This contrasts with the unexecutable concept binding produced by concept assignment alone and the low-level criterion required for program slicing.

An algorithm for computing all executable concept slices is presented in Figure 2.8. For each concept \( c_i \), it computes the slice taken with respect to the set of SDG vertices that represent statements from \( c \). Slicing on these vertices produces the ECS.

Reconsider the example shown in Figure 2.1. An engineer interested in the average
2.4. Program Slicing

function ECS (Program $P$, DomainModel $D$) returns: set of Program

    let $\{c_1, \ldots, c_n\} = \text{HB-CA-Concepts}(P, D)$
    for each $c_i \in \{c_1, \ldots, c_n\}$
        let $ECS_i = \text{Slice}(P, SDG-\text{vertices}(c_i))$
    endfor
    return $\{ECS_1, \ldots, ECS_n\}$

Figure 2.8: The Executable Concept Slicing Algorithm

sound volume could apply HB-CA. The result is the last statement of the program (shown boxed line). Slicing on the SDG vertex (in general vertices) for the statements identified by HB-CA returns the vertices represented by the bold statements Figure 2.1. These statements make up the ECS for the average concept. They form an executable program that computes average volume.

An Forward Concept Slicing (FCS) is formed in the same manner to ECS except it uses a forward slicing to capture code that could be affected by a concept. As a forward slice is not guaranteed to be executable, the resulting forward concept slice may not be executable.

De Lucia et al. [37] reported that “unions of slices are not slices” if based on definition of slice rigidly. However, this result is just interesting from a theoretical point of view. The goal of most simultaneous slicing such as concept slicing is to union slices to achieve an approximation of the sub-program with high level meaning, rather than a minimal slice that preserves the semantics of the original program.
2.4. Program Slicing

2.4.1.6 Conditioned Slicing

Canfora et al. introduced the notion of conditioned slice [29] in 1998 as a general framework for statement deletion based slicing. A conditioned slice consists of a subset of program statements which preserves the behaviour of the original program with respect to a slicing criterion for a given set of executions. The set of initial states of the program that characterise these executions is specified in the form of a first order logic formula on the input variables. Therefore, a conditioned slice is constructed with respect to three pieces of information, a variable $v$ at statement $s$, on conditions $F$. The tuple $< s, v, F >$ is referred to as a conditioned slicing criterion.

Figure 2.9 shows a simple program that computes the student’s examination result (fail, pass) and the grade by inputing the student’s overall mark and the slice highlighted in bold taken with respect to the final value of the the variable grade on the condition $F_0 \equiv (mark \geq 35 \land mark < 40)$.

Conditioned slicing allows a better decomposition of the program giving human readers the possibility to analyse code fragments with respect to different perspectives. It was claimed that conditioned slicing subsumes any other form of statement deletion based slicing method [29], i.e., the conditioned slicing criterion can be specified to obtain any form of slice. However, a more recent paper [18] showed that the subsumes relationship does not always exist-conditioned slicing does not subsume Korel and Laski dynamic slicing.

Harman et al. [64] presented and formalised the pre/post conditioned slicing method, which combines forward and backward conditioning to provide a unified framework for conditioned program slicing. The pre/post conditioned slicing can be used to improve the analysis of programs in terms of pre- and post- conditions.
2.4. Program Slicing

```c
1 void main()
2 {
3     int mark;
4     char grade, *result;
5     scanf("%d", &mark);
6     if (mark < 35){
7         result = "fail"; grade = 'F';}
8     else{result = "pass";
9         if (mark < 40) grade = 'E';
10        else { if(mark<50) grade = 'D';
11            else { if(mark<60) grade = 'C';
12               else { if(mark<70) grade = 'B';
13                 else { grade = 'A';}}}}}
```

Figure 2.9: An example Conditioned slice taken with respect to the variable grade at the last statement on the condition $F_0 \equiv (\text{mark} \geq 35 \land \text{mark} < 40)$.

2.4.1.7 Amorphous Slicing

Harman et al. introduced *amorphous slicing*, which removes the limitation to component deletion as the only means of simplification [59, 58]. In amorphous slicing, the syntactic requirement is relaxed while the semantic requirement is retained. The slice preserves the selected behavior of interest from the original program.

Amorphous slicing can be thought of as the result of combining slicing and transformation. Both techniques tend to reduce program size while preserving the (projected) semantics of the program, but are unlike slicing, transformations do not preserve syntax. On the other hand, slicing preserves the semantics of the program for some projection of the original semantics.

Combining slicing and transformation rules, including statement deletion leads to
slices that are often considerably smaller than their syntax-preserving counterparts. Figure 2.10 shows an example of an amorphous slice and the possible further reduction in slice size when compared against syntax-preserving slices. The syntax-preserving slice are highlighted in bold taken with respect to the variable `slice` at the last statement. The amorphous slice is list in the right side.

<table>
<thead>
<tr>
<th>Syntax-Preserving slice</th>
<th>Amorphous slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D = 2*r; )</td>
<td>( \text{slice} = 2*\pi<em>r</em>r + h*\pi<em>2</em>r; )</td>
</tr>
<tr>
<td>( \text{FaceArea} = \pi<em>r</em>r; )</td>
<td></td>
</tr>
<tr>
<td>( C = \pi*D; )</td>
<td></td>
</tr>
<tr>
<td>( \text{TempArea} = \pi * \text{FaceArea}; )</td>
<td></td>
</tr>
<tr>
<td>( S\text{Area} = 2*\text{FaceArea}+h*C; )</td>
<td></td>
</tr>
<tr>
<td>( \text{slice} = S\text{Area}; )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.10: An example showing Amorphous Slicing producing thinner slices by removing syntactic restrictions. Slice computed w.r.t. variable `slice` at end.

2.4.1.8 Chopping

Jackson and Rollins [72] introduced a variation of slicing – program chopping. The criterion of program chopping is defined as a pair of \((s, t)\), where \(s\) is `source` and \(t\) is `sink`. Chopping answers questions of the form “What are all the program statements that serve to transmit effects from a given source \(s\) to a given target \(t\)?”

The essence of program chopping is to extract the statements involved in a transitive dependence from the source criterion to the sink criterion. Basically, a chop for a chopping criterion \((s, t)\) is the intersection of a backward slice for \(t\) with a forward slice for \(s\), that is \(\overleftarrow{S}(t) \cap \overrightarrow{S}(s)\).

Figure 2.11 shows a chop for the same program in Figure 2.4 and the chop highlighted
in bold taken with respect to the pair($current$, $sum$). The variable of the $current$ and $sum$ are showed boxed in the figure.

```c
void main(void){
int current;
int sum = 0;
int i = 0;
  do{
    printf("Enter an integer: ");
    scanf("%d", &current);
    if (current > 0){
      sum = sum + current;
      i = i + 1;
    }
  }while (current > 0);
  printf("The sum of %d numbers is %d\n", i, sum);
}
```

Figure 2.11: A chopping example taken with respect to the boxed variables pair {current, sum}.

Jackson and Rollins restricted $s$ and $t$ to be in the same procedure. Reps and Rosay [109] presented an unrestricted interprocedual form of program chopping. Krinke [84] presented an empirical evaluation of context-sensitive chopping and context-insensitive chopping. He proposed an approximative chopping method and found that this method is much faster and can produce results almost as precise as that of Reps and Rosay. Krinke [85] also presented an approach called ‘barrier chopping’. This approach introduces ‘barriers’ which are not allowed to be passed during chopping computation. Using barriers can further reduce chopping for program comprehension.
2.4. Program Slicing

2.4.2 Intraprocedural Program Slicing Algorithms

2.4.2.1 Original Slicing Approach–dataflow Equations

Weiser’s original slicing approach is a kind of approach based on iteration of dataflow equations. In this approach, slices are computed in an iterative process by computing consecutive sets of relevant variables for each node in the CFG.

The algorithm first computes directly relevant statements for each node in the CFG by tracing transitive data dependences, and then indirectly relevant statements are gradually added to the slice by tracing control dependences. The process stops when no more relevant statements are found.

In order to be a slice with respect to criterion \((n, V)\), where \(n\) is a node in the CFG of the program and \(V\) a subset of the program’s variables, four data-flow sets are used to compute the slice. The sets \(\text{Def}(i)\) and \(\text{Ref}(i)\) denote the sets of variables defined and referenced at CFG node \(i\), respectively. The set \(\text{Control}(i)\) contains those statement that directly determine whether the node \(i\) is executed. The set \(R_C(i)\) contains the relevant variables whose values (transitively) affect the computation of \(v\) at CFG node \(i\). From \(R_C(i)\), a set of statements \(S_C\) is derived that constitutes the desired program slice.

The slice \(S\) is calculated in four steps:

1. \(R_C^0(n) = V\); Initialise the relevant set \(R_C(n)\) to \(V\)
2. \(R_C^0(m) = \emptyset\); Initialise the other relevant sets of all nodes to the empty set
3. Work backwards in the CFG, for each edge \(i \rightarrow_{\text{CFG}} j\):
   \[
   R_C(i)^0 = R_C(i) \cup \{v \mid v \in R_C(j), v \notin \text{Def}(i)\} \cup \{v \mid v \in \text{Ref}(i), \text{Def}(i) \cap R_C(j) \neq \emptyset\}
   \]
2.4. Program Slicing

\[ S_C^0 = \{ i \mid (\text{Def}(i) \cap R_C(j)) \neq \emptyset, i \rightarrow_{\text{CFG}} j \} \]

4. For each \( B_C^k = \{ b \mid \exists i \in S_C^k(i), i \in \text{CONTROL}(b) \} \), repeating step 3

\[
R_C^{k+1}(i) = R_C^k(i) \cup \bigcup_{b \in B_C^k} R_{(b, \text{REF}(b))}(i)
\]

\[
S_C^{k+1} = B_C^k \cup \{ i \mid (\text{Def}(i) \cap R_C^{k+1}(j)) \neq \emptyset, i \rightarrow_{\text{CFG}} j \}
\]

As an example, consider slicing the program of Figure 2.2 with respect to criterion \((13, \{\text{sum}\})\). Table 2.1 summarises the DEF, REF, CONTROL sets, and the set of relevant variables computed by Weiser’s algorithm. The CFG of the program was shown in Figure 2.2 as well. From the information in the table, we obtain \( S_C^0 = \{2, 3, 7, 9, 12\}, B_C^0 = \{5, 8, 11\}, \) and \( S_C^1 = \{2, 3, 5, 7, 8, 9, 11, 12\}, \) which is identical to the slice highlighted in bold in Figure 2.4.

**Table 2.1:** Results of Weiser’s algorithm for the example program in Figure 2.2 with respect to slicing criterion \((13, \{\text{sum}\})\).

<table>
<thead>
<tr>
<th>Node #</th>
<th>DEF</th>
<th>REF</th>
<th>CONTROL</th>
<th>( R_C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>{current}</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>{sum}</td>
<td>( \emptyset )</td>
<td>{current}</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>4</td>
<td>{i}</td>
<td>( \emptyset )</td>
<td>{sum, current}</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>5</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>{sum, current}</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>6</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>{5, 11}</td>
<td>{sum, current}</td>
</tr>
<tr>
<td>7</td>
<td>{current}</td>
<td>{current}</td>
<td>{5, 11}</td>
<td>{sum, current}</td>
</tr>
<tr>
<td>8</td>
<td>( \emptyset )</td>
<td>{current}</td>
<td>{5, 11}</td>
<td>{sum, current}</td>
</tr>
<tr>
<td>9</td>
<td>{sum}</td>
<td>{sum, current}</td>
<td>{8}</td>
<td>{sum, current}</td>
</tr>
<tr>
<td>10</td>
<td>{i}</td>
<td>{sum, current}</td>
<td>{8}</td>
<td>{sum}</td>
</tr>
<tr>
<td>11</td>
<td>( \emptyset )</td>
<td>{current}</td>
<td>{sum}</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>12</td>
<td>( \emptyset )</td>
<td>{i, sum}</td>
<td>{sum}</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>13</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
2.4. Program Slicing

2.4.2.2 Information-flow Relations

Bergeretti and Carré [12] proposed another approach that defines slices in terms of information-flow relations derived from a program in a syntax-directed, bottom-up manner.

For a statement (or sequence of statements) $S$, a variable $v$, and an expression $e$ that occurs in $S$, three binary relations are defined in turn:

$$\lambda_{S}, \text{ from } v \text{ to } e, (v, e)$$

$$\mu_{S}, \text{ from } e \text{ to } v, (e, v)$$

$$\rho_{S}, \text{ from } v \text{ to } v', (v, v')$$

These information-flow relations possess the following properties: $(v, e) \in \lambda_{S}$ iff the value of $v$ on entry to $S$ potentially affects the value computed for $e$, $(e, v) \in \mu_{S}$ iff the value computed for $e$ potentially affects the value of $v$ on exit from $S$, and $(v, v') \in \rho_{S}$ iff the value of $v$ on entry to $S$ may affect the value of $v$ on exit from $S$. For instance, $v\lambda_{S}e$ can be (loosely) interpreted as “the value of $v$ on entry to $S$ may be used in the evaluation of the expression $e$ in $S$.”

The set $E_{v}^{S}$ of all expressions $e$ for which $S$ can be used to construct partial statements. A partial statement of statement $S$ associated with variable $v$ is obtained by replacing all statements in $S$ that do not contain expressions in $E_{v}^{S}$ by empty statements. This yields the slice with respect to the final value of $v$.

2.4.2.3 Graph Reachability

The most popular kind of slicing approach is slicing via graph reachability [108]. In these approaches, slicing can be divided into two steps. In the first step, dependence
graphs of the program are constructed, and then the slices can be produced by performing graph reachability analysis, i.e., traversing edges on the dependence graph from a node representing the slicing criteria. Thus, in dependence graph based approaches, the slicing criterion is identified with a vertex $v$ in the graph.

It is Karl Ottenstein and Linda Ottenstein [106] (now Linda Ott) that first defined slicing as a reachability problem in a dependence graph representation of a program. They used the program dependence graph (PDG) (see Section 2.3.3) to compute single-procedure program slices. They proposed a slicing algorithm based on graph reachability in PDG. For single-procedure programs with program dependence graph $G$, the slice with respect to $v$ consists of all vertices in $G$ that can reach $v$. Figure 2.12 gives an algorithm for computing the vertices of a slice using a program dependence graph.

```
function MarkVerticesOfSlice (PDG G, Vertex v)
returns: set of Vertices

let WorkList W = {v}
mark(v) = true

while WorkList ≠ ∅
    let w = RemoveElement (W)
    foreach edge v → w ∈ E(G)
        if not mark(v)
            W = W ∪ {v}
            mark(v) = true

return the set of all marked vertices
```

Figure 2.12: The intraprocedural program slicing algorithm

The algorithm returns the set of vertices of slice, the related parts of the source text of the program can be found by maintaining a mapping between vertices of the
2.4. Program Slicing

PDG and the source text during the construction of the PDG.

In Figure 2.3, the example procedure and corresponding PDG were shown. All vertices will return in the slice with respect to return sum.

2.4.3 Interprocedural Program Slicing Algorithms

The main problem posed by interprocedural static slicing is that, in order to compute accurate slices, the call-return structure of interprocedural execution paths must be taken into account. Simple algorithms that include statements in a slice by traversing (some representation of) the program in a single pass have the property that they consider infeasible execution paths, causing slices to become larger than necessary.

2.4.3.1 Dataflow Equations

Slicing interprocedural programs is complicated due to the necessity of translating and passing slicing criteria into and out of procedures. Weiser formalises the generation of new criteria by way of functions $\text{Up}(C)$ and $\text{Down}(C)$ that map a set $C$ of slicing criteria in a procedure $P$ to a set of criteria in procedures that call $P$, and a set of criteria in procedures called by $P$, respectively. The set of all criteria with respect to which intraprocedural slices are computed consists of the transitive and reflexive closure of the $\text{Up}$ and $\text{Down}$ relations; this is denoted $(\text{Up} \cup \text{Down})^*$. Thus, for an initial criterion $C$, slices will be computed for all criteria in the set $(\text{Up} \cup \text{Down})^*(\{C\})$.

Horwitz et al. [70] report that Weiser’s algorithm for interprocedural slicing introduces an imprecision, because of what they refer to as the “calling context” problem.
2.4. Program Slicing

In a nutshell, the problem is that when the computation “descends” into a procedure $Q$ that is called from a procedure $P$, it will “ascend” to all procedures that call $Q$, not only $P$. This includes infeasible execution paths that enter $Q$ from $P$ and exit $Q$ to a different procedure. Traversal of such paths gives rise to inaccurate slices and dramatically increase slice size.

2.4.3.2 Information-flow Relations

Bergeretti and Carré [12] explain how the effect of procedure calls can be approximated in the absence of recursion. Exact dependences between input and output parameters are determined by slicing the called procedure with respect to each output parameter. Then, each procedure call is replaced by a set of assignments, where each output parameter is assigned a fictitious expression that contains the input parameters it depends upon. A call to a side-effect free function can be modeled by replacing it with a fictitious expression containing all actual parameters. Note that the computed slices are not truly interprocedural since slices are not extended to procedures other than the one containing the slicing criterion.

2.4.3.3 Graph Reachability

Horwitz, Reps and Binkley [70] proposed an interprocedural slicing algorithm that operates on the System Dependence Graph (SDG) (see Section 2.3.4). The Horwitz-Reps-Binkley (HRB) algorithm involves three steps: first, construct the SDG, which represents the program; second, compute the interprocedural summary edges, which represent transitive dependences due to procedure calls; third, slices are computed by doing a two phase traversing on the SDG. Assume that slicing starts at some vertex $v$. The first phase determines all vertices from which $v$ can be reached without
descending into procedure calls. The second phase determines the remaining vertices in the slice by descending into all previously side-stepped calls.

The slicing algorithm is efficient and can handle the calling context problem due to the summary edges of an SDG. Reps et al. [107, 108] proposed a new algorithm for computing the summary edges of an SDG, which is asymptotically more efficient than HRB algorithm [70]. Forgács and Gyimóthy [43] introduced the idea of divide and conquer to the computation of summary edges. Dividing the call graph and analysing procedures in each group in topological order can significantly improve the efficiency of interprocedural slicing method.

Several extensions of the HRB algorithm were presented. Lakhotia [87] adapted the idea of lattice theory to interprocedural slicing and presented a slicing algorithm based on the augmented SDG in which a tag is contained for each vertex of SDG. Different from the HRB algorithm, this algorithm only needs a single traversal on the SDG. Binkley [22] extended the HRB algorithm to produce executable interprocedural program slices. He also presented an interprocedural slicing algorithm in the presence of parameter aliases [17]. The algorithm is parameterised by a set of aliasing patterns for each procedure. Flow dependence edges are created on different “levels” to separately represent different possible flow dependences for different possible aliasing patterns. Slices can then be obtained by traversing the multiple-level flow dependence edges in the SDG.

2.4.4 Applications of Program Slicing

2.4.4.1 Software Quality Assurance

Testing and debugging are important activities in software quality assurance.
Debugging can be a difficult task when one is confronted with a large program. The original motivation for program slicing was to aid the location of faults during debugging activities. The idea was that with respect to the slice criterion of variable with an erroneous value, only the statements in the slice have (possibly) contributed to the computation of that value. In this case, it is likely that the error occurs in the one of the statements in the slice and localising a bug in the slice potentially allows one to ignore many statements [94]. However, slicing cannot be used to identify bugs such as ‘missing initialisation of variable’. If the original program does not contain a line of code then the slice will not contain it either.

In debugging, one is often interested in a specific execution of a program that exhibits anomalous behavior. Dynamic slicing is one variation of program slicing introduced to assist in debugging [81]. Dynamic slices are particularly useful here [2, 82], because they only reflect the actual dependences of that execution, resulting in smaller slices than static ones.

Agrawal’s thesis [1] contains a detailed discussion how static and dynamic slicing can be utilised for semi-automated debugging of programs. He proposed an approach where the user gradually “zooms out” from the location where the bug manifested itself by repeatedly considering larger data and control slices.

Testing is an important part of software engineering as it consumes at least half of the labour expended to produce a working program [11]. A program satisfies a “conventional” data flow testing criterion if all def-use pairs occur in a successful test-case. Duesterwald, Gupta, and Soffa [41] propose a more rigorous testing criterion, based on program slicing: each def-use pair must be exercised in a successful test-case; moreover it must be output-influencing. A def-use pair is output-influencing if it occurs in an output slice. It is up to the user, or an automatic test case generator to construct enough test cases such that all def-use pairs are tested. Three slicing
2.4. Program Slicing

approaches are utilised, based on different dependence graphs. Static slices are computed using static dependence graphs (similar to the PDGs of Horwitz, Reps, and Binkley [70], dynamic slices are computed using dynamic dependence graphs (similar to DDGs of Agrawal and Horgan [3], but instances of the same vertex are merged, resulting in a slight loss of precision), and hybrid slices [55, 56] are computed using dependence graphs that are based on a combination of static and dynamic information.

Regression testing consists of re-testing only the parts affected by a modification of a previously tested program, while maintaining the “coverage” of the original test suite. Applying program slicing to the problem of reducing the cost of regression testing has been studied for a long time. Gupta, Harrold, and Soffa [54] describe an approach to regression testing where slicing techniques are used. Backward and forward static slices serve to determine the program parts affected by the change, and only test cases that execute “affected” def-use pairs need to be executed again. Binkley [23] describes two complementary algorithms for reducing the cost of regression testing of multiprocedure programs.

Program slicing is also used in mutation testing. Amorphous static slicing can be used in detection of equivalent mutants in mutation testing [68]; conditioned slicing can complement partition-base testing [67]; slicing and related dependence analyses can be use to support mutation testing [63].

2.4.4.2 Software Maintenance and Re-engineering

Program slicing can be used in software maintenance. One of the problems in software maintenance consists of determining whether a change at some place in a program will affect the behavior of other parts of the program. Gallagher and Lyle [45]
introduced the concept of decomposition slicing and discussed its application to software maintenance. Static slicing was adapted for the decomposition of a program into a set of components (i.e., reduced programs), each of which captures part of the original program’s behavior. Zhao [126] introduced architectural slicing to aid architectural understanding and reuse. Canfora et al. [31] presented a method to identify reusable functional abstraction in existing code. Lanubile and Visaggio [90] proposed a method to extract reusable functions from ill-structured programs using transform slicing which is a variant of the conventional program slice.

Besides above application in software maintenance, program slicing can also be used in reverse engineering. Beck and Eichmann [10] applied program slicing techniques to reverse engineering by using it to assist in the comprehension of large software systems, through traditional slicing techniques at the statement level, and through a new technique, interface slicing, at the module level. Jackson and Rollins [72] proposed an improved program dependence graph (PDG) that satisfies both, while retaining the advantages of the PDG.

2.4.4.3 Measurement

Cohesion and coupling are two important metrics in software measurement.

Longworth [92] first studied the use of program slicing as indicator of cohesion. Ott and Thuss [102] then noted the visual relationship that existed between the slices of a module and its cohesion as depicted in a slice profile. Bieman and Ott [14] examined the functional cohesion of procedures using a data slice abstraction. Harman et al. [60] improved the cohesion metrics introduced by Ott and her colleagues by replacing the use of ‘Line of Code’ metric with their expression metrics proposed to evaluate the complexity of an expression. This work can give a more accurate measure of
cohesion. Later, Ott and Bieman [101] found a way of using syntax preserving static program slicing to measure program cohesion.

Coupling is the measure of how one module depends upon or affects the behavior of another. Harman et al. [65] proposed a method of using program slicing to measure coupling. It is claimed that this method produces more precise measurement than information flow based metrics.

### 2.5 Key Statement Analysis

Key Statement Analysis (KSA) is a refinement of concept assignment. In a concept binging, some statements contribute more to the computation embodied by the binding than others. These statements are termed key statements [47]. Key statements can be used to increase the richness of information provided to a software engineer. It may help in the extraction of a tighter subset of code for reuse, to focus attention more rapidly on the part of a computation most likely to cause a problem, or to identify poor cohesion in the source code being studied.

The algorithms for KSA are based on the notion of principal variables, which are those which might be considered to be the result of a set of statements [14, 100]. Gold et al. [47] defined three kinds of principal variables of a concept binding, i.e., global and assigned variables, call-by-reference and assigned variables, or the parameter to an output statement of a concept binding.

Figure 2.13 shows the flowchart of KSA. Given the source of program with library (knowledge base of concepts), concept assignment can extract concept bindings (CB) with the assigned concepts from the source. Principal Variables (PV) can be identified for each CB via program representation and program dependence analysis, then
Figure 2.13: The original flowchart for Key Statement Analysis

Key statements (KS) can be computed via specific algorithms.

Two algorithms to compute key statements, $\text{KSA}_{BO}$ and $\text{KSA}_{BE}$, are presented [47].

![Function KSA BO](image)

function $\text{KSA}_{BO}$ (Program $P$, Concept Binding $c$) returns: function from Statement to $\{0, 1\}$

for each variable $v_i$ in $\text{PV}(c)$

    let $s_i = \text{Slice}(P, \{\text{Finaluse(Statement}(c), v_i)\})$

endfor

let Tight = $\bigcap_i s_i$

let KS = Statement($c$) $\cap$ Tight

return $\lambda x$. If $x \in KS$ then 1 else 0

Figure 2.14: Key Statement Analysis ‘Bieman-Ott’ Style[47]

Figure 2.14 present the first KSA algorithm $\text{KSA}_{BO}$ which inspired by Bieman and Ott’s work on measuring cohesion using slicing [14] and Tightness metric introduced by Ott and Thuss [102]. The function $\text{KSA}_{BO}$ takes a program and a concept binding within it, and returns a function that describes the relative weight of each statement in the concept binding. This algorithm measures the intersection of all slices on principal variables in binding. The statement within the intersection will be assign a value 1, otherwise 0. In this case, the result is either 0 or 1, with 1 signifying that a statement is a key statement and 0 signifying that it is not.
2.5. Key Statement Analysis

function KSA_{BE} (Program \( P \), Concept Binding \( c \))
returns: function from Statement to \( R \)

let \( F = {} \)
let \( N \) be the longest acyclic path in the SDG of \( p \)
for each \( s_i \) in \( Statements(c) \)
    for each variable \( v_i \) in \( PV(c) \)
        let \( d_{ij} = \text{Dist}(p, s_i, \{\text{Finaluse(Statement(c), } v_i\})\)\)
    endfor
    let \( D_i = \min_j d_{ij} \)
    let \( F = F \cup \{s_i \mapsto \frac{N-D_i}{N}\} \)
endfor
return \( F \)

Figure 2.15: Key Statement Analysis ‘Ball-Eick’ Style[47]
2.6 Dependence Clusters

Figure 2.14 present the second KSA algorithm \( KSA_{BE} \) which inspired by Ball and Eick’s work on the visualisation of program slices project [9]. This algorithm \( KSA_{BE} \) computes the weight for each statement \( s \) by the length of the shortest path from \( s \) to a final use of principal variable, normalised with respect to the length of the longest acyclic path in the program’s SDG. These weightings will be real numbers in the range 0 to 1. Statements with KSA values closer to 1 are more ‘key’ than those with KSA values closer to 0.

2.6 Dependence Clusters

Binkley and Harman [21] introduced the concept of dependence cluster. A dependence cluster is a set of program statements that mutually depend upon one another. Any change to the computation represented at one point in a dependence cluster potentially affects the computations represented by all other points in the cluster. Binkley and Harman [21] formally defined the dependence cluster as follows:

A dependence cluster is a set of Control Flow Graph (CFG) nodes, \( N_1, \ldots, N_m \) (\( m > 1 \)), such that for all \( i, 1 \leq i \leq m \) and for all \( j, 1 \leq j \leq m \) \( N_i \) depends on \( N_j \). [21]

2.6.1 Dependence Cluster Causes

Sources of large dependence clusters include:

- Loops – in particular those with large bodies. Here the variables and function calls that (transitively) depend upon loop counter will end up in a dependence cluster. For a loop with a large body, this can yield a rather large cluster.
Global variables and function pointers [25, 98] – The statements using or assigning one global variable will produce the data dependence between them. In C programs, function pointers is widely used to simulate call-by-reference semantics used in object-oriented programs, to avoid the expensive copying of large objects, to implement list, tree, or other complex data structures. Any uses or assignments of the objects pointed by function pointer will results in data dependence on the original objects. Thus, if a large amount of uses of global variables or function pointers will results in a large dependence cluster.

Mutually Recursive Clusters (MRCs) [21] – mutual recursion naturally leads to large dependence clusters because each function (transitively) calls all the others, making the outcome of each function dependent upon the outcome of some call to each of the others.

Capillary Data Flows (CDFs) [21] – 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.

2.6.2 Dependence Clusters Identification

It is possible to identify dependence clusters by finding SDG vertices by slicing; the sources of identical slices form a dependence cluster. However, an approximation is used for ‘same slice’ which is not only more efficient, but also leads to a useful visualisation for identifying clusters: the Monotone Slice-size Graph (MSG)[21]. Rather than testing if two SDG vertices have identical slices, the approach simply compares the slice size for the two vertices.
2.6. Dependence Clusters

The conjecture that underpins this approximation is that two slices that are the same size are likely to be the same slice. Although, two slices may differ yet, coincidentally, may have identical sizes, two slices which are identical must clearly have the same size. An experiment [21] involving twenty programs provided strong evidence for the claim that ‘size agreement’ is a good approximation for ‘slice content agreement’ and thus for locating dependence clusters.

2.6.2.1 Monotone Slice-size Graphs

Monotone Slice-Size Graphs (MSG) have been developed to aid in the visual identification of dependence clusters [21]. Such a graph is defined formally as follows:

A Monotone Slice-size Graph (MSG) is a graph of slice size, plotted for monotonically increasing size. That is, slices are sorted according to increasing slice size and the sizes are plotted on the vertical axis against slice number in order on the horizontal axis. [21]

Dependence clusters can be detected by seeking parts of the MSG where a large number of slices have the same size. This manifests itself as a ‘plateau’ in the graph as shown in Figure 2.16.

2.6.2.2 Characterising Dependence Cluster Size

Many programs will have small clusters of slices forming dependence clusters. Of more interest are the larger clusters formed where programs have high levels of intra-dependence. In determining whether or not a program has a large dependence cluster or not, there is a value judgment to be made concerning its size. For the purposes of this analysis, the threshold above which a cluster is considered to be
2.6. Dependence Clusters

Figure 2.16: Sample of MSG

‘large’ is set to 10% (i.e., the slices in the cluster represented 10% or more of the program in terms of program points).

The MSG helps with the essentially subjective task of determining whether a cluster is large, relative to the code which contains it, since the length of plateau indicates a large cluster of slices of identical size. Actually, in the visualisation, small differences may appear as part of the same plateau. For example, slices including 38% and 38.1%, are not visually different. Thus, the calculation of the largest cluster includes the largest set of program points in which all program points have the exactly same slice size. As result, the size of the largest clusters will be a bit smaller than the approximation from MSG.
Chapter 3

Experimental Design

This chapter describes the subject programs used in the experiments and the tools employed to implement the experiments. Finally, measurements used in the experiment are described.

3.1 Motivation

This thesis proposed three combination approaches of concept assignment and slicing based dependence analysis. To ensure that three approaches can be applied to real programs, experimental methods must be adopted for the evaluation of three approaches.

Each approach goes through three phases: analysis, implementation, and experimentation. We first analyse the challenge of the approach and possible factors that can affect the approach, then research questions are proposed based on the analysis. Second, a framework and algorithm are designed for implementation and metrics are proposed for evaluations. Finally, experiments are run using selected program
3.2. Subject Programs

subjects. The experimental data are analysed using statistical analysis methods and the results are used to answer the proposed questions. The details are given in each substantive chapter.

### 3.2 Subject Programs

Table 3.1: Experimental Subjects used in the thesis.

<table>
<thead>
<tr>
<th>Subjects</th>
<th>LoC</th>
<th>SLoC</th>
<th>Vertices</th>
<th>Large Cluster</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct-6.3.2</td>
<td>3,890</td>
<td>2,384</td>
<td>9,827</td>
<td>No</td>
<td>Process monitoring tools</td>
</tr>
<tr>
<td>EPWIC-1</td>
<td>7,943</td>
<td>4,232</td>
<td>19,545</td>
<td>No</td>
<td>Image compressor</td>
</tr>
<tr>
<td>space</td>
<td>9,126</td>
<td>6,576</td>
<td>34,684</td>
<td>No</td>
<td>ESA space program</td>
</tr>
<tr>
<td>oracolo2</td>
<td>9,477</td>
<td>5,842</td>
<td>24,754</td>
<td>No</td>
<td>Antennae array set-up</td>
</tr>
<tr>
<td>CADP</td>
<td>12,930</td>
<td>10,246</td>
<td>52,591</td>
<td>No</td>
<td>Protocol toolbox</td>
</tr>
<tr>
<td>usersv-1.0.1</td>
<td>7,301</td>
<td>5,320</td>
<td>95,056</td>
<td>Yes</td>
<td>Access control utility</td>
</tr>
<tr>
<td>indent-2.2.6</td>
<td>8,259</td>
<td>5,424</td>
<td>20,776</td>
<td>Yes</td>
<td>Text formatter</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>10,449</td>
<td>7,342</td>
<td>32,381</td>
<td>Yes</td>
<td>Calculator</td>
</tr>
<tr>
<td>diffutils-2.8</td>
<td>10,743</td>
<td>6,767</td>
<td>26,997</td>
<td>Yes</td>
<td>File comparison utilities</td>
</tr>
<tr>
<td>findutils-4.2.25</td>
<td>28,887</td>
<td>19,216</td>
<td>96,073</td>
<td>Yes</td>
<td>File finding utilities</td>
</tr>
<tr>
<td>Total</td>
<td>109,005</td>
<td>73,349</td>
<td>421,684</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The ten programs used in the studies are summarised in Table 3.1, which lists each program with some statistics related to the program and its System Dependence Graph (SDG). These include the size of the program in lines of code (LoC) and source lines of code (SLoC), and the size of the resulting SDG in vertices. The fifth column reports whether the program includes large dependence clusters or not and the sixth column provides the brief description for each program.
Although the LoC is known as a rather crude metric, the data is included to provide a consistent measure of program size that facilitates comparison with prior studies. To provide a better estimate of program size, the SLoC is reported as the number of non-blank non-comment lines of code.

The subject programs cover a wide range of programming styles. The subjects space and oracolo2 were developed for the European Space Agency. These two programs allow the user to describe the configuration of an array of antennas using a high-level language, the Array Definition Language (ADL), instead of writing the complete list of element positions and excitations. space functions as an interpreter for ADL. The program reads a file that contains ADL statements and analyses the contents of the file for adherence to the ADL grammar and to specify consistency rules. If the ADL file is correct, the space program outputs an array data file containing a list of array elements, positions, and excitations; otherwise, the program outputs error messages.

CADP (Construction and Analysis of Distributed Processes) is a toolbox for the design of communication protocols and distributed systems. CADP was developed by the VASY team at INRIA Rhone-Alpes.

EPWIC-1 (Embedded Predictive Wavelet Image Coder) is an embedded wavelet image coder that uses the generalised Laplacian (or generalised Gaussian) distribution as a model of the coefficients of the wavelet pyramid.

The other six programs are GNU free software. Since there are multiple versions of the software available at GNU website, the programs are reported with the version used in the experiments in Table 3.1.

In order to analyse the impact caused by large dependence cluster, the programs selected include five subject programs known (from prior study) to be free of large dependence clusters (acct-6.3.2, EPWIC-1, space, oracolo2, and CADP) and five sub-
ject programs known to contain large dependence clusters (\texttt{user-1.0.1}, \texttt{indent-2.2.6}, \texttt{bc-1.06}, \texttt{diffutils-2.8}, and \texttt{findutils-4.2.25}).

### 3.3 Analysis Tools

WeSCA, a tool based on HB-CA, was used to identify concept bindings in the source programs [47]. WeSCA requires a library of concepts to drive its analysis. A generic C concept library, used in other studies [47], was used as the basis for the analysis. It was extended for each program to reflect the different purposes (and thus concepts) of each program. To extend the generic C concept library, source files from each system were examined to identify possible concepts.

CodeSurfer [53] is a deep-structure analysis tool developed by Grammatech. A scripting language is integrated which extends standard Scheme with additional data types for representing the dependence-graph representation of C programs. These extended data types, and the operations on them, are described as Application Programming Interface (API). CodeSurfer’s API provides an interface to the vertices and edges of the program’s dependence graph, the program’s variables and their types, the program’s abstract syntax tree, and the correspondence between these abstract elements and the source text.

In the studies, CodeSurfer was used to build the SDGs for each program in Table 3.1 and to implement various dependence analysis and compute program slices in the experiments reported herein. All the algorithms for combining the concept assignment and dependence analysis, such as key statement computation and Vertex Rank Model, are programmed by CodeSurfer’s scripting language and integrated within the CodeSurfer implementation environment. Using the same language to represent source code and implement algorithms on it, takes a step towards achieving the overall goal of a fully automated and integrated tool.
3.4 Statistical Analysis

Statistical analysis is used to analyse the experimental data in the study. These include correlation analysis and mean (or meant) comparison analysis.

Correlation analysis generally computes linear correlation Pearson’s $R$ coefficient or non-linear correlation Spearman rank correlation.

Hypothesis testing is used to test the significance of the difference. Because there can be no certainty that the data distribution is normal, a Wilcoxon Matched-Pairs Signed Ranks Test procedure is applied to test the significance of the difference. In this case, the null hypothesis is that there is no difference between two test data and significance level is set to at 0.05 level.

3.5 Evaluation

Slice based dependence analysis involves slice computation and measurement. As mentioned above, LoC is a somewhat crude metric for measuring slice size owing to the lack of a standard definition for LoC. Following Binkley et al. [25], size is more precisely measured using SDG vertices. In this experiment, both the size of the low-level slicing criterion and the size of the resulting slices are reported as a percentage of the source-code-representing SDG vertices. Excluded from this count are internal vertices that do not directly represent the source. The most common of these are pseudo parameter vertices added for globals, which are treated as additional parameters.

Metrics are presented in each substantive chapter to evaluate the approach. The metrics are originated from slice-based metrics proposed by Mark Weiser [122]. Modifications are introduced to improve the metrics for each specific evaluation.
3.6 Threats to Validity

In the absence of human subjects, there are only two potential threats to the validity of this investigation: the threats to external and internal validity. External validity, sometimes referred to as selection validity, is the degree to which the findings can be generalised. Here, it is possible that the selected programs are not representative of programs in general and thus the results do not apply to ‘typical’ programs. To mitigate this concern, a range of programs from varying domains were studied. The descriptions in Table 3.1 give some indication of the application domains considered. However, as most of the programs are open-source programs, it remains possible that non-open-source programs (i.e., programs built in any radically different culture) will exhibit different behaviour. Finally, programs written using different programming paradigms (e.g., functional or object-oriented) may also produce different results.

The samples are sets of concept bindings drawn from these programs available to the experiment. This denotes a potential biased sample. Sadly, as with many software engineering experiments, it is not possible to sample from the space of all possible concept bindings in an unbiased manner. To do so would require access to all programs. Therefore, hypothesis testing is said to be used in an ‘informal’ manner in this thesis. Owing to the inherent sampling issues, the results may not be generalised.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. In this experiment, the primary threat comes from the potential faults in the tools used to gather the data.

In practice, the slicer might contain errors, or employ imprecise analysis (e.g., imprecise data-flow analysis or imprecise points-to analysis). To mitigate this concern,
a mature slicing tool (CodeSurfer [53]) was used. This tool has been thoroughly tested over a period of over ten years. The computation of the metrics was also rigorously tested to ensure that it accurately reflected the definition.

Regarding construct validity, it is possible that the metrics do not work properly. To help alleviate this concern, including using mature well-studied metrics, the new metrics were illustrated better by comparison to mature well-studied metrics. The tool and metric choices reduce the impact of construct validity threats on the conclusions reached, regarding the relationship between domain-level concepts and dependence in source code.
Chapter 4

Concept Extension Using Program Slicing *

4.1 Introduction

Traditional concept assignment allows statements of interest (those that denote a concept of interest) to be drawn out from the code. Much previous work on this has focused on techniques for identification of code segments and/or statements that are likely to denote domain level concepts. However, this work does not necessarily claim to extract code segments and statements that can be compiled and executed as stand-alone code units in their own right. Prior work showed how it was desirable to extract executable concept-denoting code [62]. This technique unifies concept assignment and program slicing to produce high-level executable concept slice (ECS). An executable concept slice is constructed by the union of the slices with respect to each of statements in concept binding.

*This chapter is based on the work published by the author at WCRE 2006 [48] and JSS[49]
The ability to execute the extracted concept code opens up a new set of possibilities for the programmer. It allows the program to be divided into smaller units, each of which can be experimented with by execution, allowing concept behaviour to be explored dynamically.

The applicability of any code extraction technique for program comprehension crucially depends upon the size of the subprogram extracted. From this point of view, one might be forgiven for supposing that executable concept slicing is simply impractical, because the union of several slices will tend to extract large amounts of code; perhaps the entire program in many cases. This is the central question that this chapter addresses.

This chapter presents the challenge of executable concept slice focusing on the size problem in Section 4.2. The experimental study is described in Section 4.3, including the experiments design and metrics for executable concept slice. Section 4.4 presents and discusses the result of the study. Section 4.5 summarises and draws the conclusion.

### 4.2 Challenge

The executable concept slice is a variety of simultaneous slices where the high level concepts are taken as slicing criterion. An executable concept slice, for domain level criterion, $C$, is constructed by forming the distributed union of the slices upon each of the statements identified by the concept assignment binding for $C$. Thus, it is possible that the use of slicing to ‘complete’ a set of concept–denoting statements in this way could potentially lead to very large executable concept slice. For example, recent work on slice sizes has indicated that the typical size of a backward slice is about one third of the program [13, 24]. Naïvely then, one might initially assume
that should a concept draw out more than three statements from a program, then the effort to make the statements executable would combine at least three slices, each of which would cater for a third of the program; surely making the concepts executable forces us to draw in the whole program? This is the central question that this chapter addresses.

Slice based metrics have been proposed and applied to measure program module cohesion [123, 103]. However, as extracted subprogram, there are no evaluations for executable concept slice. Identifying proper metrics is helpful for the evaluation. This is a subsidiary problem that this chapter addresses.

4.3 Experimental Study

4.3.1 Motivation

For ECS to be considered of practical use, it is important to establish that the slices produced by the technique are not so large as to confer no advantage over understanding the whole of the original program being analysed. Comparison of the results of ECS with “typical” slicing would be an appropriate method to achieve this. However, since there is no definition of “typical” slicing criteria against which the ECS criteria (and the subsequent result of slicing on both) can be compared, four types of criteria are defined to explore key aspects of the difference between traditional criteria and ECS criteria. Larger criteria should lead to larger slices unless some other factor (e.g., conceptual coherence) affects the slice’s size.

The investigation therefore defines four criteria types from the ostensibly most coherent (ECS) down to the least coherent (random) and compares the resulting slice sizes. More formally, the investigation attempts to verify the hypothesis that a
slice criterion formed from a concept binding will produce smaller slices than an equivalently-sized slice criterion formed without any consideration for conceptual information. This leads to three research questions:

1. Does slicing on a set of conceptually related SDG vertices \(i.e.,\) those representing a single HB-CA concept) produce a smaller slice than slicing on other similarly sized sets?

2. Is the size of an ECS related to the size of the HB-CA extracted concept binding?

3. Do large dependence clusters \[21\] affect ECS size?

### 4.3.2 Case Study Organisation

#### 4.3.2.1 Criteria Definitions

The study compares the size of slices constructed using four different generators of low-level slicing criteria. Each generator produces a set of SDG vertices. For example, the HB-CA generator returns the vertices representing all the statements identified as belonging to a particular concept binding.

The four criteria types used in the investigation are defined below. Informally, the first of the four, hereafter referred to as \(CA\) \((Concept-Assigned)\), uses the concept binding produced by a concept assigner (in the study the WeSCA implementation of HB-CA \[52\]). This approach is expected to generate the most coherent sets of criteria and thus the smallest slices. The remaining three types generate similar sized, but less coherent sets of criteria. \(CN\) \((Contiguous-Node)\) and \(CL\) \((Contiguous-Line)\) select SDG vertices that represent contiguous program statements starting
from a random statement; thus, it is unlikely that these statements correspond to
a single concept. A CN criterion includes a prescribed number of vertices. In the
investigation, this is always the same number as found in the corresponding CA
criterion. This type of criterion reflects most closely the internal structural form
of a CA criterion. A CL criterion includes a prescribed number of lines of code.
In the experiment, this is always constructed to be the same number of lines as
found in the corresponding CA criterion in order to facilitate a fair comparison.
This type of criterion reflects the lexical structure of a CA criterion. Finally, RN
(Random-Node), the least coherent type, includes a random collection of vertices.
In the experiment, the size of this set is constructed to be the same as that of the
corresponding CA criterion for comparison.

The four types are now formalised. In the definitions, lines of code include state-
ments, comments, and blank lines, as these are all relevant to concept assignment.

**Concept Assigned (CA)**

For a concept binding $C$ from program $P$, the Concept-Assigned criterion for $C$
includes the vertices of $P$’s SDG that represent the statements of $C$.

**Contiguous-Node (CN): Contiguous, Same Vertex Count, Random Start**

For a program $P$, given a natural number $n$ and a starting statement $s_i$, a Contiguous-
Node criterion includes the first $n$ vertices of $P$’s SDG that represent the statements
$s_i, s_{i+1}, \cdots$. 
4.3. Experimental Study

Contiguous-Line (CL): Contiguous, Same LoC Count, Random Start

For a program $P$, given a natural number $n$ and a starting statement $s_i$, a Contiguous-Line criterion includes the vertices that represent the $n$ statements $s_i, s_{i+1}, \ldots, s_{i+n-1}$.

Random-Node (RN): Non-Contiguous, Same Vertex Count, Random Points

For a program $P$ and a given natural number $n$, a Random-Node criterion is a collection of $n$ randomly selected vertices from $P$’s SDG.

![Figure 4.1: Example of the four types of criteria.](image)

Figure 4.1 illustrates the four types of criteria. In the figure a statement is represented by a light grey line and an SDG vertex by a dot. *Selected* components are shown in darker grey. Line 12 separates two concepts. Thus, assuming the concept from Lines 2-11 is of interest then the 17 vertices that represent these 10 lines would make up the CA criterion. In Figure 4.1, the example CN criterion is generated by selecting a random starting location (Line 6 in the example) and including contiguous statements until 17 vertices are included (when multiple vertices represent a statement, the starting and ending positions within the line are used to uniquely
order the vertices for selection). The example CL criterion is generated by selecting a random starting location (line 4 in the example) and including contiguous statements until 10 lines are included. Finally, the example RN criterion includes 17 randomly selected vertices.

### 4.3.2.2 Metrics for ECS

Coverage and Overlap are two important slice based-metrics proposed to measure program module cohesion by Mark Weiser [122] and formalised by Ott and Thuss [103]. The original, module based version of Coverage, denoted $Coverage_M$, compares the length of slices to the length of the entire module, similarly $Overlap_M$ is the average percentage of common statements within a module.

To study ECSs, coverage is defined with the module being the entire program. Let $P$ be the entire program, $V_C$ be the set of criteria defined in Section 4.3.2.1, $ECS_i$ corresponds to the slices computed for $C_i \in V_C$. The $Coverage_{ECS}$ is

$$Coverage_{ECS}(P) = \frac{1}{|V_C|} \sum_{i=1}^{|V_C|} \frac{|ECS_i|}{length(P)} = \frac{1}{length(P)} \frac{\sum_{i=1}^{|V_C|} |ECS_i|}{|V_C|}$$

which is in direct proportion to the average slice size.

The $Overlap_M$ is based on the number of statements common to all slices. If the intersection of all slices is null, then overlap is zero for ever. This neglects the fact of overlapping between partial slices. Here, the $Overlap_{ECS}$ is redefined. Let $ECS_{union}$ be the union of all the $ECS_i$, that is

$$ECS_{union} = \bigcup_{i=1}^{|V_C|} ECS_i$$

(4.2)
4.3. Experimental Study

The overlap is the average ratio of the size of each ECS to the number of statements in the union of all of the slices, that is

\[
Overlap_{ECS}(P) = \frac{1}{|V_C|} \sum_{i=1}^{|V_C|} \frac{|ECS_i|}{|ECS_{union}|}
\] (4.3)

The value of \(Overlap_{ECS}\) ranges between \(1/|V_C|\) and 1, where \(1/|V_C|\) occurs when there is no overlap and 1 when there is 100% overlap.

Table 4.1: Example Metrics Computations.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(SL_1)</td>
<td>(SL_2)</td>
<td>(SL_3)</td>
</tr>
<tr>
<td>1</td>
<td>1  0  0  0</td>
<td>1  1 0  0</td>
<td>1  1 1  1</td>
</tr>
<tr>
<td>2</td>
<td>1  0  0  0</td>
<td>1  1 0  0</td>
<td>1  1 1  1</td>
</tr>
<tr>
<td>3</td>
<td>0  1  0  0</td>
<td>0  0 0  0</td>
<td>0  0 0  0</td>
</tr>
<tr>
<td>4</td>
<td>0  1  0  0</td>
<td>0  0 0  0</td>
<td>0  0 0  0</td>
</tr>
<tr>
<td>5</td>
<td>0  0 1  0</td>
<td>0  0 1  0</td>
<td>0  0 0  0</td>
</tr>
<tr>
<td>6</td>
<td>0  0 1  1</td>
<td>0  0 1  1</td>
<td>0  0 0  0</td>
</tr>
</tbody>
</table>

\[
Overlap_M = \frac{1}{3}\left(\frac{0}{2} + \frac{0}{2} + \frac{0}{2}\right) = 0 \quad \frac{1}{3}\left(\frac{0}{2} + \frac{0}{2} + \frac{2}{2}\right) = 0 \quad \frac{1}{3}\left(\frac{2}{2} + \frac{2}{2} + \frac{2}{2}\right) = 1
\]

\[
Overlap_{ECS} = \frac{1}{3}\left(\frac{2}{6} + \frac{2}{6} + \frac{2}{6}\right) = \frac{1}{3} \quad \frac{1}{3}\left(\frac{2}{4} + \frac{2}{4} + \frac{2}{4}\right) = \frac{1}{3} \quad \frac{1}{3}\left(\frac{2}{4} + \frac{2}{4} + \frac{2}{4}\right) = 1
\]

Table 4.1 shows three examples with different coverage and overlap values produced by three slices of a six statements program. Each column represents a slice, in which a ‘1’ denotes a statement in the slice and a ‘0’ a statement not in the slice. In Example 1, three slices cover all six statements without any pairwise-overlap; in Example 2, the three slices cover four of the statements and \(SL_1\) and \(SL_2\) completely overlap each other, but not \(SL_3\); in Example 3, three slices cover the same two statements with full overlapping. The original \(Overlap_M\) has the same value for Examples 1 and 2 and thus does not distinguish between no overlap and partial
overlap, where as $\text{Overlap}_{ECS}$ not only does distinguish it but also can distinguish between different level of partial overlap.

### 4.3.3 Expected Results

The primary comparison made in the study is between CA and RN. In essence, the question being asked is “does using concept assignment to select a set of slicing criteria differ from merely selecting random vertices?” If there is no difference then an ECS is likely to become too large to be useful in practice. It is anticipated that the average ECS size will be similar to the average backward slice. This should be the case when conceptual coherence translates into dependencies as might be hoped.

Comparison of the resulting slices for CA with those for CN and CL allows the effect of conceptual coherence (as defined by the HB-CA algorithm) and spatial coherence (adjacent lines of code) to be better understood. If the spatial aspect of CA is the only size determinant, then slices on CN and CL will produce the same coverage value as corresponding CA criterion. Conversely, if conceptual coherence has merit then CA criterion will lead to smaller coverage.

This difference is unlikely to be large, because any consecutive sequence of statements has some conceptual coherence. To better understand the relationship, consider, for example, a program that is partitioned into concept bindings each consisting of exactly 10 statements. A randomly chosen sequence of 10 statements has a 10% probability of being identical to one of the existing concept bindings (if its starting point is the starting point of a concept). The other 90% of the time such a randomly selected sequence of 10 statements will include portions of exactly two concepts. If these two concepts depend upon one another, then the slices on both will include significant overlap. Therefore, it would be realistic to expect that CN
and CL would lead to a coverage increase compared to CA criterion, but it would be unrealistic to expect that there would be a large difference.

Overlap\textsubscript{ECS} is used to verify the presence of conceptual similarity as the slices on different conceptually related criteria are less likely to be independent of each other. In other words, they are very likely to show significant overlap. Thus, CA criterion should produce lower overlap than CN, CL, and RN.

### 4.3.4 Method

The case study was performed by executing WeSCA on the subject programs to generate a set of CA for each program. These were exported to data files that were imported into CodeSurfer and mapped into sets of SDG vertices that form the low-level slicing criterion. Finally, slices for each vertex of the set were computed using CodeSurfer. The result of each slicing operation is a set of vertices, and the union of these sets forms the vertices of the ECS.

To compare the effect of the different types, for each CA criterion, a corresponding CN, CL, and RN criterion of a matching size to the CA was generated. Since CN, CL, and RN involve an element of randomness, the average of 30 generations of each criterion was used for comparison.

Table 4.2 lists the number of concept bindings that were identified by WeSCA for each subject program. Over all ten programs there were a total of 631 concept bindings.
4.4 Results and Discussion

4.4.1 Relationship between Criterion Coherence and Slice Coverage

Coverage is based on the average slice, so the larger the value of Coverage, the larger the average slice size. Comparing the slices taken with respect to CA and RN, as shown in Figure 4.2, it is clear that RN always generates larger coverage than the more computationally coherent CA. This is important for two reasons. First, it provides evidence that HB-CA is identifying conceptually related vertices (and thus vertices with similar slices). Second, it provides evidence that slicing from a large set of vertices $V$ does not lead to a significant increase in slice size when $V$ has a degree of conceptual coherence (e.g., includes vertices representing a concept). In contrast, as can be seen from the results, slicing on a set of vertices $V$ that are purely random
4.4. Results and Discussion

Figure 4.2: Average Coverage value comparison for CA and RN.

(and therefore have no conceptual coherence) does lead to a significant increase in size.

It is possible that the larger coverage when using RN results from a lack of spatial locality, rather than a lack of conceptual locality. That is, instead of indicating that HB-CA identifies semantically related program components (and thus their slice is smaller), the results comparing CA and RN might indicate that selecting lexically close program components (spatial locality) produces smaller slices. CN and CL were included in the experiment in order to investigate this possibility. Both CN and CL are built from contiguous program statements and thus should have similar spatial locality with CA criterion.

Using CN or CL provides a very strict test. Although the starting points for both criteria are random, by including contiguous program statements, some contiguous sequence of one or more concepts binding will always be included. Indeed, this is inevitable, since concept assignment partitions the source code into a set of non-overlapping concepts. Figure 4.3 provides evidence that CA does lead to smaller
coverage increases than CN and CL. That is, in all but two cases, the CA lead to smaller size executable concept slices. Thus, it is very likely that conceptual locality and not spatial locality causes executable concept slices to be smaller than those computed using RN. In one of the two exceptions, the program EPWIC, CA, CN, and CL have essentially the same coverage, however in the other, the program userv, CN had a notably smaller coverage value. The cause of this is uncovered and discussed further in Section 4.4.3.

Finally, it is interesting to note that, for the programs with large dependence clusters, the coverage from all criteria show greater similarity. Figure 4.4 shows a box plot of the values and their variance. This is important because it demonstrates that dependence clusters could have an impact on any algorithm that exploits dependence information and thus evaluations of such algorithms should explicitly address the effect of such clusters.

Figure 4.3: Average Coverage for CA, CV, and CL.
4.4. Results and Discussion

4.4.2 Relationship between Criterion Coherence and Slices Overlap

Figure 4.5 presents the average overlap for each program and each type of criterion. Comparing the results for CA and RN, RN with its lack of conceptual coherence always generates greater overlap. This provides evidence that CA has greater con-
ceptual coherence. Comparing the results for CA with CN and CL, which have similar spatial locality, in all but two cases, CA leads to lower overlap in the executable concept slices; thus, it can be expected that conceptual locality and not spatial locality is the cause for CA’s smaller overlap. In one of the two exceptions, the program EPWIC-1, the difference is rather small. In the other, the program diffutils-2.8, CL produced a significant smaller overlap.

Inspecting the source code, of the five programs with large dependence clusters, diffutils-2.8 contains the largest percentage of comments and blank lines (37% LoC) and the smallest dependence cluster. Thus, it is very likely that the concept bindings selected by CL contain a large portion of comments, which results in the smaller slices than a real concept binding. Furthermore, if these vertices are not included in the dependence cluster then the size of the slices will be significantly smaller than the slice with respect to the criterion containing any vertex of the dependence cluster. The overlap is consequently smaller. The size of the ECS for diffutils for CL in Figure 4.8 provides evidence that many random binding selected by CL produce very small ECSs.

4.4.3 Relationship between Criterion Size and Slice Size

Scatter plots make an effective visualisation for investigating relationships between criterion size and slice size. To illustrate the information found in these plots, Figure 4.6 shows two representative examples for the CA criterion where the $x$ axis represents the size of criterion and the $y$ axis is the corresponding slice size. These two highlight the two different distributions seen. For the program space, which is free of large dependence clusters, the slice size varies even on the same criterion size. For the program userv, which contains a large dependence cluster, an obvious horizontal line appears in the figure at 60% on $y$ axis. This shows how, in the
presence of dependence clusters, many CA criteria of varying size produce the same slice size.

The scatter plots for all ten programs, are divided into two groups depending on the existence of large dependence clusters. These are shown in Figures 4.7 and 4.8, respectively. In both figures, the columns present each program and the rows the criterion types. The detail in each figure is less important than its overall shape. Finally, note that for CN, CL, and RN, slices from all 30 runs of the program are shown.

Examination of all four types shows that there is a clear difference in the pattern for the first five programs and the last five programs. The limited number of bands for the last five programs are caused by the presence of large dependence clusters. To better understand this, consider a program that is one large dependence cluster. Any slice of such a program will include a component from the cluster and thus will include the entire cluster (that is, entire program). In the more realistic case, the existence of a few significant dependence clusters means that only a limited number of output slice sizes are possible. For example, consider the scatter plot for indent using the RN criterion. Less the 10% of the slices managed to miss the one large dependence cluster (the remainder are similar enough in value to appear as two dots on the scatter plot). As a result the size of most slices is just over 80%. By contrast, using CA, over a dozen of the 49 concepts (25%) map to vertices outside this large dependence cluster. As shown in Figure 4.8, the impact of large dependence clusters is the presence of characteristic horizontal lines in the size graphs. These lines bear witness to the large interdependencies between program components.

Looking at each criterion type independently, the random criterion (RN) shows the expected increase in slice size accompanying an increase in criterion size. This effect is masked by large dependence clusters; thus, it is easier to see in the first
Figure 4.6: Two samples of scatter plots for slice size on CA criterion in Figures 4.7 and 4.8
Figure 4.7: Scatter plots to show the relationships between the size of criterion and the size of slices for all four types of criteria for the subjects without large dependence clusters.
Figure 4.8: Scatter plots to show the relationship between the size of criterion and the size of slices for all four types of criteria for the subjects with large dependence clusters.
four programs (Figure 4.7) than in the final five programs (Figure 4.8). The best example is the space program where the pattern is most apparent.

Statistically this kind of non-linear relationship should give rise to a strong Spearman rank correlation, which simply quantifies how frequently an increase in $y$ accompanies a (possibly non-linear) increase in $x$. For acct, EPWIC-1, space, oracolo2, and CADP, the data from the RN scatter plots have Spearman $R$’s coefficients of 0.91, 0.86, 0.87, 0.82, and 0.86, respectively. All being greater than 0.80, these indicate strong rank correlations.

This non-linearity can be understood by considering the selection of scattered vertices as a sequential process of adding slices to the union one at a time. The first few vertices selected have a much greater chance of covering previously uncovered parts of the program than those selected later. As more slices are added, there remains less and less uncovered program.

By contrast, with one exception, CA do not exhibit any predictive relationship between criterion size and slice size. This is visually apparent in the scatter plots. The one exception is acct where a linear correlation exists yielding a Pearson’s $R$ coefficient of 0.79 (a strong linear relation). The model shows a 19% increase in slice size accompanying a 1% increase in criterion size. This indicates that acct’s concepts are essentially nested (from a dependence standpoint) based on size; thus, there is rarely a smaller concept whose vertices depend on the vertices of a larger concept. This relationship holds for the graphed data in which the size of the criterion is less than 2% of the SDG vertices. Clearly this growth is not sustainable for larger criterion sizes.

Owing to the spatial locality of CN and CL, they include a limited number of concepts (most often 1 or 2). The scatter plots reinforce the results shown in Figures 4.2
4.5. Summary

and 4.3 and Figure 4.4. They show that the averages are not unduly influenced by any outliers and thus are a fair summary of the slice sizes. Finally, comparing the scatter plots for EPWIC-1 using CA and CN, the cause of the lower average for CN can be seen. The two plots have similar structure; however, the key difference lies in the larger number of small slices for CN (the dark line where \( y \) is close to zero). For CN there are a slightly larger proportion of criteria that produce these small slices and thus the slightly smaller average.

At this point it is appropriate to examine the extent to which the three research questions of this chapter have been addressed:

1. Does slicing on a set of conceptually related SDG vertices (i.e., those representing a single HB-CA concept) produce a smaller slice than slicing on other similarly sized sets? The results indicate that slicing on conceptually related SDG vertices produces smaller slices than would be expected if conceptually unrelated SDG vertices are used.

2. Is the size of an ECS related to the size of the HB-CA extracted concept binding? The results show that there is no predictive relationship between the size of the concept binding and the size of the resulting executable concept slice.

3. Do large dependence clusters\[21\] affect ECS size? The results show that large dependence clusters substantially affect the size of executable concept slices.

4.5 Summary

Though concept assignment derives a high degree of expressibility from the domain level at which its extraction criteria are expressed, it lacks the executability
of program slicing. However, slicing, though it produces executable subprograms, can only do this using extremely low level criteria. This chapter presents the first experimental evidence that it is possible to combine the expressive power of these concept assignment techniques with the semantic guarantees that accrue from the executability of program slicing. It experimentally studies executable concept slicing using ten subject programs. More than 600 concept bindings were identified and studied. The results from the study provide evidence to support the claim that the statements identified by concept assignment are not arbitrary: they form a coherent set of related parts of the program, because the slices constructed from them have a small degree of \text{Coverage} and \text{Overlap} between each other. This is an encouraging finding, because it indicates that there is no ‘size explosion’ when constructing executable concept slices.

The impact of large dependence clusters on this and other work suggests that identifying and handling these artefacts of dependence is an important issue to be addressed when evaluating any dependence-based source code analysis.
Chapter 5

Concept Abbreviation Using Slice-based Dependence Analysis *

5.1 Introduction

Concepts expressed in source code can be extended into an executable concept slice using program slicing. The previous chapter has described this technique and given the evidence that concepts produce smaller slices than randomly selected criteria. However, the consequent executable concept slice would not be smaller than the average slice size for each single program point, i.e., one third of the whole programs according to a large scale empirical study [24]. Furthermore, it is interesting to note that the concept bindings are not small themselves at all. The statistics show that the average concept binding size for ten programs are 44 LoC. Some of them are more than 300 LoC and the largest binding contains 1,525 LoC. Either these concept bindings or concept slices are too large to be analysed further. It is necessary to condense concept by identifying some statements within bindings to represent the

*This chapter is based on the work published by the author at SCAM 2008 [19]
5.2. Challenge

core of the concept itself. The reduction of the size of concepts can be helpful for users to focus attention more rapidly on the core of a concept binding.

The notion of Key Statement Analysis (KSA) that identifying those statements contribute more to the computation embodied by the binding than others was introduced by Harman et al. [47, 62] and focused on concept binding only.

This chapter presents an improved KSA$_{BO}$ algorithm in Section 5.3.4 that maintains the same impact and cohesion while reducing the size of the set of key statements. Section 5.3 proposes a framework for parameterised key statement analysis that supports wider applications in program comprehension and source code analysis. Two metrics are defined formally in Section 5.4. The chapter concludes by presenting an experimental study with the same ten programs used in previous chapter that shows the feasibility of the approach for carrying out general key statement analysis.

5.2 Challenge

Key Statement Analysis was originally defined as the process of key statements extraction from associated concepts. Two KSA algorithms, KSA$_{BO}$ and KSA$_{BE}$, are described [47]. Both algorithms compute key statements based on principal variables. This potentially leads to the issue that for concept bindings that do not involve principal variables, no key statement can be identified. KSA$_{BE}$ was found to be too complicated to implement [47]. For KSA$_{BO}$, if a concept binding contains only one principal variable, all statements within the binding that dependent upon this principal variable are key statements. In the worst case all statements in a concept binding could be key statements, making the key statement size as large as the associated concept binding. Furthermore, for concept bindings with smaller key statements, there is no method to evaluate the key statements, which requires
suitable metrics to be proposed.

For key statement analysis to be considered of practical use, it is important to establish that the key statements extracted by the algorithms are the same in contribution to the compilation of the whole program as the original concept binding. This can be determined by measuring the program parts which are potentially affected by the concept binding and associated key statements respectively. If the affected parts are close, the key statements are very likely to have the same contribution to the whole program as the concept binding does. Comparing key statements with concept bindings, key statements would be smaller in size and higher in cohesion. The investigation, therefore, addresses the following research questions:

1. Is the number of key statements considerably smaller than the number of statements found in the associated analysis scope from which they are extracted?

2. Do key statements have similar sized impact on the computation of the program as that of the overall associated analysis scope from which they are extracted?

3. Do key statements form a more cohesive unit than those of the associated analysis scope from which they are extracted?

4. Do (large) dependence clusters [21] affect key statement identification?

5.3 The Framework of Parameterised KSA

In this section, a parameterised key statement analysis approach is introduced with three stages: extraction – extracting the analysis scopes from the source code, decomposition – identifying the Principal Variables (PV) within the scope, and ab-
5.3. The Framework of Parameterised KSA

Abbreviation – computing the Key Statements (KS) within the scope based on the principal variables.

The framework of key statement analysis is shown in Figure 5.1. Scope, PV and KS in the dashed line box on the right part of flowchart are the output at each stage.

![Diagram](image)

Figure 5.1: The Framework of Parameterised Key Statement Analysis

A parameter is assigned for each stage that represents the available methods or algorithms. Since the goal of each stage can be achieved by variety methods, the range of the parameter values are the available methods or algorithms in each stage. They are described in detail in the following sections.

5.3.1 Extraction

In this study, the analysis scope is defined as a segment of source code, in which the statements are contiguous. Let \( P \) be the program and \( s \) be the statement in \( P \), that is

\[
P = \{s_1, \ldots, s_n|s_i \in P \land n = |P|\}
\]
Let $S$ be an analysis scope in $P$, that is

$$S = \{s_m, s_{m+1}, \ldots, s_n | s \in P \land m, n \in \mathbb{N} \land 1 \leq m \leq n \leq |P|\}$$

The analysis scope can be extracted from source code by many methods. Here two methods are presented: subroutine and high-level concepts.

### 5.3.1.1 Subroutine

Programs can be broken up into subroutines, such as functions, methods, procedures, or subprograms etc. A subroutine is a portion of source code within a larger program that performs a specific task which is relatively independent of the remaining code. Therefore, a subroutine contains a specific functionality. Subroutines are generally separated by natural segment boundaries such as procedure boundaries which make the extraction easy to be implemented. Thus, there are many advantages to consider the subroutines as the candidates of the analysis scope, such as the high cohesion and high level functionality of the scope, and cheap extracting algorithm etc.

### 5.3.1.2 High-Level Concepts

Programs can express domain-level concepts. Concept assignment aims to determine which concepts are implemented in a program and where those concepts are implemented. Extracting concepts based on domain knowledge lets concepts having a degree of conceptual cohesion, therefore concepts are much more focused on a specific meaning. This meaning is generally more ‘concentrated’ than that of procedures. That is, concept segmentation initially uses natural breakpoints such as procedure boundaries, so normally a concept is smaller than procedure in size.
Concepts as the analysis scope are easier to be understood and analysed than procedures because of the smaller size and high-level knowledge associated with the concept bindings. There are many approaches for concept assignment, such as Hypothesis-Based Concept-Assignment (HB-CA) \cite{46} and Latent Semantic Indexing (LSI) based approaches \cite{96}. These were discussed in Chapter 2.

Besides the subroutines and concepts, there are also other methods to extract portions of code from the program as the analysis scope. Users can even simply chop any interesting portion of code as the analysis scope by inspection on the source. In this experiment, the first parameter $P_{\text{Extraction}}$ can be defined as:

$$P_{\text{Extraction}} \in \{\text{Subroutines, Concept Bindings}\}$$

### 5.3.2 Decomposition

Variable identifiers are basic components of expressions in programs. Decomposing the statements into variables is a general analysis approach. However, not all variables need to be analysed at the beginning. Some variables play more important roles in programs than others, denoted as principal variables. In the second stage, the task is identifying these principal variables for a determined analysis scope.

For an analysis scope in source code, considering the more important variables, users may care more about which variables could be the result by the scope at first. These variables were considered as principal variables \cite{14, 100}. Ott et al. \cite{100} defined principal variables as follows:

A variable $v$ in a analysis scope $S$ is a principal variable, denoted by $v \in PV(S)$, iff it is
• a global variable and assigned in $S$, denoted by $PV_G$.

• a call-by-reference parameter and assigned in $S$, denoted by $PV_{CBR}$, or

• the parameter to an output statement of $S$, denoted by $PV_O$.

Three types principal variables can be employed independently to study the single contribution to the scope by each of them respectively, or the union by all three types of principal variable or by any two of them to study the pairwise contribution to the key statements identification.

Let $P_{\text{Decomposition}}$ be the parameter of the second stage, that is

$$P_{\text{Decomposition}} \in \{PV_G, PV_{CBR}, PV_O\}$$

### 5.3.3 Abbreviation

In the third stage, key statements are computed as the abbreviation of the analysis scope based on the principal variables identified in the second stage. Generally, the statements embodied by the scope and contributing the computation for all principal variables are extracted. For the different types of principal variables, there are corresponding types of key statements. Let $KS$ be the set of key statements in a scope $S$, $KS_O$ be the output key statement (the key statements based on output principal variables) and $KS_G$ be the global key statement (the key statement based on global principal variables).

The key statement analysis algorithms assign the relative weight for each statement in the scope that signifying whether a statement is a key statement. Let KSA() be the algorithm, then $KS$ can be defined as
5.3. The Framework of Parameterised KSA

\[ \mathcal{KS} = \{ s | s \in \mathcal{S} \land \text{KSA}(s) = 1 \} \]

There are two key statements analysis algorithms, \(\text{KSA}_{BO}\) and \(\text{KSA}_{BE}\), described in Section 2.5.

Let \(P_{\text{Abbreviation}}\) be the parameter of the third stage, that is

\[ P_{\text{Abbreviation}} \in \{ \text{KSA}_{BO}, \text{KSA}_{BE} \} \]

In case no principal variables are in a scope, or there are principal variables, but an algorithm outputs no statements, the result is that no key statements are identified for the scope.

### 5.3.4 An Improved KSA\(_{BO}\) Algorithm

The number of key statements could be large. An improved algorithm \(\text{KSA}'_{BO}\) that reduces the size of key statements without loss of its impact is introduced. It exploits the observation that statements in the forward slice of a key statement can be removed from the set of key statements.

Figure 5.2 presents the \(\text{KSA}'_{BO}\) algorithm used to identify key statements based on \(\text{KSA}_{BO}\). It takes as input an analysis scope \(\mathcal{S}\) from a program \(P\) and a set of principal variables \(\mathcal{PV}\). It computes the set of key statements \(\mathcal{KS}\). The algorithm, upon which that of Figure 5.2 is based, computes the key statements as those found in the intersection of the backward slices taken with respect to the final use of
5.3. The Framework of Parameterised KSA

function $\text{KSA}_{BO}'$ (Program $P$, Analysis Scope $S$, PVs $\mathbb{P}V$)

returns: a set of Statements from $S$

\{

for each variable $v$ in $\mathbb{P}V$

let $\text{slice}_i = \overline{S}(P, \text{Finaluse}(S, v))$

endfor

let $\text{KS} = \text{Statements}(S) \cap \{ \bigcap_i \text{slice}_i \}$

for each (remaining) statement $s_i$ in $\text{KS}$

for each key statement $s_j \neq s_i$ in $\overline{S}(S, s_i)$

remove $s_j$ from $\text{KS}$

endfor

endfor

return $\text{KS}$

\}

Figure 5.2: $\text{KSA}_{BO}'$ Algorithm for identifying those statements that effect all of an analysis scope’s principal variables.
5.3. The Framework of Parameterised KSA

each principal variable. In the figure, Finaluse(\mathcal{S}, v) is used to denote the final use of variable $v$ of analysis scope \mathcal{S}. Comparing with KSA_{BO}, KSA'_{BO} removes the statements from the set of key statements if they are in the forward slice of a key statement.

5.3.5 Example

Figure 5.3 shows an example of the key statement computation based on the set of principal variables \mathbb{PV}_O. The example function computes the area and volume of a cylinder given its radius. From Lines 7 and 8, \mathbb{PV}_O includes \{area, volume\}. The intersection of the backward slices taken with respect to the final uses of these two principal variables includes only Line 3 (shown boxed in Figure 5.3). Thus, Line 3 is the sole key statement. Here the intuition behind this statement’s keyness is that the variable undersurface contributes to the computation of both area and volume variables.

1 Diameter=2*r;
2 perimeter=PI*Diameter;
3 undersurface=PI*r*r;  
4 sidesurface=perimeter*h;
5 area=2*undersurface+sidesurface;
6 volume=undersurface*h;
7 printf("\nThe Area is %d", area);
8 printf("\nThe Volume is %d", volume);

Figure 5.3: An example of key statement computed by KSA_{BO} algorithm for the concept binding.
5.4 Key Statement Evaluation

In this section, two metrics are introduced, Impact and Cohesion, to evaluate the key statements. These two metrics represent two analyses: Impact analyses the relationship between analysis scope and associated key statements, and Cohesion analyses the cohesion of key statements themselves.

5.4.1 Metric 1: Impact

Impact analysis aims to determine and assess the effects of making a set of changes to a software system. It finds the ripple effects of changing given statements of criterion. Forward Slicing \([70]\), revealing the program parts that are potentially affected by the slicing criterion, has been previously used in identification of the candidate impact statements. Roughly, the larger the forward slice size, the bigger impact of the criterion.

In this study, Impact is defined to measure the relative influence of a set of key statements \(S\) compared to the influence of the analysis scope from which they were drawn. Semantically, this influence can be defined as those statements computing with a tainted value when the statements if \(S\) introduce the taint \([119]\). This slice is taken with respect to the set of SDG vertices that represent the statement in \(S\).

For a single statement \(s\) from analysis scope \(S\) of program \(P\), Impact is the ratio of the size of the forward slice taken with respect to \(s\) to the size of the forward slice taken with respect to the statements of \(S\):

\[
\text{Stmt-Impact}(s, S, P) = \frac{|\mathcal{S}(P, s)|}{|\mathcal{S}(P, S)|}
\]
5.4. Key Statement Evaluation

For a set of statements, the average impact is used in the evaluation. Given a set of (key) statement $S$ from analysis scope $S$ of program $P$, $\text{Impact}(S, S, P)$ is defined as follows:

$$\text{Impact}(S, S, P) = \frac{1}{|S|} \sum_{s \in S} \text{Stmt-Impact}(s, S, P)$$

$$= \frac{1}{|S|} \sum_{s \in S} \frac{|\bar{S}(P, s)|}{|\bar{S}(P, S)|}$$

$$= \frac{\sum_{s \in S} |\bar{S}(P, s)|}{|S| \times |\bar{S}(P, S)|}$$

A set of key statements $S$ is always a subset of $S$; thus, value of $\text{Impact}(S, S, P)$ ranges between zero and one, where zero occurs when there are no key statements and one when the slice on each key statement is identical to the slice taken with respect to the entire analysis scope. Conceptually, the higher the value of $\text{Impact}$ the more key the key statements. In the sequel $\text{Impact}$ is used in place of $\text{Impact}(S, S, P)$ when the statements, analysis scope, and program are clear from the context.

5.4.2 Metric 2: Cohesion

Whereas impact measures the outward influence of the key statements, $\text{Cohesion}$ measures their inward connectedness. It compares the influence of each key statement to the influence of all the key statements taken collectively. Greater overlap in the influence indicates higher cohesion. High cohesion is obtained when each
of the key statement has a similar influence. For a set of \( n \) key statements with fixed impact, the range of cohesion values is from \( 1/n \), where each key statement had a completely separate influence, to one, where each key statement has exactly the same influence. Taken together, high Cohesion means the key statements have similar influence and high Impact means that they have a large influence.

Cohesion is computed as the ratio of the average slice size for the key statements to the size of the slice taken with respect to all the key statements collectively. For the set of key statements \( S \) from program \( P \), Cohesion \((P, S)\) is defined as follows:

\[
\text{Cohesion}(P, S) = \frac{1}{|S|} \sum_{s \in S} \left| \mathcal{S}(P, s) \right| \left| \mathcal{S}(P, S) \right|
\]

\[= \sum_{s \in S} \frac{\left| \mathcal{S}(P, s) \right|}{|S| \times \left| \mathcal{S}(P, S) \right|}
\]

As with Impact, Cohesion is used in place of Cohesion \((P, S)\) when the program and set of key statements are clear from the context. The definition of Cohesion slightly develops based on Weiser’s original slice-based metric Overlap [123] when the intersection of the set of slices is empty. Weiser’s metric is defined as the average ratio of the size of the intersection of all slices to the size of each slice. When the intersection is empty, this metric takes the value zero. The difference is illustrated in Table 4.1 on Page 85, which shows three examples that have different Cohesion and Overlap values.
5.5 Experimental Study

5.5.1 Subject Programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>No. User Defined Functions</th>
<th>No. Concept Bindings (HB-CA)</th>
<th>Large Dependence Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct-6.3.2</td>
<td>50</td>
<td>24</td>
<td>No</td>
</tr>
<tr>
<td>EPWIC-1</td>
<td>124</td>
<td>63</td>
<td>No</td>
</tr>
<tr>
<td>space</td>
<td>136</td>
<td>67</td>
<td>No</td>
</tr>
<tr>
<td>oracolo2</td>
<td>135</td>
<td>41</td>
<td>No</td>
</tr>
<tr>
<td>CADP</td>
<td>450</td>
<td>24</td>
<td>No</td>
</tr>
<tr>
<td>userv-1.0.1</td>
<td>114</td>
<td>52</td>
<td>Yes</td>
</tr>
<tr>
<td>indent-2.2.6</td>
<td>48</td>
<td>49</td>
<td>Yes</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>94</td>
<td>74</td>
<td>Yes</td>
</tr>
<tr>
<td>diffutils-2.8</td>
<td>91</td>
<td>109</td>
<td>Yes</td>
</tr>
<tr>
<td>findutils-4.2.25</td>
<td>202</td>
<td>128</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,444</strong></td>
<td><strong>631</strong></td>
<td></td>
</tr>
</tbody>
</table>

The same ten programs are studied in this experiment. Table 5.1 lists each program with the number of user defined functions and the number of concept bindings identified. Over all ten programs there were a total of 1,444 user defined functions and 631 concept bindings.

Recall that five programs are known to be free of large dependence clusters, while five programs known to contain large dependence cluster.
5.5.2 Analysis Tools

The data per location of concept bindings are same as those in previous chapter which were identified by the WeSCA tool [51]. The CodeSurfer[53] tool’s API is used to identify principal variables in analysis scopes and to implement the KSA algorithm to find key statements.

5.5.3 KSA Parameters Used in the Experiment

In the parameterised key statement analysis, the parameters offer different choices. The values assigned for each parameter in the experiment are described as follows.

\[ P_{\text{Extraction}} = \{ \text{FUNCTIONS, HB-CA} \} \]

Decomposition

In the C programming language, there is no call-by-reference parameter, although it can be simulated by passing the address of a variable. In the experiments, Global and Output principal variables are identified, and the union of Global and Output principal variable is studied as well. Thus, the parameter of principal variable kinds
5.5. Experimental Study

offers 3 natural choices:

\[ P_{\text{Decomposition}} = \{ PV_G, PV_O, PV_G \cup PV_O \} \]

For a scope \( S \) in a program \( P \), \( SDG_P \) are the corresponding system dependence graph. Global principal variables are those global-formal-out vertices in \( SDG_P \) control dependent on \( S \)'s entry vertex; Output principal variables are those actual-in vertices in \( SDG_P \) control dependent on a call to output statement such as ‘printf’, ‘write’ etc.

Abbreviation

The algorithm KSA\(_{BE}\) is a more elaborate approach that can assign weights to each statement to indicate relative impact. Unfortunately, this approach is based on the dependence graph and the shortest path between each variable to the principal variables needs to be computed, which is extremely computationally expensive. Thus it is not possible in our current implementation. In this chapter, the improved KSA\(_{BO}\) is employed to compute key statements, that is

\[ P_{\text{Abbreviation}} = \{ \text{KSA}'_{BO} \} \]

Figure 5.4 summarises all assigned parameter values for three stages in the experiment. Each path in the graph indicates one kind of key statement identified by the specific parameters’ values. For example, the path (Functions → \( PV_O \) → KSA\(_{BO}\)) is the output principal variable based key statement computed by the KSA\(_{BO}\) algorithm in functions of the program. As shown in the figure, there are total six types of key statement to be investigated in this study.
5.6 Results and Discussion

This section discusses the results of principal variables, key statements and their relation to analysis scope. The results of two metrics on key statement evaluation are presented with statistical analysis for function and concept binding respectively. The effects of large dependence clusters are also discussed.

This section first discusses the results related to the analysis scopes that contains principal variable and key statements. It then considers the principal variables discovered in the analysis scope. It also discusses the key statements identified from the three kinds of principal variables and compares their size, Impact and Cohesion. This is followed by a comparison of the number of key statements compared to the number of statements in the analysis scope from which they are extracted. The section finally returns to the four research questions from the introduction and interprets the experimental results in terms of these questions.

Figure 5.4: The value of parameters used in the experiment and their combinations
5.6. Results and Discussion

5.6.1 Analysis Scope

Key statement analysis depends on the principal variables within analysis scope. But an analysis scope containing principal variables does not mean it must have any key statement, since, based on the KSA algorithm, the key statements is the intersection of backward slices on principal variables, which could be empty.

In this section, the number of analysis scopes containing principal variables and the number of analysis scopes containing key statements are counted respectively. This will give the general picture of the identification of principal variables and key statements. Furthermore, comparing two numbers, if the latter is smaller than the former, that is the number of analysis scopes with key statements is smaller than the number of analysis scopes with principal variable, it will show the cases that some analysis scopes contain principal variables but no key statements.

5.6.1.1 Analysis Scopes Containing Principal Variable

Table 5.2 summarises the number of analysis scopes (functions and concept bindings) containing different types of principal variables for each program. Overall, 373 of the total 1,444 user defined functions and 231 of the total 631 concept bindings contain principal variables. In programs space and oracolo2, no global variable is used. In acct, EPWIC-1 and CADP which without large dependence clusters, the number of scopes with global principal variables is much smaller than scopes with output principal variables. However, the same result is not shown in the five programs with large dependence clusters. This may be explained by the large amount of global variables in programs making them more likely to result in large dependence clusters.
Table 5.2: The Summary of number of functions and concept bindings containing principal variables.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Number of Functions</th>
<th></th>
<th></th>
<th></th>
<th>Number of CBs</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>$PV_G \cup PV_O$</td>
<td>$PV_O$</td>
<td>$PV_G$</td>
<td>$PV_G \cup PV_O$</td>
<td>$PV_O$</td>
<td>$PV_G$</td>
<td>$PV_G$</td>
</tr>
<tr>
<td>acct-6.3.2</td>
<td>50</td>
<td>24</td>
<td>24</td>
<td>2</td>
<td>24</td>
<td>19</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>EPWIC-1</td>
<td>124</td>
<td>22</td>
<td>18</td>
<td>9</td>
<td>63</td>
<td>16</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>space</td>
<td>136</td>
<td>41</td>
<td>41</td>
<td>0</td>
<td>67</td>
<td>22</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>oracolo2</td>
<td>135</td>
<td>41</td>
<td>41</td>
<td>0</td>
<td>41</td>
<td>15</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>CADP</td>
<td>450</td>
<td>65</td>
<td>57</td>
<td>9</td>
<td>24</td>
<td>10</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>userv-1.0.1</td>
<td>114</td>
<td>38</td>
<td>29</td>
<td>11</td>
<td>52</td>
<td>12</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>indent-2.2.6</td>
<td>48</td>
<td>26</td>
<td>14</td>
<td>16</td>
<td>49</td>
<td>36</td>
<td>18</td>
<td>27</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>94</td>
<td>44</td>
<td>16</td>
<td>36</td>
<td>74</td>
<td>41</td>
<td>17</td>
<td>33</td>
</tr>
<tr>
<td>diffutils-2.8</td>
<td>91</td>
<td>34</td>
<td>30</td>
<td>9</td>
<td>109</td>
<td>40</td>
<td>26</td>
<td>17</td>
</tr>
<tr>
<td>findutils-4.2.25</td>
<td>202</td>
<td>38</td>
<td>10</td>
<td>29</td>
<td>128</td>
<td>20</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>1,444</td>
<td>373</td>
<td>280</td>
<td>121</td>
<td>631</td>
<td>231</td>
<td>149</td>
<td>111</td>
</tr>
</tbody>
</table>
5.6. Results and Discussion

Figure 5.5 and Figure 5.6 present the number of analysis scopes for each sole type principal variable for functions and concept bindings respectively. For functions, 75% (1,071) do not contain any principal variables; 17% (252) and 6% (93) functions have sole output and global principal variables respectively; only 2% (28) functions have both kinds of principal variables.

![Number of Functions](image)

Figure 5.5: The number of functions with and without principal variables

![Number of Concept Bindings](image)

Figure 5.6: The number of concept bindings with and without principal variables

Compared to functions, the total number of concept bindings is much smaller than the number of functions (631:1,444), but the percentage of the concept bindings containing principal variables is larger than that in functions (25%:37%). There is still a large number of concept bindings (400, 63%) that do not have any principal variable. In the concept bindings with principal variables, 19% (120) and 13% (82)
of them have sole output and global principal variables respectively, and in 5% (29) of concept bindings both kinds of principal variables are exist. This is a slightly larger percentage than for functions.

In the C language, a concept binding is a subset of all statements of a function, but in one function there may be more than one concept binding identified. If the probability to extract concept bindings for all functions is same, the percentage of the scopes with principal variables in functions should be same as that in concept bindings. However, the experimental results show the higher percentage value of the scopes with principal variables in concept bindings. Comparing two analysis scopes, functions are separated by natural segment boundaries, and HB-CA concept binding are separated based on lexical analysis with domain knowledge. This experimental results may give the evidence that HB-CA is more likely to catch the principal variables during the segmentation.

Both Figure 5.5 and 5.6 show that there are a large number of analysis scopes without any output or global principal variables. These analysis scopes, by definition, will have no key statements. This suggests that one direction for future investigation is the consideration of additional kinds of principal ‘variables’. For example, expressions returned by a function.

5.6.1.2 Analysis Scopes Containing Key Statement

According to the KSA computation algorithm in Figure 5.2, key statements are those statements within the analysis scope that are involved in all computations of principal variables. If there is more than one principal variable in an analysis scope, it is possible that no statement is involved in all the principal variables’ computation. In such cases, the analysis scope might be divided into several parts, where each part
Table 5.3: The Summary of numbers of functions and concept bindings containing Key Statements.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Number of Functions</th>
<th>Number of CBs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>$\text{KS}_{\text{Union}}$</td>
</tr>
<tr>
<td>acct-6.3.2</td>
<td>50</td>
<td>24</td>
</tr>
<tr>
<td>EPWIC-1</td>
<td>124</td>
<td>22</td>
</tr>
<tr>
<td>space</td>
<td>136</td>
<td>41</td>
</tr>
<tr>
<td>oracolo2</td>
<td>135</td>
<td>41</td>
</tr>
<tr>
<td>CADP</td>
<td>450</td>
<td>65</td>
</tr>
<tr>
<td>userv-1.0.1</td>
<td>114</td>
<td>38</td>
</tr>
<tr>
<td>indent-2.2.6</td>
<td>48</td>
<td>26</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>94</td>
<td>44</td>
</tr>
<tr>
<td>diffutils-2.8</td>
<td>91</td>
<td>34</td>
</tr>
<tr>
<td>findutils-4.2.25</td>
<td>202</td>
<td>38</td>
</tr>
<tr>
<td>Total</td>
<td>1444</td>
<td>373</td>
</tr>
</tbody>
</table>
5.6. Results and Discussion

concentrates on the computation for a single principal variable. It is necessary to investigate this issue for all programs.

Table 5.3 summaries the number of functions and concept bindings containing a variety of types of key statement for all programs. Comparing these data with those in Table 5.2 which are number of functions and concept bindings having principal variables, there is no difference for the function numbers for all programs, but the number of concept bindings having key statements is different with that having principal variables. These differences are illustrated in Figure 5.7. It presents two number of concept bindings: concept bindings containing principal variables and concept bindings containing key statements. These data are presented for all subject programs for three pairs of principal variables and key statements respectively.

It is clear that the number of concept bindings with principal variables is always larger than those with key statements for all three pairs for all programs. This implies that in some concept bindings with principal variables, no key statement is identified using the current algorithm.

Note that this difference does not exist for the functions, where for those functions with principal variables, key statement always can be found. This can be explained by the difference of the dependence structure between functions and HB-CA based concept bindings. In system dependence graph (SDG), a function corresponds to a program dependence graph (PDG) in which there is always an entry vertex and an exit vertex. The statement with entry vertex is always identified as key statement, because any variable in a PDG depends upon the entry vertex of the PDG. But such cases would not occur for concept bindings, as an HB-CA concept binding is a sub set statements of a function. The entry vertex may or may not be included in a concept binding.
Figure 5.7: The number of concept bindings having principal variables and key statements in three pairs respectively for all programs
5.6. Results and Discussion

5.6.2 Principal Variable

Principal variable can be identified automatically using CodeSurfer scripting language. In this section, the number of principal variables identified in functions and concept bindings are presented respectively. Over all ten programs, there are a total of 2,494 principal variables in functions and 1,565 principal variables in concept bindings.

5.6.2.1 Principal Variables in Functions

The total of 2,494 principal variables in functions in the ten programs studied include 953 global principal variables and 1,541 output principal variables.

Figure 5.8 shows three scatter plots, one for each kind of principal variable. Each plot shows the number of principal variables per function on the $y$-axis and each program on the $x$-axis. In all three plots, most functions have between 1 to 25 principal variables. Comparing the charts for $PV_O$ (Output PV) and $PV_G$ (Global PV), the number of output principal variables exist in a tighter range of values. Thus, over the ten programs considered, the output of variables is more consistent than the modification of global variables. The lower plot shows the distribution for $PV_O \cup PV_G$ (Union PV). As the union of $PV_O$ and $PV_G$, this scatter plot shows where $PV_O$ or $PV_G$ dominate the union. Table 5.4 presents the mean and standard deviation for the Union data. It is interesting to note that the mean number of principal variables does not appear to be affected by the presence of large dependence clusters. However, the presence of large dependence clusters does appear to produce a higher standard deviation (at least with some programs).

This large standard deviation is partially caused by two clear outliers. Interestingly, as seen in the scatter plots, one each attributed to $PV_O$ (Output PV) and $PV_G$
Figure 5.8: The number of principal variables per function for each of the three kinds of principal variables.
Table 5.4: The mean and standard deviation of the number of principal variables in $\text{PV}_O \cup \text{PV}_G$.

<table>
<thead>
<tr>
<th>Program</th>
<th>Functions</th>
<th>Mean</th>
<th>Std. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct-6.3.2</td>
<td>24</td>
<td>7.13</td>
<td>10.49</td>
</tr>
<tr>
<td>EPWIC-1</td>
<td>22</td>
<td>8.59</td>
<td>12.23</td>
</tr>
<tr>
<td>space</td>
<td>41</td>
<td>3.12</td>
<td>4.85</td>
</tr>
<tr>
<td>oracolo2</td>
<td>41</td>
<td>3.90</td>
<td>8.21</td>
</tr>
<tr>
<td>CADP</td>
<td>65</td>
<td>5.89</td>
<td>5.22</td>
</tr>
<tr>
<td>userv-1.0.1</td>
<td>38</td>
<td>4.00</td>
<td>2.81</td>
</tr>
<tr>
<td>indent-2.2.6</td>
<td>26</td>
<td>21.08</td>
<td>45.46</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>44</td>
<td>7.73</td>
<td>19.55</td>
</tr>
<tr>
<td>diffutils-2.8</td>
<td>34</td>
<td>6.35</td>
<td>9.20</td>
</tr>
<tr>
<td>findutils-4.2.25</td>
<td>38</td>
<td>5.42</td>
<td>12.53</td>
</tr>
<tr>
<td>Total</td>
<td>373</td>
<td>6.68</td>
<td>15.96</td>
</tr>
</tbody>
</table>

(Global PV). The two, having over 100 principal variables, are shown as the solid points and labeled ‘1’ and ‘2’ in the lower scatter plot. Looking at the source code, Point ‘1’ comes from the main function of the program indent. Inspection of this function reveals that it is a large function with 1,657 LoC and 222 principal variables (219 of which come from $\text{PV}_G$). As comments in the code indicate, the program’s author chose to communicate values from main to the rest of the program using global variables; thus, creating a large pool of globals many of which are modified.

Point ‘2’, the second outlier, is from the function yyparse which contains 1,190 LoC and 131 principal variables. Here, 102 are from $\text{PV}_O$ and 29 from $\text{PV}_G$. From an inspection of the source code for this Bison-generated parser, it is evident that there is a significant number of output statements devoted to generating error messages associated with various error conditions that the parser may encounter.
5.6. Results and Discussion

5.6.2.2 Principal Variables in Concept Bindings

The total of 1,565 principal variables in concept bindings in the ten programs studied include 667 global principal variables and 898 output principal variables.

Figure 5.9 shows three scatter plots, one for each kind of principal variable. Each plot shows the number of principal variables per function on the y-axis and each program on the x-axis. In all three plots, most functions have between 1 to 20 principal variables which is a little bit of smaller than the PV number in functions.

This is not a surprising result, since the concept binding is a subset of the statements of the function and it should contain less principal variables than the function. Comparing the charts for \( PV_O \) and \( PV_G \), the number of output principal variables exist in a tighter range of values. This is the same as the result for \( PV_O \) and \( PV_G \) in functions, that is over the ten programs considered, the output of variables is more consistent than the modification of global variables. The lower plot shows the distribution for \( PV_O \cup PV_G \). As the union of \( PV_O \) and \( PV_G \), this scatter plot shows where \( PV_O \) or \( PV_G \) dominate the union.

Table 5.5 presents the mean and standard deviation for the Union data. It is similar to PV in functions that there is no obvious difference of the mean number of principal variables for programs with and without large dependence clusters.

This large standard deviation is partially caused by a few outliers. The largest one which is shown as the solid point and labeled ‘1’ in the lower scatter plot.

This concept binding includes 77 principal variables. Inspection of the source code reveals that this concept binding is extracted from the function \texttt{indent} in subject \texttt{indent}, which contains the largest number of principal variables in all functions in ten programs. This concept binding has 346 SLoC, which is 37.28% of the whole function. All 77 principal variables are from \( PV_G \), which is not surprising since most principal variables are global in this function.
Figure 5.9: The number of principal variables per concept binding for each of the three kinds of principal variables.
5.6. Results and Discussion

Table 5.5: The mean and standard deviation of the number of principal variables in $\mathbb{PV}_O \cup \mathbb{PV}_G$.

<table>
<thead>
<tr>
<th>Program</th>
<th>Functions</th>
<th>Mean</th>
<th>Std. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct-6.3.2</td>
<td>19</td>
<td>5.68</td>
<td>4.55</td>
</tr>
<tr>
<td>EPWIC-1</td>
<td>16</td>
<td>6.50</td>
<td>8.03</td>
</tr>
<tr>
<td>space</td>
<td>22</td>
<td>4.05</td>
<td>6.18</td>
</tr>
<tr>
<td>oracolo2</td>
<td>15</td>
<td>9.40</td>
<td>15.53</td>
</tr>
<tr>
<td>CADP</td>
<td>10</td>
<td>15.60</td>
<td>18.66</td>
</tr>
<tr>
<td>userv-1.0.1</td>
<td>12</td>
<td>4.50</td>
<td>4.30</td>
</tr>
<tr>
<td>indent-2.2.6</td>
<td>36</td>
<td>11.89</td>
<td>16.95</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>41</td>
<td>6.41</td>
<td>7.99</td>
</tr>
<tr>
<td>diffutils-2.8</td>
<td>40</td>
<td>3.90</td>
<td>2.79</td>
</tr>
<tr>
<td>findutils-4.2.25</td>
<td>20</td>
<td>3.20</td>
<td>3.00</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>231</strong></td>
<td><strong>6.77</strong></td>
<td><strong>10.34</strong></td>
</tr>
</tbody>
</table>

5.6.3 Key Statements

Three types of key statements are computed based on the three kinds of principal variables: from $\mathbb{PV}_G$ *global key statements*, denoted $\mathbb{KS}_G$, from $\mathbb{PV}_O$ *output key statements*, denoted $\mathbb{KS}_O$, and from $\mathbb{PV}_G \cup \mathbb{PV}_O$ *union key statements*, denoted $\mathbb{KS}_{Union}$, respectively.

This section compares the size of key statements and two metrics, *impact* and *cohesion* as defined in Section 5.4, for all three types key statements. Finally, we take the union key statement as representative result of KSA and compare it to the associated function or concept binding using the same metrics. The results are presented with statistical analysis.
5.6. Results and Discussion

5.6.3.1 Key Statement Size

To gain an initial impression for the data, the size of the set of key statements is first compared to the size of the associated analysis scope. Here, a lower value is preferred. For example, the ratio for 2 key statements in a 5 statement function is 40%, while 2 key statements in a 10 statement function is only 20%.

Table 5.6: The Summary of average key statement size (as a percentage of scope size) for functions and concept bindings respectively.

<table>
<thead>
<tr>
<th>Programs</th>
<th>KS in Functions</th>
<th>KS in Concept Bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KS\textsubscript{\text{Union}}</td>
<td>KS\textsubscript{O}</td>
</tr>
<tr>
<td>acct</td>
<td>36.8% 36.8% 45.0%</td>
<td>13.3% 12.3% 11.9%</td>
</tr>
<tr>
<td>EPWIC</td>
<td>13.0% 12.6% 14.9%</td>
<td>15.7% 14.1% 13.7%</td>
</tr>
<tr>
<td>space</td>
<td>15.6% 15.6% 0.0%</td>
<td>12.7% 12.7% 0.0%</td>
</tr>
<tr>
<td>oracolo2</td>
<td>12.9% 12.9% 0.0%</td>
<td>7.4% 7.4% 0.0%</td>
</tr>
<tr>
<td>CADP</td>
<td>31.5% 32.9% 25.9%</td>
<td>9.8% 15.2% 1.7%</td>
</tr>
<tr>
<td>userv</td>
<td>52.2% 56.6% 30.1%</td>
<td>25.5% 26.8% 27.6%</td>
</tr>
<tr>
<td>indent</td>
<td>11.7% 8.2% 13.1%</td>
<td>12.3% 14.2% 9.2%</td>
</tr>
<tr>
<td>bc</td>
<td>25.5% 29.2% 21.6%</td>
<td>18.6% 23.1% 17.0%</td>
</tr>
<tr>
<td>diffutils</td>
<td>22.3% 19.5% 23.6%</td>
<td>16.4% 11.7% 26.9%</td>
</tr>
<tr>
<td>findutils</td>
<td>22.9% 25.0% 21.6%</td>
<td>17.8% 25.2% 6.0%</td>
</tr>
<tr>
<td>Total</td>
<td>25.2% 25.4% 21.9%</td>
<td>15.2% 15.2% 15.1%</td>
</tr>
</tbody>
</table>

Figure 5.10 and 5.11 present the average ratio of the key statement size to associated functions size and concept binding size respectively for all programs. Three bars represent union key statements, output key statements and global key statements. The corresponding numeric ratios are presented in Table 5.6. Overall, the average for each kind is similar in single analysis scope, being around 25% in functions and 15% in concept bindings.
5.6. Results and Discussion

Figure 5.10: The average key statements size in functions for all programs

Figure 5.11: The average key statements size in concept bindings for all programs
Two relevant observations about the key statements can be made from this graph. First, several of the averages for $\KS_G$ show what appears, at first sight, to be an interesting anomaly whereby the percentage for $\KS_{\text{Union}}$ is lower than that for $\KS_G$ (a similar pattern exists between $\KS_O$ and $\KS_{\text{Union}}$). This occurs when the backward slice taken with respect to each of the global principal variables in $\PV_G$ includes similar statements, while the backward slice taken with respect to the variables in $\PV_O$ includes different statements. Thus, when considering $\PV_G \cup \PV_O$, a smaller set of key statements (i.e., a smaller intersection) is identified. That this pattern is not more pronounced supports the notion that the key statements identified are truly key to the computation.

The second observation is that, for all three types of key statements, the absence of large dependence clusters appears to correspond to smaller sets of key statements. Furthermore, large dependence clusters seem to accompany lower variability in the average over all subject programs. This is most pronounced for $\KS_G$. This suggests that large dependence clusters are masking the variability seen in the cluster-free programs. This provides evidence that large dependence clusters affect KSA and further that they do so through global variables.

5.6.3.2 Impact

This section considers the impact for each of the three types of key statements in functions and concept bindings respectively. To begin with, for the key statements to truly form the nucleus of an analysis scope’s computation then they should have an impact similar to that of the scope from which they are extracted.
5.6. Results and Discussion

Key Statement Impact in Functions

Figure 5.12 presents the average impact value for the key statements in functions for all programs and Figure 5.13 shows the associated box plot of the impact value and their variance. The impact is only computed for the functions containing key statement.

In the experiment, \textit{Impact} has the value 100% when the key statements have the same impact as the function from which they are extracted.

As can be seen in Figure 5.12, $KS_G$ tends to have a slightly higher \textit{Impact} than $KS_O$. Overall, the \textit{Impact} of the key statements exceeds 70% of that of the associated function for 19 of the 28 bars (discounting the two programs that have no globals). These include 7 of 10 for $KS_{Union}$, 7 of 10 for $KS_O$, and 5 of 8 for $KS_G$.

The distribution of the \textit{Impacts}, shown in Figure 5.13, clearly demonstrates the undesirable effect of large dependence clusters [21]. Consider first, the top row of Figure 5.13, which shows the box plots for $KS_{Union}$. In the absence of large dependence clusters, the data are distributed and have only a few outliers. By contrast, in four of the five dependence–cluster–having programs, the ‘box’, in essence, includes only the median value. These programs also have a considerable number of outliers and extreme values. For $KS_O$ the data is better (three programs show such cases), but still shows the negative impact of large dependence clusters. There are fewer globals, but the data still point to dependence clusters generating a greater number of outlier and extreme values. Finally, more than the means, the median values show how in the first five programs (those without large dependence clusters), \textit{Impact} is more consistent.
5.6. Results and Discussion

Figure 5.12: Average **Impact** value for each type of key statement in functions for all programs

![Figure 5.12: Average Impact value for each type of key statement in functions for all programs](image)

Figure 5.13: The boxplot of **Impact** in functions for all programs

![Figure 5.13: The boxplot of Impact in functions for all programs](image)
5.6. Results and Discussion

Figure 5.14: Average impact value for each type of key statement in concept bindings for all programs

Figure 5.15: The boxplot of impact in concept bindings for all programs
Key Statement Impact in Concept Bindings

Figure 5.14 presents the average impact value for the key statements in concept bindings for all programs and Figure 5.15 shows the associated box plot of the impact value and their variance. The average impact is only computed for the concept bindings with key statements.

Comparing the average impact of the key statements in concept bindings with that in functions which showed in Figure 5.12, there is no obvious difference between them. \( KSG \) tends to have a slightly higher impact. Overall, the impact of the key statements exceeds 70% of that of the associated function for 17 of the 28 bars (discounting the two programs that have no globals). These include 7 of 10 for \( KS_{Union} \), 5 of 10 for \( KS_O \), and 5 of 8 for \( KS_G \).

The distribution of the impacts in concept bindings, shown in Figure 5.15, is also similar as that in functions. The undesirable effect of large dependence clusters [21] is demonstrated by the narrow distribution (the ‘box’ in essence includes only the median value) for \( KS_{Union} \) and \( KS_G \) for the five dependence–cluster–having programs.

The similar impact of the key statements in functions and in concept bindings provides the evidence that our parameterised KSA can identify high impact key statement in both kinds of analysis scope.

5.6.3.3 Cohesion

This section considers the cohesion for each of the three types of key statements in functions and concept bindings respectively. Cohesion was only computed for the analysis scopes with two or more key statements because cohesion is, by definition, always 100% in the case of a single key statement.
5.6. Results and Discussion

Key Statement Cohesion in Functions

Figure 5.16 presents the average cohesion of the key statements in functions for all programs and Figure 5.17 shows the associated box plot of the cohesion value and their variance.

Higher Cohesion comes from higher overlap between the forward slices taken with respect to key statements. As seen in Figure 5.16, Cohesion for \( \mathsf{KS}_{\text{Union}} \), \( \mathsf{KS}_O \), and \( \mathsf{KS}_G \) is always greater than 75%. This shows that each type of principal variable tends to generate a highly cohesive set of key statements.

For programs with large dependence clusters, there is a high probability that a given key statement will be within a large cluster and consequently have the same influence as other key statements that are also within the cluster. As such, the Cohesion for such programs is expected to be similar and higher than that of programs free from large dependence clusters. The data in Figure 5.16 bear this out. In particular, the bars for the rightmost five programs are higher than those on the left.

Similar to the box plots for Impact, the box plots for Cohesion in Figure 5.17 show how the presence of dependence clusters all but remove any spread in the data. With the union data, the distribution of Cohesion in four of the programs with dependence clusters is essentially 100%. In exception subject diffutils there are some variance but the median line is still at 100%.

Key Statement Cohesion in Concept Bindings

Figure 5.18 presents the average cohesion value of the key statements in concept bindings for all programs and Figure 5.19 shows the associated box plot of the cohesion value and their variance.
5.6. Results and Discussion

Figure 5.16: Average **Cohesion** value for each type of key statement in functions for all programs

Figure 5.17: The boxplot of **Cohesion** in functions for all programs
5.6. Results and Discussion

Figure 5.18: Average Cohesion value for each type of key statement in concept bindings for all programs

Figure 5.19: The boxplot of Cohesion of key statements in concept bindings for all programs
As can be seen in Figure 5.18, $KS_G$ tends to have a slightly higher Cohesion. There are no Cohesion of $KS_G$ for programs acct and findutils, but Table 5.3 shows there are 3 concept bindings in acct and 10 in findutils containing key statements. Note that cohesion was only computed for the analysis scopes with two or more key statements (Cohesion is always 100% in the case of a single key statement), thus these concept bindings only include a single key statement in $KS_G$ which is not counted in this chart.

Comparing the key statement extracted from concept bindings with that extracted from functions (which was shown in Figure 5.16), the Cohesion is less than 70% of that of the associated concept binding for only 4 of 28 bars. The overall distribution of the key statement cohesion in concept bindings in Figure 5.19 is similar as that in functions, but the data shows higher convergence because of less outliers in Figure 5.19.

### 5.6.4 Key Statements and their Associated Analysis Scope

The previous section compared the Impact and Cohesion of the three kinds of key statements with each other. This section compares the Impact and then the Cohesion of the key statements with the Impact and Cohesion of the analysis scope from which they are drawn. Because the same metric is computed for the matched pair of key statements and the analysis scope from which these statements are drawn, and because there can be no certainty that the distribution is normal, a Wilcoxon Matched-Pairs Signed Ranks Test procedure was applied to test the significance of the difference. In this case, the null hypothesis is there is no difference between the metric for the key statements and associated analysis scope.
5.6. Results and Discussion

5.6.4.1 Impact

Figure 5.20 and 5.21 present the comparison between the impact of the key statements from $\text{KS}_\text{Union}$ taken from analysis scope $\text{S}$ with $\text{Impact}(\text{KS}_\text{Union}, \text{S}, P)$ that of $\text{S}$ itself, that is $\text{Impact}(F, F, P)$ of associated functions and $\text{Impact}(\text{CB}, \text{CB}, P)$ of concept bindings respectively.

It is clear that the key statements always have higher impact than the associated functions or concept bindings. The Wilcoxon Matched-Pairs Signed Ranks Test procedure is employed to test the significance of the difference, where the null hypothesis is there is no difference between the impact of key statements and that of the associated analysis scope.

Table 5.7: Wilcoxon Matched-Pairs Signed Ranks Test comparing impact of key statements with their associated function and concept binding.

<table>
<thead>
<tr>
<th>Function</th>
<th>No.</th>
<th>Analysis Scope</th>
<th>Z-value</th>
<th>Asymp.Sig.(2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{KS}<em>\text{Union} - \text{Function}</em>\text{Union}$</td>
<td>300</td>
<td>-12.819</td>
<td>.000</td>
<td></td>
</tr>
<tr>
<td>$\text{KS}_O - \text{Function}_O$</td>
<td>225</td>
<td>-10.618</td>
<td>.000</td>
<td></td>
</tr>
<tr>
<td>$\text{KS}_G - \text{Function}_G$</td>
<td>101</td>
<td>-8.515</td>
<td>.000</td>
<td></td>
</tr>
<tr>
<td>Concept Binding</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{KS}<em>\text{Union} - \text{CB}</em>\text{Union}$</td>
<td>150</td>
<td>-7.244</td>
<td>.000</td>
<td></td>
</tr>
<tr>
<td>$\text{KS}_O - \text{CB}_O$</td>
<td>104</td>
<td>-4.967</td>
<td>.000</td>
<td></td>
</tr>
<tr>
<td>$\text{KS}_G - \text{CB}_G$</td>
<td>68</td>
<td>-6.518</td>
<td>.000</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7 includes the results of the statistical analysis. For each kind of key statement, it presents the standardised signed-ranks difference (Z-Value) and statistical significance $p$-value (Asymp.Sig.(2-tailed)). If the $p$-value shown in the final column less than 0.05, each null hypothesis can be rejected and the difference taken as statistically significant at 95% level.
5.6. Results and Discussion

Figure 5.20: The average impact for $\KS_{\text{Union}}$ and the associated functions. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)

Figure 5.21: The average impact for $\KS_{\text{Union}}$ and the associated concept bindings. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)
The first three rows of Table 5.7 concern Function. For all three kinds of key statements, the p-value is less than 0.05; thus, the null hypothesis can be rejected. Hence, the difference between the Impact of each kind of key statement is significantly different from that of the corresponding function. Visually, Figure 5.20 suggests that the impact for KS Union is higher than that of the associated function.

The last three rows of Table 5.7 include the results for Concept Binding. For the three types of key statements, the p value is less than 0.05; thus, the null hypothesis is rejected in each case and the difference taken to be statistically significant. This provides the evidence that the key statements of KS Union have higher Impact than associated function.

5.6.4.2 Cohesion

Figure 5.22 and 5.23 present the comparison between the average Cohesion of the key statements with associated functions and concept bindings respectively. It is clear that the key statements always have higher Cohesion than the associated functions or concept bindings. Similarly, the Wilcoxon Matched-Pairs Signed Ranks Test is employed to test the significance of the difference between the Cohesion of key statements and that of the associated scope. The null hypothesis is that there is no difference between the Cohesion of key statement and that of the associated analysis scope.

Tables 5.8 presents the data of paired-samples t test for two pairs: functions with associated key statements and concept bindings with associated key statements. As the significance values for both pairs are less than 0.05 which rejects the null hypothesis, the difference between both pairs are significant. This provides the evidence that the key statements have higher cohesion than associated analysis scope.
5.6. Results and Discussion

Figure 5.22: The average Cohesion for $KS_{Union}$ and the associated functions. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)

Figure 5.23: The average Cohesion for $KS_{Union}$ and the associated concept bindings. (Note that lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)
Table 5.8: Wilcoxon Matched-Pairs Signed Ranks Test comparing Cohesion of key statements with their associated function and concept binding.

<table>
<thead>
<tr>
<th>Function Scope</th>
<th>No. Analysis Scope</th>
<th>Z-value</th>
<th>Asymp.Sig.(2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( KS_{Union} ) - Function_{Union}</td>
<td>196</td>
<td>-11.920</td>
<td>.000</td>
</tr>
<tr>
<td>( KS_{O} ) - Function_{O}</td>
<td>154</td>
<td>-10.303</td>
<td>.000</td>
</tr>
<tr>
<td>( KS_{G} ) - Function_{G}</td>
<td>61</td>
<td>-6.791</td>
<td>.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concept Binding</th>
<th>No. Analysis Scope</th>
<th>Z-value</th>
<th>Asymp.Sig.(2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( KS_{Union} ) - CB_{Union}</td>
<td>63</td>
<td>-5.123</td>
<td>.000</td>
</tr>
<tr>
<td>( KS_{O} ) - CB_{O}</td>
<td>49</td>
<td>-4.073</td>
<td>.000</td>
</tr>
<tr>
<td>( KS_{G} ) - CB_{G}</td>
<td>30</td>
<td>-4.541</td>
<td>.000</td>
</tr>
</tbody>
</table>

### 5.6.5 Summary of Results

Having presented the data gathered in the experimental study, this section concludes by returning to the four research questions:

1. **Is the number of key statements considerably smaller than the associated analysis scope from which it is extracted?**
   The data presented in Table 5.6 presents the average number of key statements in \( KS_{Union} \), \( KS_{O} \), and \( KS_{G} \) in functions as 25.2%, 25.4%, and 21.9%, and in concept bindings as 15.2%, 15.2%, and 15.1%, respectively. This data, visually supported by Figure 5.10 and Figure 5.11, which show that for all the three kinds of principal variables, a considerable reduction of the key statements size is achieved in both functions and concept bindings.

2. **Do key statements have similar impact on the computation of the program as the associated analysis scope from which they are extracted?**
5.6. Results and Discussion

The data presented in Figure 5.12 and Figure 5.14 show that the Impact of the key statements exceeds 70% of that of the associated function for 19 of 28 cases and the associated concept bindings for 17 of 28 cases. Furthermore, from Figure 5.20 and Figure 5.21 and the statistics in Table 5.7, the average impact of the key statements is higher than both that of functions and that of concept bindings. Thus, the parts of a program impacted by an analysis scope but not its key statements are, in some sense, less consequential computations.

3. Do key statements form a more cohesive unit than those of the associated analysis scope from which they are extracted?

Figure 5.16 and Figure 5.18 show that the key statements computed for each of the three types of principal variable have similar cohesion for both functions and concept bindings. Furthermore, as shown in the Table 5.8 and visually apparent in Figure 5.22 and Figure 5.23, key statements have a higher Cohesion than the statements of either the function or concept binding from which they are drawn.

4. Finally, Do (large) dependence clusters affect key statement identification?

There is clear support for an affirmative answer to this question. Starting with Table 5.2, the general distribution of the principal variables hints at a difference: dependence-cluster free programs tend to have a greater proportion of their principal variables in $\mathbb{PV}_O$, while programs with large dependence clusters tend to have a larger proportion of their principal variables in $\mathbb{PV}_G$. In Figure 5.12 and Figure 5.14 (and the following box plots), the variability in Impact is higher for programs with large dependence clusters. Finally, large dependence clusters produce more cohesive sets of key statements, as seen in Figure 5.16 and Figure 5.18. Thus, large dependence clusters have a clear and largely negative impact on key statement identification.
5.7 Summary

In this Chapter, the approach of concept abbreviation, parameterised key statement analysis, is presented. The framework together with an improvement over previous approaches to KSA [62, 47] is shown. The keyness of the identified statements is measured using their Impact and Cohesion. The experiments were implemented not only for concept bindings but also for functions. The results indicate that the identified key statements have high Impact and Cohesion and thus represent the core of a function’s computation. The identification of key statements can be helpful for users to focus attention more rapidly on the core of a concept binding.

Key statements are computed as an optimised intersection of the intraprocedural static backward slices on Bieman and Ott’s ‘principal’ variables. The results reveal that only approximate one third analysis scope contains principal variables.
Chapter 6

Concept Refinement: Dependence Based Concept Assignment

6.1 Introduction

The previous two chapters presented the experimental studies on concept extension and abbreviation using dependence analysis, which produced executable concept slices and key statements respectively from the concept bindings identified by concept assignment.

This chapter presents a third approach that combines concept assignment and dependence analysis, namely concept refinement. Using dependence analysis, this approach refines the HB-CA concept bindings, from which highly dependent statements are extracted, without losing domain knowledge. The refined concept binding contains the same impact as the original HB-CA concept binding but has a smaller size by ruling out the non-concept-dependent statements.

In the following sections, the framework of concept refinement is presented in Sec-
tion 6.3, where the three steps of the approach are explained. Two metrics that measure the refined concept bindings are introduced in Section 6.6. The chapter concludes with an experimental study based on the same ten programs used throughout the thesis, which shows the feasibility of the dependence based concept refinement approach.

6.2 Challenge

Chapter 4 presents the executable concept slice as the dependence extension with respect to the concept binding, where the technique enlarges the concept substantially to make it executable. This technique may be useful in program maintenance and testing. However, the large size of the executable concept slice produced may also limit its application.

Chapter 5 presents the key statement as an impact-based core statement within the concept. The set of all key statements is much smaller than the original concept. Key statement analysis is useful for program comprehension, however experimental results reveal an important limitation of the technique: approximately only one third of the total concept bindings contain key statements according to the dependence based approach.

This chapter introduces a new approach to combining concept bindings with dependence analysis. Instead of extending or abbreviating concept bindings, the new approach refines the concept binding itself using dependence analysis, consequently resulting in a dependence based concept binding.

The goal of this approach is to extract both highly domain concept related and highly interdependent structures from HB-CA concept bindings, rather than to find
a small set of core statements (as key statement analysis does). However, the size of the refined concept binding is also a matter of concern. Comparing a pair of concept bindings before and after refinement, a size reduction shows that the refinement approach does remove some statements from the original concept binding. Furthermore, a similar impact of such concept bindings shows that those removed statements do not affect impact largely, that is, they are more likely to be non-concept-dependent statements.

Comparing refined concept binding with original HB-CA concept binding, it is expected that the refined concept binding would have a similar impact and with a higher cohesion, while being reduced in size. The investigation therefore leads to the following research questions:

1. Does concept refinement reduce the size of the original HB-CA concept binding?

2. Do refined concept bindings have similar impact on the computation of the program as that of the original HB-CA concept bindings?

3. Do refined concept bindings form a more cohesive unit than those of the original HB-CA concept bindings?

4. Do refined concept bindings still relate to the same domain concepts as the original HB-CA concept bindings?

5. Do (large) dependence clusters [21] affect concept binding refinement?
6.3. Framework of Concept Refinement

Figure 6.1: The Framework of Concept Refinement
6.3 Framework of Concept Refinement

Figure 6.1 shows the framework of the concept refinement. The input to the process is the concept bindings identified using the HB-CA approach. These bindings are processed via three steps indicated by the dotted line boxes on the figure. The output are refined dependence based concept bindings. The three steps are explained in detail as follows.

**Step 1: Build Dependence Graph for Concept Bindings**

In the first step dependence graphs for concept bindings are built, where the vertices are the program points of the binding and the edges are the data dependence and control dependencies between vertices.

Note that an HB-CA concept binding is extracted from a procedure, so the dependence graph for a concept binding can be formed by extracting a subgraph from the PDG of the corresponding procedure in which the concept binding resides. This subgraph only includes the vertices corresponding to the program points in the concept binding and edges between the vertices.

The concept binding’s dependence graph is optimised after extraction. This is done by removing some kinds of vertices (e.g. entry and exit vertex) and edges (e.g. parameter-in, parameter-out, and call edges). The details are presented in Section 6.5.2.

For a concept binding two types of dependence graph are considered based on the direction of edges. One is the forward dependence graph in which the edges’ direction is forward, denoted by $\overrightarrow{DG_{CB}}$; the other is backward dependence graph in which the edges’ direction is reversed (backward), denoted by $\overleftarrow{DG_{CB}}$. 
Step 2: Identify Source and Sink of Concept Bindings

The essence of refinement is using program chopping to extract the statements involved in a transitive dependence from one specific statement (the source criterion) to another (the sink criterion). Thus, in the second step the Vertex Rank Model (VRM) is presented to identify the source and sink for a concept binding.

Generally the VRM ranks all vertices in a dependence graph by assigning a weight value to each vertex based on the dependencies between them. A vertex with higher dependence (directly or indirectly) upon others is assigned a higher value. The details of the VRM are presented in Section 6.4.

Applying the VRM to a forward dependence graph (\(DG_{CB}\)), the vertex with the highest weight value is the vertex that receives the most ‘dependencies’ from the others (i.e., the sink vertex). In the situation that multiple vertices have the same highest value, all these vertices are treated as sink vertices. Similarly, to identify the source vertices by applying the VRM to a backward dependence graph (\(DG_{CB}\)), the vertices with highest weight value are source vertices.

Step 3: Refine the Concept Binding by Chopping

The last step, chopping, is used to extract the refined dependence-based concept bindings.

Chopping is a generalisation of slicing. Chopping a program identifies a subset of its statements that account for all ‘transmit’ effects from source to sink. The vertices in a chop taken with respect to a source and sink, are the intersection of the forward slice with respect to the source and the backward slice with respect to the sink. If the source and sink both relate to the concept, the chop are those statements that
depend on each other and relate to the concept. In this study, the chop is taken as the refined concept binding.

Note that the source and sink identified by the VRM may not necessarily be solo vertices, then the chopping criteria would be all combinations of pairs consisting of each vertex from the source and sink. The largest chop is taken as the final refined concept binding.

Example

```c
1 Diameter=2*r;
2 perimeter=PI*Diameter;
3 undersurface=PI*r*r;
4 sidesurface=perimeter*h;
5 area=2*undersurface+sidesurface;
6 volume=undersurface*h;
7 printf("\nThe Area is %d\n", area);
8 printf("\nThe Volume is %d\n", volume);
```

Figure 6.2: An example of dependence based concept binding.

Figure 6.2 provides a simple example that refines an HB-CA concept binding to the corresponding dependence based concept binding. The portion of source code is the computation of cylinder area and volume. Suppose the domain model includes indicators for the concept area, indicating the computation of the area concept. The fifth and seventh statements include an indicator (the variable area) for the area concept. In this simplified example, Lines 5-7 are assigned to the area concept, which is indicated using bold font in Figure 6.2. Note that HB-CA generally assigns concepts to regions rather than single lines.
6.3. Framework of Concept Refinement

Table 6.1: The corresponding SDG vertices for the area concept binding in Figure 6.2.

<table>
<thead>
<tr>
<th>Vertex Kind</th>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>area=2*undersurface+sidesurface</td>
</tr>
<tr>
<td>expression</td>
<td>volume=undersurface*h</td>
</tr>
<tr>
<td>call-site</td>
<td>printf()</td>
</tr>
<tr>
<td>actual-in</td>
<td>&quot;\nThe Area is %d\n&quot; in call printf()</td>
</tr>
<tr>
<td>actual-in</td>
<td>area in call printf()</td>
</tr>
</tbody>
</table>

In this concept binding there are five SDG vertices corresponding to the source code. These are listed in Table 6.1 with the vertex kind and corresponding source.

Figures 6.3(a) and 6.3(b) show the original dependence graph and corresponding optimised dependence graph for the example concept binding area in Figure 6.2 respectively. In Figure 6.3(a), there is a data edge from the expression vertex area=2*undersurface+sidesurface to the actual-in vertex area and two control edges from call-site vertex printf() to both actual-in vertices. Before applying the VRM, the two control edges from the call-site vertex are removed, as Figure 6.3(b) shows.

In this very simple example, only one edge is left after optimisation. Thus it is clear that the expression vertex area=2*undersurface+sidesurface is the source and the actual-in vertex area is the sink. After chopping, Lines 5 and 7 are returned as the refined dependence based concept, denoted by the boxed lines in Figure 6.2.

Compared with the original HB-CA concept binding, Line 6 (that is not involved in the computation of area) is removed. The refined concept binding is more accurate because the statements remaining are only related to the concept of area and the non-concept-dependent statement Line 6 is removed.
6.3. Framework of Concept Refinement

(a) Original dependence graph

(b) Filtered dependence graph

Figure 6.3: The dependence graphs for the area concept binding in Figure 6.2. The solid arrows represent data flow dependence edges and dashed arrows represent control dependence edges.
6.4 Vertex Rank Model

In this section the detail of the Vertex Rank Model (VRM) is presented, which is based on the eigenvector centrality measure of a graph. Essentially, the VRM is the PageRank Model [27] used by Google to rank web pages.

6.4.1 Eigenvector Centrality

This subsection is closed based on the account given by Freeman [44].

Centrality [44] is a measure of a vertex within a graph that determines the relative importance of a vertex within the graph. This has been widely used in network analysis. The simplest measurement is degree centrality, which is defined as the number of links incident upon a vertex. A more sophisticated centrality, Eigenvector Centrality, takes into account not simply the number, but also the quality of the links – it assigns relative scores to all vertices in the graph following the principle that connections to high-scoring vertices contribute more to the score of the vertex in question than equal connections to low-scoring vertices. The VRM is an application of the eigenvector centrality measure.

6.4.2 Vertex Rank Model Algorithm

These definitions are based on those given by Inoue et al. [71].

Let $G = (V, E)$ be the dependence graph for a concept binding; $G$ is the directed graph, in which $N$ represents the vertices and $E$ represents the edges, i.e., the dependencies between vertices.
Let $u \rightarrow v$ denote the existence of an edge from $u$ to $v$. The graph can be regarded as a network, in which the dependence is transitive (i.e., if $(v_1 \rightarrow v_2)$ and $(v_2 \rightarrow v_3)$, then it is implied that $(v_1 \rightarrow v_3)$).

Let $w(v_i)$ be the weight value of $v_i$, which represents the importance of the dependence of a vertex; $w(v_i)$ will be distributed following the outgoing edges of $v_i$ and inherited through the incoming edges of $v_i$.

For a vertex $v_i$ in $G$, all vertices linked to $v_i$ via incoming edges, i.e. the predecessors of $v_i$, are denoted by $\text{Pred}(v_i)$ and all vertices linked by outgoing edges of $v_i$, i.e. the successors of $v_i$, are denoted by $\text{Succ}(v_i)$.

Let $d_{ij}$ be the distribution ratio of $v_i \rightarrow v_j$, where $0 \leq d_{ij} \leq 1$ and total of $d_{ij}$ for each $v_i$ is 1, that is

$$\sum_{v_j \in \text{Succ}(v_i)} d_{ij} = 1 \quad (6.1)$$

It is assumed that the weight of a vertex is the sum of the weights inherited through the incoming edges and that this incoming weight is equally distributed along the outgoing edges (i.e., if a vertex $v_i$ in a graph $G$ with the weight $w(v_i)$ has 3 outgoing edges and 5 incoming edges, then each successor vertex linked by outgoing edges receives only one-third of $w(v_i)$, and $w(v_i)$ is equal to the sum of the weights received by 5 predecessor vertices linked by incoming edges).

Let $\textbf{W}$ be the vector of a vertex’s weights, that is
6.4. Vertex Rank Model

\[ W = \begin{pmatrix} w_{v_1} \\ w_{v_2} \\ \vdots \\ w_{v_n} \end{pmatrix} \]

where

\[ n = |V| \]

and

\[ w(v_i) = \sum_{v_k \in \text{Pred}(v_i)} d_{ki} \times w(v_k) \]

Let \( D \) be the matrix of the distribution ratios, that is

\[ D = \begin{pmatrix} d_{11} & d_{12} & \ldots & d_{1n} \\ d_{21} & d_{22} & \ldots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \ldots & d_{nn} \end{pmatrix} \]

Therefore, the weights of vertices can be computed iteratively, that is

\[ W' = D^T W \] \hspace{1cm} (6.2) \]

where \( D^T \) is the transposed matrix of \( D \).

The computation starts with a non-zero initial value, \( W^0 \), and computes successive iterations \( W^{(k)} = DW^{(k-1)} \), until all weights become stable. Figure 6.4 shows the top level VRM algorithm.
6.4. Vertex Rank Model

function VRM(D, W₀, ϵ) {
    W = W₀;
    do
        W' = DᵀW;
        δ = ||W' - W||₁;
        W = W';
    while (δ > ϵ);
    return W;
}

Figure 6.4: The VRM algorithm

In general initial values W₀ may change the final weight of vertices W, but they will not affect the rank. For simplicity in the calculation, initial weights of 1/n (n is the total number of vertices) is assigned to each vertex. The algorithm terminates when stability is reached. Stability is measured as the difference between two iterations below a threshold ϵ, which is assigned 0.0001 in this study.

Figure 6.5: An example of initial weights and stable weights assigned to vertices and edges

Figure 6.5 shows an example of a dependence graph with initial weights and final computed weights, where v₁ has two outgoing edges that both have a 0.5 distribution
ratio (i.e., \( d_{12} = d_{13} = 0.5 \)); \( v_2 \) and \( v_3 \) have only one outgoing edge, thus \( d_{23} = d_{31} = 1 \). The initial weights for each vertex are 0.34, 0.33 and 0.33, and the sum of weights is 1. These weights are computed iteratively until stability is achieved, as shown by the right hand side graph in Figure 6.5, where the weights of the three vertices become 0.4, 0.2 and 0.4 respectively.

### 6.4.3 Convergence of Computation

The dependence graph for a concept binding is extracted from the PDG of the procedure from which the concept binding is extracted. There is an entry vertex and an exit vertex in each PDG, where the entry vertex has no incoming edges and the exit vertex has no outgoing edges. As shown in Figure 6.4, the VRM algorithm computes the vertices’ weights iteratively, and it is terminated only when the weights become stable. However, at the exit vertex the transition of the flow will be broken down because there is no outgoing edges, thus the iteration computation terminates abnormally. This is so-called “sink problem” [27] that can be avoided by introducing pseudo dependence (pseudo edges in PDG) between all vertices.

The weight of a vertex can transit along both real dependence and pseudo dependence edges. According to Equation (6.1), the sum of successors’ weights must be 1. Therefore, to satisfy this equation, the original distribution ratio needs to be amended after the introduction of pseudo edges. In order to guarantee that there is no significant effect on final vertices’ rank as a result of the introduced pseudo dependence, the distribution ratio is amended to distribute the weight with the most flowing along the real outgoing edges and only a small portion along the pseudo outgoing edges, rather than equally distributing the weights on all outgoing edges. The resulting weight values are affected by distribution ratio, but the resulting ranks are insensitive to the distribution ratio.
The distribution ratio needs to be amended in two cases. For those vertices without successors (i.e., $|\text{Succ}(v_i)| = 0$), the pseudo edges are added from $v_i$ to all vertices (including themselves), and the distribution ratio is assigned equally, that is $1/n$. For those vertices with successors (i.e., $|\text{Succ}(v_i)| > 0$), the pseudo edges are added from $v_i$ to all other non-successor vertices, and a small value is assigned to these pseudo edges introduced. Let $p(0 < p < 1)$ be the ratio between the weight of real edges and that of pseudo edges. A fairly large value is assigned to $p$ based on the discussion above. In this study, 0.85 is used, thus only 85% of $w(v_i)$ distributes along the real edges to successors, and only 15% along the pseudo edges. The amended distribution ratio $d'_{ij}$ is

$$d'_{ij} = \begin{cases} 
1/n & \text{if } |\text{Succ}(v_i)| = 0 \\
 p \times d_{ij} + (1 - p)/n & \text{if } |\text{Succ}(v_i)| > 0 
\end{cases}$$

Figure 6.6 presents an example of distribution amendment, in which the graph on the right is the amended dependence graph corresponding to the original one on the left. The dashed lines are the pseudo edges added, while compared with the solid lines which are the real edges. The original distribution ratio matrix and corresponding amended distribution ratio matrix are given under the two dependence graphs in Figure 6.6.

This example dependence graph includes three vertices (i.e. $n = 3$). To re-calculate the distribution ratio after adding the pseudo edges, because all three vertices have successors, formula $d'_{ij} = p \times d_{ij} + (1 - p)/n$ is used. For example,

$$d'_{11} = p \times d_{11} + (1 - p)/n = 0.85 \times 0 + (1 - 0.85)/3 = 0.05,$$

$$d'_{12} = p \times d_{12} + (1 - p)/n = 0.85 \times 0.5 + (1 - 0.85)/3 = 0.475.$$
6.4. Vertex Rank Model

Figure 6.6: An example of initial dependence graph and amended dependence graph with distribution ratio matrix ($p = 0.85$).
6.4.4 Computational Complexity

The VRM has to be invoked for each identification of source and sink vertices. Consequently, it is imperative that the underlying algorithm be efficient. A complexity analysis of the algorithm is presented in this section.

The main part of the VRM algorithm consists of calculating the weights of vertices. According to Equation (6.2), the computation is matrix multiplication. The cost of multiplying a $n \times m$ matrix by a $m \times p$ matrix is $O(nmp)$. Equation (6.2) has $n \times n$ by $n \times 1$ therefore the VRM algorithm’s cost is $O(n^2)$, where $n$ is the number of vertices. Note that the analysis scope is concept bindings, and then the number of vertices, $n$, would be small.

6.5 Source and Sink Identification Using the VRM

This section presents the definitions of source and sink that form the chopping criteria for a given concept binding and an approach to their identification using the VRM.

6.5.1 Concept Binding’s Source and Sink

In general, the source and sink are considered as the start and end vertices in a dependence flow path. For example, the source is the initialisation of a variable and the sink is the final use of the same variable in the flow path. In terms of a dependence graph, the source can be considered as a vertex without incoming edges and a sink as the vertex without outgoing edges.

A procedure’s source and sink can be easily identified. There is an entry vertex
and an exit vertex for a PDG and all vertices within the PDG are generally linked
together directly or indirectly, thus the entry and exit vertices can be simply taken
as source and sink. However, it is not so simple to identify the source and sink for
a HB-CA concept binding as it is for a procedure. A concept binding is a portion
of the source code within a procedure, and the corresponding dependence graph is
a sub-PDG that contains only the program points in the concept binding and their
dependencies, so there is no guarantees that an entry and an exit vertex both exist
in a concept binding’s dependence graph.

Furthermore, the dependencies from the program points within the concept bind-
ing to those program points without the concept binding may be broken when
extracting the concept binding’s dependence graph. It is much more likely that
there are multiple start and/or end vertices or even separated sub-graphs in the
extracted dependence graph (i.e., the separated dependence flows in the graph).
For instance, if the dependences in a tree-root structure are broken, such as the
statement \texttt{switch}(p) \texttt{case value1:... case value2:...}, the extracted sub-PDG
may contain two or more start or end vertices. Dependence graphs containing, ei-
ther multiple start and/or end vertices, or separated sub-graphs, raise the question:
“which vertices should be recognised as source and sink?”

To address this problem, the VRM is used to identify the source and sink for a
concept binding. As discussed in Section 6.4.3, pseudo edges are added between all
vertices without real edges to ensure convergence. Those additional pseudo edges
also solve the problem of separating sub-graphs in a dependence graph, since they
link all unlinked vertices, thereby linking the separated sub-graphs together. Note
that, as discussed in Section 6.4.3, the distribution ratio for pseudo edges is very
small, which will not affect the final rank.

To identify the sink (respectively source), the concept binding’s forward dependence
6.5. Source and Sink Identification Using the VRM

graph $\overrightarrow{DG_{CB}}$ (respectively backward dependence graph $\overrightarrow{DG_{CB}}$) in the VRM is applied to rank all vertices and the vertex with the highest weight value is the sink (respectively source).

The source and sink identified using the VRM are more likely to be related to the original concept. HB-CA concept bindings are extracted based on the string fragments of identifiers, keywords or comments that are related to the domain concepts. One specific HB-CA concept binding typically contains several identified strings to indicate the domain concept. For example these strings might be the same variable that are redefined and reused in the concept. In such cases, dependences may exist between these variable instances and there are likely to be more dependences than those between variables unrelated to the concept. These “strongly connected variables” are likely to have a high rank using the VRM.

6.5.2 Optimising Concept Binding’s Dependence Graph for the VRM

A concept binding’s graph includes control dependence and data dependence. There is no specific structure for data dependence in a dependence graph. However, for structured programs, control dependence tends to be “tree like” rather than graph like [20]. The tree like structure of control dependence comes from the following circumstances:

1. control dependences from call-site vertex to actual-in and actual-out vertices for each of the parameters of a called procedure,

2. control dependences from entry vertex to vertices which are not nested within any loop or conditional,
3. the control dependence from a control predicate to vertices nested within associated loop or conditional containing the predicate.

The first two types of control dependence produce a tree-root-like structure (all edges link to the same root vertex). The third type of control dependence reflects the nesting of control structures which gives the tree shape to the control dependence, producing a hierarchical control dependence graph: a tree-branch-like graph.

Consider the PDG examples shown in Figure 6.7. The control dependence from the entry vertex to each non-nested vertex gives rise to a tree-root-like control dependence graph, as shown in Figure 6.7(a). The nesting of decision constructs naturally give rise to a hierarchical, tree-branch-like control dependence graph, as shown in Figure 6.7(b).

In a backward dependence graph in which the direction of edges are reversed, such tree-root-like or tree-branch-like structure will dominate the VRM computation, because the dependences are concentrated along the tree structure to the root vertex. This can cause the root vertex to gain a extremely high weight in the VRM.

In a tree-branch-like structure that reflects the nature of control dependence in a program, the root vertex is the predicate expression vertex, which is the core of analysis in most cases. However, in a tree-root-like structure, the entry or call-site vertex is the root vertex, which are not of interest for a concept binding, since they are not associated with the domain concept.

In this study, the entry vertex (a pseudo vertex) and its consequent control dependence are removed from the dependence graph; the call-site vertex that represents a call site in the program is reserved, but the consequent control dependences which are tree-root-like structures are removed.
6.5. Source and Sink Identification Using the VRM

```c
int add (int a, int b)
{
    int sum=0;
    sum = a+b;
    return sum;
}
```

(a) Entry vertex give rise to tree-root-like control dependence graphs

(b) Standard control constructions give rise to tree-branch-like control dependence graphs

Figure 6.7: Examples tree-root-like and tree-branch-like control dependence graphs
Though data dependence and control dependence reflect different kinds of dependence in a program, they are not distinguished in the dependence graph for the VRM algorithm. That is, the weight value for a vertex will be equally distributed along data edges and control edges. In practice data dependences tend to be much more numerous than control dependences in SDG, especially after optimisation, where the control edges from entry vertex and call-site vertex are removed.

### 6.6 Evaluation

This section presents two metrics \textit{Impact} and \textit{Cohesion} to evaluate the refined dependence based concept binding.

#### 6.6.1 Impact

Impact measures the outward influence of a concept binding. Forward slicing is used in the definition of impact for a concept binding as discussed in Section 5.4.1. Roughly the larger the forward slice, the larger the impact of the vertex (or set of vertices) from which the slice was computed.

For a single statement $s$ from program $P$, \textit{Impact} is the ratio of the size of the forward slice taken with respect to $s$, to the size of program $P$:

$$\text{Impact}(s, P) = \frac{|\mathcal{S}(P, s)|}{|P|}$$

For a set of statements, \textit{Impact} is defined as the distributed union of statement impact. That is, given a set of statements $S$ from program $P$, \textit{Impact}(S, P) is defined as follows:
\[
\text{Impact}(S, P) = \frac{\left| \bigcup_{s \in S} \overline{S}(P, s) \right|}{|P|}
\] (6.3)

Note that \text{impact} defined here is different from Definition 5.1 in Section 5.4.1 (Page 111) that measures the \textit{average-relative-impact} of a set of statements (key statements) \(S\) from concept binding (or function) \(F\) of program \(P\). Here it measures the \textit{absolute-impact} of a set of statements (concept binding) from program \(P\).

For a set of statements of a concept binding \(S\), let \(S'\) be the set of statements of the refined concept binding from \(S\); \(S'\) is always a subset of \(S\); thus the value of \(\text{Impact}(S', P)\) is no larger than the value of \(\text{Impact}(S, P)\). Recall that a ‘good’ refined concept binding should have roughly the same impact as the original concept binding, in other words, the difference between \(\text{Impact}(S', P)\) and \(\text{Impact}(S, P)\) should be insignificant.

Note that if the refined concept binding is the same as the original concept binding (i.e. \(S' = S\)), then they have same \text{Impact} (i.e., \(\text{Impact}(S', P) = \text{Impact}(S, P)\)) by definition. Therefore, it is necessary to identify whether the impact is caused by the same statements or by a highly dependent subset of statements. To address this concern, the size reduction after refinement is measured to help identify what caused the impact. For a refined concept binding \(S'\) with the same impact as that of the original concept binding \(S\), the equivalence of impact is caused by a highly dependent subset statements rather than the same concept if the refined concept binding is smaller (i.e. \(\text{Impact}(S', P) = \text{Impact}(S, P)\) & \(|S'| < |S|\)).
6.6.2 Cohesion

The concept of Cohesion defined in Section 5.4.2 is employed again here to measure the inward dependence complexity of a concept binding. The cohesion of a HB-CA concept binding is no larger than that of the refined concept binding. This can be explained by the refinement approach used in the study: A refined concept binding is the intersection of forward slice on source and backward slice on sink from an HB-CA concept binding. After refinement, all statements within the binding are dependent upon both source and sink. The original concept binding may contain statement(s) not related to the source or sink, which decreases the cohesion of the binding. Therefore, the refined concept binding should have a higher cohesion than that of the original HB-CA concept binding.

6.7 Experimental Study

This section first presents the experimental set-up and initial value of the parameters used in the VRM, followed by discusses on the experimental results.

6.7.1 Experimental Set-up

The same ten programs described in Table 3.1 are used throughout the thesis. The number of HB-CA concept bindings for each program is listed in Table 4.2. Overall, there are a total of 631 HB-CA concept bindings in the ten programs used for refinement. Note that there are five programs known to be free of large dependence clusters and five programs known to contain large dependence clusters.

For each program, the SDG is built using CodeSurfer, Grammatech’s deep-structure
6.7. Experimental Study

analysis tool [53]. For each HB-CA concept binding, vertices corresponding to the source of binding and their dependences are computed to form the binding’s dependence graph.

6.7.2 The VRM Implementation

Before applying the VRM, the concept binding’s dependence graph is optimised as discussed in Section 6.5.2. The VRM is programmed using CodeSurfer’s integrated scripting language (Scheme), so that all processes of the experiment can be combined, including construction and optimisation of HB-CA concept binding’s dependence graph, identification of source and sink of binding using the VRM, and the implementation of program chopping to return refined dependence based concept binding.

Recall the initialisation of the VRM parameters, $1/n$ ($n$ is the number of vertices) is assigned as initial weights of each vertex, and the distribution ratio $p$ is set to 0.85.

6.7.3 Results and Discussion

This section compares the refined concept binding with the associated original HB-CA concept binding by size and by the two metrics, i.e. Impact and Cohesion. The experimental results are analysed statistically, to assess the statistical significance of the difference in mean Impact and Cohesion between refined concept binding and the original.
6.7.3.1 Size of Refined Concept Binding

Concept binding refinement extracts the dependence based concept binding from an HB-CA concept binding. Some statements in the original binding might be removed by the refinement. To gain an ‘initial feel’ for the effects of refinement, the size of the refined concept binding is first compared with the size of the associated HB-CA concept binding. Here, the size of refined concept binding is expressed as a percentage of the associated HB-CA concept binding size; thus the value ranges between 0 and 1, where 0 occurs when no refined concept binding is extracted and 1 when the refined concept binding is the same as the original HB-CA concept binding.

![Figure 6.8: The size of refined concept bindings over all concept bindings](image)

Figure 6.8 shows the size over all refined concept bindings, plotted in monotonically increasing order on the x-axis. The x-axis measures the number of concept bindings as a percentage of total number of concept bindings. The y-axis measures the size of the refined concept bindings.

As can be seen from the Figure 6.8, of all the concept bindings of the ten programs, about 25% of the refined concept bindings have the same size as the associated the HB-CA concept bindings (evidenced by the long flat plateau at the top of the chart); the first approximately 5% refined concept bindings’ size is 0. That is, there
are no refined dependence based concept found in 5% of the associated HB-CA concept bindings; approximately 70% of the concept bindings are reduced in size after refinement, which is shown by the middle part of slanting line.

For the 25% refined concept bindings that have the same size as associated HB-CA concept bindings, the statements within the bindings ought to contain high dependence, because all statements in the binding are dependent on both the source and the sink of the binding, according to the refinement algorithm. These concept bindings may be extracted from a dependence cluster in which all statements are dependent on each other.

![Diagram](image)

**Figure 6.9:** The number of refined concept bindings that are same as the original HB-CA concept bindings by programs

The statistical analysis shows that there are 155 concept bindings that remain unchanged. That is they are identical before and after refinement. Figure 6.9 shows the numbers of such unchanged concept binding for each program. Clearly there is no ‘unchanged’ case in program `acct` and `oracolo2`; the programs with large dependence clusters tend to have much more unchanged concept bindings than those without. Numerically, 124 unchanged concept binding in the programs with large cluster, only 31 are in the dependence-cluster free programs.
6.7. Experimental Study

For those 5% (34) of the refined concept bindings in which no refined dependence based concept is found, the statements within the original HB-CA concept binding tend to have low dependence. This occurs when the statements within the binding do not depend upon one another or there is a dependence between some statements but no dependence between the source and sink of the binding, thus the chop with respect to the source and sink could be empty.

![Average size of refined concept bindings per program](image)

**Figure 6.10:** The average size of refined concept bindings per program

Figure 6.10 presents the average size per program of refined concept binding. As mentioned before, the size of refined concept binding is expressed as a percentage of the associated HB-CA concept binding size. Overall, the average is more than 40% for all 10 programs, more than 60% in 7 of 10 programs and more than 80% in 2 of 10 programs.

Two relevant inferences about the refined concept binding can be made from the size reduction. First, the refinement approach proposed does remove statements from an HB-CA concept binding. As discussed in Section 6.2, these statements are more likely to be non-concept-dependent statements. This is important because it provides evidence that similar impact between the refined concept binding and original concept binding is not caused by the same set of statements.
6.7. Experimental Study

Secondly, the refined concept binding contains more than half of the statements of the original HB-CA concept binding and the semantic elements which are related to domain concept within the binding are much likely to be reserved after refinement. Note that as discussed in Section 6.5.1, the source and sink of the concept binding identified using the VRM to form the refined concept binding is also more likely to be the variables related to the domain concept. Thus the refined concept binding is much more likely to relate to the domain concept than the original HB-CA concept binding.

6.7.3.2 Impact

This section compares the Impact for HB-CA concept bindings and refined dependence based concept bindings. To begin with, due to a highly dependence based concept binding formed by the refinement, it should maintain a similar impact to that of the associated HB-CA concept binding.

![Graph showing the ratio of Impact(RCB, P) to Impact(CB, P) over all concept bindings.](image)

Figure 6.11: Ratio of Impact(RCB, P) to Impact(CB, P) over all concept bindings

Figure 6.11 shows the ratio of Impact(RCB, P) to Impact(CB, P) over all concept bindings.
6.7. Experimental Study

bindings. The line in the figure is the ratio plotted in monotonically increasing order on the x-axis. The x-axis measures the number of concept bindings as a percentage of all concept bindings and the y-axis measures the ratio. For example 100% of the ratio means the refined dependence based concept binding has the same impact value as the original one.

From the figure it can be seen that approximately 80% of the refined dependence based concept bindings have the same impact as the original HB-CA concept bindings.

![Figure 6.12: Average Impact value for HB-CA concept binding (CB) and refined dependence based concept binding (RCB)](image)

Figure 6.12 shows the average impact on the HB-CA concept bindings and the refined dependence based concept bindings by programs. Overall the original concept binding tends to have a slightly higher impact than it does after refinement. Apart from two cases (CADP and diffutils), the difference can be visualised in Figure 6.12. The reduction in impact caused by refinement has to be balanced against the gain in manageability that comes from the reduced size.

Wilcoxon Matched-Pairs Signed Ranks Test procedure is employed to test the sig-
Figure 6.13: Boxplot Impact value for HB-CA concept binding (CB) and refined dependence based concept binding (RCB)

Significance of the difference, where the null hypothesis is that there is no difference between the Impact of HB-CA concept binding and their refined concept binding. The results presented in Table 6.2 shows that the Impact difference is significant (P-value < .001).

Table 6.2: Wilcoxon Matched-Pairs Signed Ranks Test comparing Impact of HB-CA concept bindings with their refined concept bindings.

<table>
<thead>
<tr>
<th>No. Cases</th>
<th>Z-value</th>
<th>Asymp.Sig.(2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCB - CB</td>
<td>547</td>
<td>-12.942</td>
</tr>
</tbody>
</table>

The distribution of the Impacts, shown in Figure 6.13, demonstrates the undesirable effect of large dependence clusters [21]. In the programs without large dependence clusters, the data have a reasonable distribution and only a few outliers. By contrast, with three of the five dependence-cluster-having programs, the ‘box’, in essence, include only the median value. These programs also have a considerable number of outliers and extreme values.


### Size and Impact

The case that concept binding and its refined concept binding have the same impact could be because they are the same set of statements.

Figure 6.11 shows that approximate 80% of the refined concept bindings have the same impact as the original, but Figure 6.8 shows only of the 25% refined concept bindings have same size as the original concept binding. Thus, it is possible that 55% of the refined concept bindings have the same impact but with smaller size. If the size of a refined concept binding is significantly smaller, it is clear that our concept refinement approach can produce a dependence based concept binding that have an effective reduction in size while retaining a similar impact.

In experiment, the size difference is examined only for the refined concept bindings that have the same impact as the original concept bindings (they are considered to be same when the difference < .001). The Wilcoxon Matched-Pairs Signed Ranks Test procedure is employed to test the significance of the size difference where the null hypothesis is that there is no difference between the size of HB-CA concept binding and their refined concept binding. The results presented in Table 6.3 shows that the size difference is significant (P-value < .001).

<table>
<thead>
<tr>
<th>No. Cases</th>
<th>Z-value</th>
<th>Asymp.Sig.(2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCB - CB</td>
<td>318</td>
<td>-11.505</td>
</tr>
</tbody>
</table>

Table 6.3: Wilcoxon Matched-Pairs Signed Ranks Test comparing size of HB-CA concept bindings with their refined concept bindings.
6.7.3.3 Cohesion

This section compares the Cohesion for HB-CA concept bindings and refined dependence based concept bindings. As discussed in Section 6.6.2, the refined concept binding should have a higher Cohesion than the original concept binding.

Figure 6.14 shows the average Cohesion on the HB-CA concept bindings and their refined dependence based concept bindings (by program). Clearly refined concept bindings have higher Cohesion than the corresponding original concept bindings. The Wilcoxon Matched-Pairs Signed Ranks Test in Table 6.4 shows that the Impact difference is significant (P-value < .001).

![Figure 6.14: Average Cohesion value for CB and RCB](image)

Table 6.4: Wilcoxon Matched-Pairs Signed Ranks Test comparing Impact of HB-CA concept bindings with their refined concept bindings.

<table>
<thead>
<tr>
<th></th>
<th>No. Cases</th>
<th>Z-value</th>
<th>Asymp.Sig.(2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCB - CB</td>
<td>547</td>
<td>-16.020</td>
<td>.000</td>
</tr>
</tbody>
</table>

Unlike the Impact, the distribution of the Cohesion, shown in Figure 6.15, does
6.7. Experimental Study

not indicate a significant effect for programs with large dependence clusters. The data gathered have a reasonable distribution for all programs except that the five dependence–cluster–having programs have a considerable number of outliers and extreme values.

6.7.3.4 Summary of Results

Having presented the data gathered in the experimental study, this section concludes by returning to the four research questions:

1. Are the refined concept bindings considerably smaller than the original HB-CA concept bindings?

Figure 6.8 and Figure 6.10 show that a size reduction is achieved by concept refinement. Furthermore, for the 318 (of 631) concept bindings where the refined concept bindings retain identical impact, the result of Wilcoxon Matched-Pairs Signed Ranks Test in Table 6.3 shows the size reduction is significant.
2. *Do refined concept bindings have similar impact on the computation of the program as that of the original HB-CA concept bindings?*

Figure 6.11 shows that approximately 80% of the refined concept bindings have the same impact as the original HB-CA concept bindings. Figure 6.12 shows the difference of the average Impact per program for the concept bindings before and after refinement. Although the difference is visually small, it is statistically significant, as shown in Table 6.2.

3. *Do refined concept bindings form a more cohesive unit than those of the original HB-CA concept bindings?*

As visually apparent in Figure 6.14, the refined concept bindings have a considerably higher Cohesion than the original HB-CA concept bindings. This is also shown to be statistically significant in Table 6.4.

4. *Do refined concept bindings still relate to the domain concept of the original HB-CA concept bindings?*

The refined concept binding is a subset of statements of the original HB-CA concept binding. As discussed in Section 6.5.1 and Section 6.7.3.1, the source and sink of the concept binding are believed to be more likely to be related to the domain concept. Note that, it is not experimentally shown that the source and sink are related to the domain concept, but this seems to be a reasonable starting assumption. Because the remaining statements formed by the source and sink retain similar Impact, the refined concept bindings are much more likely to be domain-related concept bindings, assuming that the source and sink are at the first place.

5. *Do (large) dependence clusters [21] affect concept binding refinement?*

In Figure 6.10 the average size of a refined concept binding is visually larger
in the five programs that contain large dependence clusters than those in the
dependence-cluster free programs. As shown in Figure 6.9, there are 93 of 124
concept bindings taken from the five programs with large dependence clusters
remain the same after refinement.

6.8 Summary

In this chapter the approach of dependence based concept refinement is presented.
The framework of the approach is shown and the chapter presents the details of each
step of the approach. The VRM is constructed to identify the source and the sink
for a concept binding and chopping is implemented to extract dependence based
concept bindings.

The experiments are performed on the same 10 programs used previously. The
results from the experiments and the statistical analysis indicate that the VRM is
an efficient technique for computing the source and sink vertices and more than
80% of the refined the dependence based concept bindings have the same impact as
the original HB-CA concept bindings. The significance of size reduction confirms
the approach has the ability to rule out the non-concept-dependent statements.
Therefore, the resulting refined concept bindings are more accurate than the original
concept bindings, which reduces cost and effort of analysing.
Chapter 7

Conclusions And Future Work

This chapter provides a summary of the outcomes of the research. Our motivation is geared toward providing a whole framework of the possible combination of approaches for low-level program slicing-based-dependence analysis and high-level concept assignment and to refine dependence based concept assignment.

Section 7.1 explains the contribution of the work and Section 7.2 summarises the work carried out. Section 3.6 discusses some of the threats to validity of the experiments in this thesis, and future work is outlined in Section 7.3.

7.1 Summary of Thesis Achievements

Is it possible to fully automate and implement the combination of approaches?

The combination of two techniques involves two separate implementations. Therefore, full automation requires fully automated implementations for each of the tech-
niques and an automated integration of the two implementations.

For concept assignment a knowledge base is necessary. A standard knowledge library can be produced. However for each program, such a knowledge base needs to be rebuilt by programmers depending upon the specific task goals. As such there is some need of human involvement. After that the process can be implemented automatically.

Dependence analysis can be implemented automatically once the criterion is determined.

Current experiments involve two steps: (1) implement HB-CA using WeSCA which outputs the concept information to a datafile; (2) implement slicing based dependence analysis for HB-CA concepts using CodeSurfer and input the concept information from the data file.

However, both tools can be run in command line mode. Using a script language it is easy to link the two implementations.

With CodeSurfer’s scripting language, a variety of types of computations can be implemented easily. That means the HB-CA algorithm can be programmed in CodeSurfer. Thus, we believe the overall goal of a fully automated dependence based concept assignment can be achieved.

**Is concept refinement better than concept extension and concept abbreviation?**

The concept slice produced by concept extension using dependence based analysis is somehow too large, since the typical slice size upon a single criterion is about one third of the whole program [25].
7.2 Conclusions

The key statements produced by concept abbreviation using dependence based analysis can be small. However, only one third of concepts studied were found to have key statements.

The dependence based concepts produced by concept refinement using dependence based analysis are close in size to the original concepts. Concept refinement improves the accuracy of concept bindings produced by HB-CA by removing statements that are not related the concepts, while retaining the same impact and simultaneously increasing cohesion. Therefore, the refined concepts may be more suitable for program comprehension.

7.2 Conclusions

The work presented here has been an experimental study of existing concept assignment and dependence analysis combination approaches and an innovative combination of techniques introduced by the author. The existing approaches include:

- concept extension using program slicing
- concept abbreviation using program slicing based dependence analysis

It is important to mention that, although the framework of these approaches was proposed previously, the lack of experimental studies meant that their feasibility and practicability were not explored prior to this thesis. We studied executable concept slicing and key statement analysis using ten subject programs. More than 600 concept bindings were identified and studied to provide evidence of both advantages and disadvantages of these two approaches. These results also provide the foundation
for our innovative combination of techniques, which retains the advantages and overcomes the disadvantages of the two previously proposed approaches.

The novel contributions of this work include:

- the new framework for dependence-based concept assignment using slicing-based dependence analysis to refine the current HB-CA concepts;

- the Vertex Rank Model (VRM) that computes the source and sink vertices for any analysis scope in roughly linear complexity in $O(n^2)$;

- the dependence based metrics used to measure HB-CA concepts

An additional contribution is formed by the experimental evaluation of existing approaches, which hitherto, remained unevaluated. With the framework for dependence based concept assignment, we conducted experiments using the same ten programs with the same HB-CA concept bindings, in order to compare the dependence based concept assignment with executable concept slicing and key statement analysis. The results showed the dependence based concepts are more precise than the original HB-CA concepts, whose size are more reasonable for program comprehension than executable concept slice and more general than key statements.

### 7.3 Future Work

This section suggests some possible future research directions.
7.3. Future Work

7.3.1 Dependence Graph Refinement for VRM

The focus of Vertex Rank Model was confined to dependence graph, where data
dependence and control dependence were regarded as the same and the distribution
values of them are same in the VRM. Therefore, a natural extension to the research
would be to set up with different distribution value for data and control edges in
the dependence graph. In the experiment, we have optimised the dependence graph
before the VRM is applied by removing some pseudo vertices and edges. It would
be interesting to evaluate the VRM results for each kind of vertex in the dependence
graph. Furthermore, the application of different kinds of source representation graph
to VRM could also produce interesting results.

7.3.2 Combination of HB-CA, Syntactic Analysis and Dependency Analysis

Further approaches for dependence based concept could combine HB-CA concept
assignment with both syntactic and dependence analysis. Applying syntactic analy-
sis could help the segmentation of the HB-CA concepts to give more precise concept
boundaries.

7.3.3 Dependence Clusters

Further work on the effect of large dependence clusters could be implemented by
investigating the causes of large dependence clusters and removing large dependence
clusters.
Bibliography


BIBLIOGRAPHY


[74] M. Kamkar, P. Fritzson, and N. Shahmehri. Three approaches to interprocedural dynamic slicing. In EUROMICRO 93: Nineteenth EUROMICRO symposium on microprocessing and microprogramming on Open system design:


