Identifying code smells in source code and automatic removal of Long Method code smell using Structure Dependence Graph (SDG) (APS-5)

Omkarendra Tiwari
(134056003)

Under the guidance of
Prof. R. K. Joshi
Department of Computer Science,
IIT Bombay.

February 13, 2019
Abstract

This report presents *Segmentation*, a new approach based on successive edge contraction is introduced for extract method refactoring. It targets identification of distinct functionalities implemented within a method. Segmentation builds upon data and control dependencies among statements to extract functionalities from code by successive contraction of edges in the *Structure Dependence Graph* (SDG). Three edge contractions are explored, namely *structural control edge contraction*, *exclusive data dependence edge contraction*, and *sequential data dependence edge contraction*. The SDG is first constructed from the program, which is then collapsed into a segment graph that captures dependence between subtasks. An intermediate representation for data and control dependencies among statements keeps the technique language independent. The approach is evaluated on four case studies, including three from the open source domain, and the findings are reported. Also, an eclipse plug-in is developed for automatic extract-method refactoring based on proposed segmentation algorithm. Further, the use of SDG and segmentation algorithm is inspected to detect other code smells.
## Contents

1 Overview of Report .......................... 5
  1.1 APS-4 ........................................ 5
  1.2 APS-5 ........................................ 5
  1.3 Overview of the report .................... 5

2 Introduction .................................. 6

3 Structural Representation .................... 8
  3.1 Structural control edges .................... 8
    3.1.1 Segment IR .............................. 8
  3.2 Structure Dependence Graph (SDG) ............ 11
    3.2.1 Source vertex, Control dependence, Data Dependence: ....... 11
    3.2.2 Chains: .................................. 11
    3.2.3 Edge Contraction, Seg Id: .................. 11
    3.2.4 Control depth, Control region: ............... 12
  3.3 Pruning SDG ................................ 13
  3.4 Segments .................................... 13
    3.4.1 Control independence ..................... 13
    3.4.2 Data independence: ....................... 14
    3.4.3 Segments ................................ 14

4 Overview of Segmentation Approach .......... 15

5 Segmentation ................................. 19
  5.1 Control Edge Contraction (CEC) ............... 19
    5.1.1 Lack of Computational Strength (LoCS): ................. 20
    5.1.2 Parent Affinity: ................................ 21
    5.1.3 Contraction: .................................. 23
  5.2 Exclusive Source Contraction ................. 23
  5.3 Sequential Data Dependence Contraction (SDDC) .... 24
    5.3.1 Chain Structure Selection: ................ 24
  5.4 The Segmentation Algorithm .................. 25

6 Evaluation .................................. 28
  6.1 Performance metrics used for comparison ........ 28
  6.2 The Process of Tuning ..................... 29
  6.3 Application to OSS code .................... 30
## List of Figures

3.1 Example graph for chains and Edge Contraction .......................... 11  
3.2 Control region example ....................................................... 12  
3.3 Control Edge Properties and Control Independent graph ............... 14  

4.1 Intermediate Representation of Fibonacci code and corresponding SDG .......... 15  
4.2 Structure Based Refactoring using Segmentation .......................... 17  
4.3 Fibonacci Prime source code and corresponding refactored code by segmentation .. 18  
4.4 Architecture of segmentation based refactoring .......................... 18  

5.1 Non-contributing and Contributing Producers ............................ 20  
5.2 A sample SDG with nested Control Blocks ............................... 21
List of Tables

3.1 Code to Segment IR transformation ........................................ 9
3.2 Code to Segment IR transformation ........................................ 10

5.1 Analysis of Control Block 9 in Figure 5.2 .............................. 22

6.1 Evaluation Case Studies ....................................................... 28
6.2 Precision and Recall obtained after tuning ............................ 30
6.3 Tuning segmentation for best results ................................. 30
6.4 Extract method opportunities for all methods ...................... 31
6.5 Comparison for JHotDraw .................................................. 32
6.6 Comparison for XData grading system ................................. 32
6.7 Comparison for JUnit ....................................................... 32
6.8 Performance over long methods (LOC>150) ......................... 32
Chapter 1

Overview of Report

This chapter presents the overview of the work done in APS-4 and overview of the work included in APS-5.

1.1 APS-4

We developed and evolved a graph called structure dependence graph to better fit for representing a source code for the purpose of static analysis and proposed an algorithm, segmentation, to identify the long method code smell and corresponding extract method opportunities(EMO). To prevent the user from choosing a sound EMO from the suggested functionalities a quality measure is introduced. Finally, the approach is applied over two case-studies and performance is measured.

1.2 APS-5

1.3 Overview of the report

The remainder of this report is structured as follows. Introduction of the extension to segmentation and SDG is presented in Chapter 1. Previous work is briefed in Chapter 2. Chapter 3 contains the details of the extension introduced in segmentation. Extension of SDG for inter-procedural code and its use in solving 'feature envy' and 'long parameter list' code smell is detailed in Chapter 4. Lastly a case study is analyzed.
Chapter 2

Introduction

The design quality of software can be improved by restructuring. As noted by Arnold [1], it not only helps improve comprehension and subsequent changes but also helps in increasing the value of software. Though manual restructuring by expert designers may generate a high quality of structure, as noted by Griswold and Notkin [2], for large systems, it can take a long time with a rise in error risk and cost. Therefore, an automation of the restructuring process becomes highly desirable.

Modern restructuring approaches focus on design aspects keeping in view the traditional wisdom of cohesion and coupling. The terms refactoring [3] and restructuring [4] are commonly used to refer to such design based structural enhancements respectively in object-oriented programs and procedural contexts. Fowler [5] described several kinds of code smells and the related refactorings. One of them is the long method code smell, for which extract method is known to be a suitable refactoring solution. Kim et al. [6] in their investigation of refactoring practices report that readability, maintainability, reuse, reduction in code duplication and bugs, and ease of adding new features are among the primary benefits of refactoring. Murphy-Hill et al. [7] analyzed refactoring practices and observed that extract method refactoring tools are seldom used, and instead, such refactoring is carried out manually.

In this paper, a novel graph-based approach for identifying and ranking extract method opportunities is developed and demonstrated. The approach is called segmentation, in which, extract method opportunities form segments. The focus of the approach is on identifying and suggesting segments that are of manageable size, are cohesive and functionally sound, and are possibly reusable with little or no modification. An important aspect of refactoring activity is the role of human involvement in automation of the process, while maximizing its benefits. For example, some approaches [8] require human support in identifying seed statements or the variable of interest, whereas, some others [9], [10] require human support in the selection of precise extract method opportunities. The first one requires human involvement in initial phase of seeding, while the latter uses involvement in the final phase of selection. Our approach automatically suggests extract method opportunities, going closer to full automation, with manual intervention minimized to the final stage on need basis. Segmentation automatically ranks and picks up only a subset of all identified segments. In order to make the refactored code acceptable at the user end, the technique also facilitates human intervention to override a chosen segment by another suggested variant of it by enlisting the alternatives.

The source code to be refactored is represented in the form of a graph called Structure Dependence Graph (SDG). SDG is a variant of the well-known Program Dependence Graph (PDG) structure [11], but it differs in the representation of control dependence among statements and
blocks. Control edges present in SDG reflect the hierarchal control dependence structure of the code, as opposed to the non-hierarchal structure in PDGs. An intermediate code representation (IR) is designed primarily to bring out the necessary structural information from the source code. It also helps in mapping the source code to SDG. The IR is referred to as segment IR. It helps to keep the semi-automatic refactoring technique programming-language independent.

Segmentation transforms the SDG into a reduced version of it called segment graph. This reduction process consists of two stages, (i) identification of characteristic structures, and (ii) application of corresponding transformations on them using edge contraction. A vertex of a segment graph is intended to represent a cluster of statements implementing a distinct functionality. The proposed segmentation algorithm suggests extract method opportunities from segment graphs. One of the challenges in segmentation is handling of overlapping among possible opportunities under consideration. We use two affinity metrics to resolve the issue of overlapping.

The approach is validated through the following four case studies. Firstly, it is applied to a semi-modular version of an implementation of this segmentation approach itself. The experimental semi-modular source code contained about 350 lines of code in C, which included methods of varying sizes that were synthetically created by unfolding method bodies in the original code. From this synthetically created non-modular code, our technique was more or less able to identify the original methods. Three open source case studies were used for validating the approach. Among them, the JUnit and the JHotDraw are two Java-based programs of Silva et al. [12]. These are the versions that they created by inlining callee methods into the caller methods. Both the case studies have earlier been used as benchmark in the works of Silva et al. [12] and Charalamidou et al. [9]. The third open source program that we used is the XData system [13], which is used to assist in grading answers that are SQL queries in a learning environment.
Chapter 3

Structural Representation

This section describes representations used in the segmentation approach. These include an intermediate representation (Segment IR) for representing programming constructs and a graph representation for data and control dependencies (SDG).

3.1 Structural control edges

Structural control edges are introduced to capture the structural aspects of control in a program under refactoring. They represent the structure of control blocks in terms of control dependencies with their internal statements. Structural control edges are different from control edges of control flow-graphs [14], in that the former show structural control dependence as opposed to behavioral control dependence in control flow graphs. Structural control edges are used in a data structure called Structure Dependence Graph (SDG). Structural control edges result from structural control blocks such as if-else, for, and Do-case. These structures are discussed in Section 3.1.1. Hereafter, the term control edge is used to refer to structural control edges.

3.1.1 Segment IR

To support language independent refactoring, an intermediate representation called Segment IR is defined. The transformation of input code to segment IR aims at minimizing the number of tokens used in the representation of code, at the same time preserving data and control dependence among the statements. Segment IR is the link between the SDG and the source code. And thus, makes the visualization of mapping the source code to graph easy and vice versa. The IR requires only two kinds of information from any statement of the input code, the kind of the operation being performed, and a list of variables and their roles in execution of the operation. The operations are classified into ten kinds, namely, Input, Output, Assign, If, Else, Elseif, DoCase, Case, Loop and Invar.

To represent conditional or iterative block statements, the segment IR encodes the blocks by marking the sizes of the block instead of using block markers. Actual operators are not captured, since we are interested in dependency of data and control for segmentation. The set of mappings for transformation of statements from input code to Segment IR is shown in Table 3.1 through examples. The table also shows the corresponding SDG models (SDGs are explained in Section 3.2). The kinds of operations included in the IR are explained below.

- **Input/Output**: In segment IR, an input statement such as scanf is represented by the keyword Input, which is followed by the list of variables participating in the input operation. Input
<table>
<thead>
<tr>
<th>Kind</th>
<th>Example Code</th>
<th>Equivalent Segment IR</th>
<th>SDG Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td><code>scanf(&quot;%d %d&quot;, &amp;n1, &amp;n2);</code></td>
<td>0. <code>input n1</code></td>
<td><img src="image1" alt="Input SDG Model" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. <code>input n2</code></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td><code>printf(&quot;%d %d&quot;, n1, n2);</code></td>
<td>0. <code>output n1</code></td>
<td><img src="image2" alt="Output SDG Model" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. <code>output n2</code></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td><code>x = a + b</code></td>
<td>0. <code>assign x a b</code></td>
<td><img src="image3" alt="Assignment SDG Model" /></td>
</tr>
<tr>
<td>If</td>
<td><code>if(x &lt; (y+10)) x++;</code></td>
<td>0. <code>if x y 1</code></td>
<td><img src="image4" alt="If SDG Model" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. <code>assign x x</code></td>
<td></td>
</tr>
<tr>
<td>IF-Else</td>
<td><code>if (x) y = x;</code></td>
<td>0. <code>if x 2</code></td>
<td><img src="image5" alt="IF-Else SDG Model" /></td>
</tr>
<tr>
<td></td>
<td><code>else x =y;</code></td>
<td>1. <code>assign y x</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. <code>else 1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. <code>assign x y</code></td>
<td></td>
</tr>
<tr>
<td>If-ElseIf-Else</td>
<td><code>if (x &gt; y) v = x;</code></td>
<td>0. <code>if x y 2</code></td>
<td><img src="image6" alt="If-ElseIf-Else SDG Model" /></td>
</tr>
<tr>
<td></td>
<td><code>else if(y &gt; z)</code></td>
<td>1. <code>assign v x</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>v =y;</code></td>
<td>2. <code>elseif y z 2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>else v = z;</code></td>
<td>3. <code>assign v y</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. <code>else 1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. <code>assign v z</code></td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td><code>for (i=1;i&lt;=n;i++) f = f *i;</code></td>
<td>0. <code>assign i</code></td>
<td><img src="image7" alt="Loop SDG Model" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. <code>loop i n 2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. <code>assign f f i</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. <code>assign i i</code></td>
<td></td>
</tr>
<tr>
<td>Reduced Loop</td>
<td><code>for (i=1;i&lt;=n;i++) f = f *i;</code></td>
<td>0. <code>assign i</code></td>
<td><img src="image8" alt="Reduced Loop SDG Model" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. <code>loop i n 1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. <code>assign f f i</code></td>
<td></td>
</tr>
<tr>
<td>DoCase</td>
<td><code>switch(x){</code></td>
<td>0. <code>docase x 1</code></td>
<td><img src="image9" alt="DoCase SDG Model" /></td>
</tr>
<tr>
<td></td>
<td>case 1: <code>s =0;</code></td>
<td>1. <code>case 3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>break;</td>
<td>2. <code>assign s</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>default: <code>printf(&quot;Default&quot;);</code></td>
<td>3. <code>break</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>4. <code>case 1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. <code>invar</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 3.2: Code to Segment IR transformation

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Equivalent Segment IR</th>
<th>SDG Model</th>
<th>Pruned SDG Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-loop &amp;</td>
<td>0. assign i</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>Backward edge</td>
<td>1. loop i num 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. assign fact fact i</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. assign i i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple edges</td>
<td>0. loop i num 1</td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>1. assign fact fact i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

is a special case of assign. Similarly, an output statement such as printf is represented by the keyword Output, which is followed by the list of variables participating in the output operation. If a single input or output function reads/writes values of multiple variables, those variables can be listed in multiple input/output statements, which helps the algorithm exercise finer control in identifying segments. For example, in the table, two inputs in scanf statement are split in two statements in the segment IR. The same can be observed for printf statement in the table. However, such separation of variables is not always required, as in the case of file input/output, in which a file-pointer cannot be separated from program variables used for file data. Such group of variables that should go together can be listed together. The algorithm does not split them.

- **assign**: An assignment statement is represented by keyword assign, which is followed by a list of variables participating in the assignment operation. The first variable in the list represents the l-value variable that is being modified and rest of the variables in the list are variables accessed as r-values. The table shows an example of this case with two inputs and a result.

- **If, Else and Elseif**: The If follows a list of variables and an integer. The list contains all the variables participating in decision regarding execution of the block. An integer defines the block scope as the count of immediately following statements dependent on it (block length). Block lengths are also noted for the else and the elseif blocks. Separate examples are provided in the table for the three statements.

- **Loop**: The iterative statements are represented in two forms. In the first form, three components initialization, condition checking, and increment/decrement are represented separately. Whereas, the reduced loop representation does not consider the loop step operations. This may possibly reduce the number of edge contractions in the algorithm resulting in coarser segments. Both kinds are represented by the keyword Loop followed a list of variables and an integer. The list contains all the variables participating in the decision making and the integer defines the scope, that is the number of dependent statements. The example for both are shown in the table.

- **DoCase**: The DoCase keyword is followed by a variable. The case statements and the default statement are represented by the keyword Case. It is noted that no special keyword is required
Invar: It is a statement that does not use a variable. For example, print statement which uses constants is an Invar statement. An instance of such a statement is in the last row of Table 3.1.

3.2 Structure Dependence Graph (SDG)

The structure dependence graph $G = (V_G, E_G)$ is a directed graph, where $V_G$ is a set of vertices and $E_G$ is a set of labeled edges. Each vertex in the graph corresponds to exactly one statement of the segment IR. The label of a vertex is an integer representing its position in segment IR or the statement count. Statement count for the first statement in segment IR is 0. Properties related to SDG are now discussed below.

3.2.1 Source vertex, Control dependence, Data Dependence:

A source vertex is a vertex with no incoming data edges and at least one outgoing data edge. There can be more than one source vertices in a SDG. A source vertex in SDG captures an input statement or definition of a variable with constant in the program. An edge $⟨u, v⟩$ labeled $C$ represents control dependence relation between the statements represented by vertex $u$ and $v$. Similarly, label $D$ on edge $⟨u, v⟩$ represents data dependence between statements represented by end vertices, denoted as head $u$ and tail $v$.

3.2.2 Chains:

They define one type of collapsible structure in our model. It is a directed path $u_0 \rightarrow u_1 \rightarrow ...u_{k-1} \rightarrow u_k$ consisting of vertices and data edges in the SDG such that each vertex except $u_0$ has exactly one predecessor, and each vertex except $u_k$ has exactly one successor. The vertices $u_0, u_k$ are called head, tail of the chain respectively. Such a structure captures sequential data dependence in the program and may correspond to a subtask. Length of the chain is measured by the total number of edges present in it. In Figure 3.1(a), paths $\{2 \rightarrow 3\}, \{4 \rightarrow 5\}, \{6 \rightarrow 7\}$ are chains, while paths $\{0 \rightarrow 2 \rightarrow 3\}, \{3 \rightarrow 4 \rightarrow 5\}$ are not.

3.2.3 Edge Contraction, Seg Id:

Edge contraction is an operation which involves removal of an edge and merging of its two end vertices. As a result a new vertex is created. All the edges incident on either of the end vertices of the contracted edge are mapped to a single new vertex. After edge contraction, the resulting
vertex is assigned a new label called as \textit{segment id} (or seg id). For an edge \(\langle u, v \rangle \in E_G\), which is being contracted, the seg id is identified as follows:

1. \(u\) is seg id if \(\langle u, v \rangle\) is a control edge, else
2. \(v\) is seg id vertex if \(u\) is a source vertex, else
3. \(u\) is seg id if \(\langle u, v \rangle\) is a part of \textit{chain}

An example transformation of an SDG by a sequence of edge contractions (of colored edges) is shown through Figures 3.1 (a), (b), (c) and (d). Colored edges show the edges to be contracted, and colored vertices show the \textit{seg ids} for the resulting vertices.

### 3.2.4 Control depth, Control region:

The \textit{control depth} of a vertex \(v\) from a source vertex \(u\) is the number of control edges present in the path from \(u\) to \(v\). For example, in Figure 4.1, control depth of vertex 6 and 7 from the source vertex 1 is two. The \textit{control region} of a vertex reflects the control block to which a vertex belongs. Control region of a vertex \(v\) the vertex-id of the vertex on which it is control dependent. It is set to -1, if it is not control dependent on any other vertex. In other words, if a vertex \(v\) has an incoming control edge (edge with label 'c') from some vertex \(u\) then its control region is 'u', and it is -1 otherwise. Vertices in same color in Figure 4.1 share the same control region. For example, vertices 4 and 5 are in same region 3, whereas 9, 10, 11, and 12 are in region 8.

```
int ctrlRegionDemo ()
{
    int n1, n2=0;
    scanf("%d",&n1);
    if (n2!=1)
    {
        scanf("%d",&n2);
        printf("%d",n1%n2);
    }
}
```

(a) Source Code

```
0. assign n2
1. input n1
2. if n2 2
3. input n2
4. output n1 n2
```

(b) Segment IR

```
(int) SDG
```

(c) SDG

Figure 3.2: Control region example

The concept of control region permits detection of data edge head vertices that can be merged into tail vertices without altering program behavior. Edge contraction is not permitted when head and tail vertices of a data edge do not belong to the same control region. In Figure 3.2, a source code, it's segment IR and SDG are shown in (a), (b) and (c) respectively. Statement ids in (b) correspond to vertex ids in (c). As shown in (c), vertices 3 and 4 are in the same control region 2, whereas, vertices 0, 1 and 2 are under control region -1, they being not control dependent on another vertex. Vertex 4 has an incoming exclusive source data edge from vertex 1, indicating a closed data association between the two. But vertex 1 can’t be merged into vertex 4, without moving the corresponding statement into the if-block denoted by control region 2, since it would change the program’s behavior. On the contrary, vertex 3 can be merged into vertex 4 as they are in the same control region 2, and hence the exclusive data edge to vertex 4 from vertex 3 can be contracted.
3.3 Pruning SDG

Dotted edges in SDG models (as in Table 3.1) represent self loops, multiple edges and backward edges. They are shown in the figure for the sake of completeness, and they can be removed by pruning. Presence of such edges does not refine the solution, but it increases the number of edge contraction operations. The prunable edges are discussed below.

**Self loop:** Statements like `i++` cause a data edge in form of self loop. A self loop does not help in clustering as it starts and ends at the same vertex. So, removal of a self loop from SDG does not affect the formulation of segments. For example, in first row of Table 3.2, vertex 2 and 3 have self-loops which are pruned.

**Multiple edges:** If there exists more than one edge from a vertex $u$ to a vertex $v$ then all the edges can be removed except one. If all such edges are of the same type (have the same label), only one edge of that type is sufficient to preserve the dependence relation. In the case of presence of both control and data edges, the control edge is given a higher priority over the data edge. For example, vertices 0 and 1 share multiple edges sourced from vertex 0 as shown in the second row of the table.

**Backward Edge:** Inside a control block, any backward edge from a vertex to its sibling and parent can be removed. This pattern is demonstrated in Table 3.2, where both backward edges from vertex 3 in the loop are removed (along with one self-loop edge). Since, the vertices joined by these edges already share a control edge from their parent vertex, removal of such edges does not affect the resulting segment.

3.4 Segments

The refactoring process requires structural enhancement, which can be done by binding closely related statements into separate units and thus lowering their interdependency. Now, we discuss the approach to extract such modular units as **segments** from the SDG graph, which in turn is extracted from the IR corresponding to input code.

3.4.1 Control independence

**Primary and Secondary Control Vertices:** A vertex representing one of If, Loop, or DoCase statements is called *primary control vertex*. A vertex representing one of Else, Elseif, Case, break, or continue statements is called a *secondary control vertex*, since such control statements are control dependent on a primary control vertex.

Before the definition of control independence the following properties of SDG are noted:

1. A secondary control vertex can be direct predecessor of a control vertex only if they share at least one primary control vertex as a predecessor.

2. A secondary control vertex can have exactly one primary control vertex as direct predecessor.

Figure 3.3(a) provides examples of valid and invalid edges w.r.t. the above properties. Edges $\langle s_2, s_1 \rangle$ and $\langle s_1, s_3 \rangle$ are invalid since they violate rule 1, whereas, edge $\langle p_1, s_2 \rangle$ violates rule 2. A syntactically correct program can not produce such cases.

Let $G = (V_G, E_G)$ be the SDG, and $C = (V_C, E_C)$ be a weakly connected subgraph (i.e., connected not withholding the directions) of $G$. Graph $C$ is *control independent* if, (i) no vertex in $V_{G-C}$ has an incident control edge from a vertex in $V_C$ (ii) no vertex except a primary control vertex in $V_C$ has an incident control edge from a vertex in $V_{G-C}$, and (iii) every secondary control...
vertex in C has at least one predecessor in C which is a primary control vertex. For example, in Figure 4.1, weakly connected subgraph $C = (V,E)$, where $V = \{21,22\}$ and $E = \{(21,22)\}$, is not control independent as primary control vertex for vertex 21 is not present in C.

Figure 3.3(b) provides examples of valid and invalid edges for subgraph C of SDG G to be control independent. Crossed edges (also shown in red) are prohibited.

### 3.4.2 Data independence:

Let $G=(V_G,E_G)$ be the SDG, $D=(V_D,E_D)$ be a weakly connected subgraph of $G$. Let vertices $u, v \in V_D$, $w \in V_{G-D}$. Graph D is data independent if either data edges $\langle u, v \rangle, \langle u, w \rangle \notin E_G$ or $u, w$ has different control region. Weakly connected subgraph $D = (V,E)$, $V = \{8,9,10,11,12,13\}$, $E = \{(8,9), (8,10), (8,11), (8,12), (9,11), (11,13)\}$ shown in Figure 4.1 is data independent.

### 3.4.3 Segments

A segment is a subgraph S of SDG G, which is both control independent and data independent in G. Vertices present in a segment can be classified in the following three categories.

1. **Producer** Often a segment contains statements providing data to other statements in the segment to accomplish their tasks. Such statements are called producer statements. In a segment, such statements are represented by vertices which share data edges with other vertices.
2. **Consumer** A set of statements involved in computations of intermediate and final results are consumer statements. These vertices are data dependent directly or indirectly on producer vertices.
3. **Relays** A set of statements communicating computed results with other segments are called relays. Thus, these are also producer vertices.

It can be observed that relay statements in a segment contain the final result of the segment computation. Thus, a relay statement can also be a consumer statement. Desirable and undesirable segments can be characterized in terms of producers, consumers, and relays. For example, segments having more than one relay statements may not be desirable.
Chapter 4

Overview of Segmentation Approach

This section provides an overview of the process of refactoring using the segmentation approach through an example. The segmentation algorithm is developed in the next section. A program for finding Fibonacci Primes is used in the example. Figure 4.3 shows the input code and the refactored code after applying segmentation. Intermediate representation for the input source code and its corresponding structure dependence graph (the SDG, defined in Section 3.2) is shown in Figure 4.1. Transformation of this SDG to a graph called segment graph is achieved by a process of successive edge contraction as illustrated below.

Segment IR:
0. invar
1. input n
2. assign a
3. if n 2
4. output a
5. else 3
6. assign b
7. assign i
8. loop i 4
9. assign t a b
10. assign a b
11. assign b t
12. assign i i
13. output b
14. assign i
15. loop i b 2
16. if b i 1
17. break
18. assign i i
19. if b i 2
20. invar
21. else 1
22. invar

Figure 4.1: Intermediate Representation of Fibonacci code and corresponding SDG
The SDG shown in Figure 4.1 contains vertices and edges in multiple colors. In this figure, vertices represent statements. Edges are labeled D and C, representing data and control dependence respectively. A set of statements represented by a distinct color (non-white) form a control block, and it can be seen that they are all dependent on a common control vertex. For example, vertices {4,5} and {6,7,8} form two separate control blocks that are control dependent on control vertices 3 (if-block) and 5 (else-block) respectively. Blocks can be nested.

Statements corresponding to each such block can be found at the same indentation level in the segment IR shown on the right. Vertex labels represent the statement numbers in the IR. A segment IR includes constructs such as conditional, iterative and input/output statements of input code, which are required for the analysis.

The next step in the refactoring process is to investigate control blocks in bottom up order for identification of distinct functionality. A block containing significant functionality is collapsed to a single vertex by contracting all the control edges present in the block. Further, the newly collapsed block is investigated for a possible merger with statements that supply data to it.

In our example, the first control vertex that is analyzed is vertex 19 (if block). The block at 19 has two control dependent statement 20 and 21. Vertex 21 represents a control block at else-statement and consists of one control dependent statement 22. It can be seen that block at 19 has no outgoing data-edge connecting to any vertex outside of the block (i.e. after vertex 22). Thus, block 19 is not producing any result usable by rest of the code. Hence, it is not considered as an extraction opportunity, and it is collapsed (contracted) into vertex 19 as shown in Figure 4.2(a).

Control block rooted at vertex 16 (if-block) is analyzed next. This block does not define (i.e. assigns value to a variable) any variable. Hence, it is not inspected for extraction. The block is not contracted because it is control dependent on vertex 15. Control block rooted at vertex 15 (iterative block) is analyzed next. It can be seen that it has an outgoing data edge from vertex 18. Hence, this block is further examined for assessing its extraction possibility. Lack of computational strength measure for this block happens to be 0.5, that is higher than a default threshold of 0.41. So, this block is not selected for extraction. As this block is not control dependent on any other block, it is contracted as shown in Figure 4.2 (b). Now, an exclusive data dependency on the contracted block can be observed in form of a data edge ⟨14,15⟩. Such a dependency is identified by source vertex and such a data dependency between two vertices shows strong association between them. Hence, as the next step, vertex 14 is merged with vertex 15 by contracting edge ⟨14,15⟩ as shown in Figure 4.2 (c).

Such contractions on the SDG are applied whenever there is computational strength, exclusive data dependence, and also sequential data dependence. Figures 4.2 (d)-(h) trace the subsequent contraction sequence in bottom up manner. The final graph in Figure 4.2(h) is called the segment graph, which provides a solution for refactoring in terms of functions.

Figure 4.3 shows the refactored code as per suggestions produced by the segmentation approach. The segment suggestions as produced by the algorithm can be used in a computer assisted refactoring process. Segmentation algorithm can also be tuned for a customized extraction based on application context.

A high level view of the segmentation approach is shown in BPMN notation in Figure 4.4. The Figure shows various phases of graph transformations. The first phase transforms the input code to segment IR using an IR generator tool. The segment IR is then used to generate the base SDG, which is used as input to the segmentation algorithm. Segmentation results in a segment graph, which provides a solution for refactoring in terms of functions.
(a) Contracting Block at '19'
(b) Contracting Block at '15'
(c) Exclusive Source Contraction at '15'
(d) Contracting Block at '8'
(e) Exclusive Source Contraction at '8'
(f) Contracting Block at '5' and '3'
(g) Exclusive Source Contraction and extraction of '3'
(h) Chain contraction and extraction of '3'

Figure 4.2: Structure Based Refactoring using Segmentation
Input Code:

```c
void FiboPrime() {
    int i, n, a, b, t;
    printf("Enter value of n (>0)\n");
    scanf("%d", &n);
    a = 0;
    if (n == 1)
        printf("Fibo Term is %d\n", a);
    else {
        b = 1;
        for (i = 3; i <= n; i++){
            t = a + b;
            a = b;
            b = t;
        }
        printf("Fibo Term is %d\n", b);
    }
    for (i = 2; i <= b/2; i++){
        if (b%i == 0)
            break;
    }
    if (b<=1 || i <= b/2)
        printf("Not Prime\n");
    else
        printf("Prime\n");
}
```

Refactored Code:

```c
int FiboPrime3() {
    int i, a, b, n, t;
    scanf("%d", &n);
    a = 0;
    if (n == 1)
        printf("Fibo Term is %d\n", a);
    else {
        b = 1;
        for (i = 3; i <= n; i++){
            t = a + b;
            a = b;
            b = t;
        }
        printf("Fibo Term is %d\n", b);
        return b;
    }
}
void FiboPrime() {
    int i, b;
    printf("Enter value of n (>0)\n");
    b = FiboPrime3();
    for (i = 2; i <= b/2; i++){
        if (b%i == 0)
            break;
    }
    if (b<=1 || i <= b/2)
        printf("Not Prime\n");
    else
        printf("Prime\n");
}
```

Figure 4.3: Fibonacci Prime source code and corresponding refactored code by segmentation

Figure 4.4: Architecture of segmentation based refactoring
Chapter 5

Segmentation

The segmentation algorithm is now developed in this section in terms of the concept and terms introduced in the previous sections. Segmentation is a transformation of the SDG into a Segment Graph. Each vertex of the segment graph represents a segment, which is a candidate function. The segmentation approach exploits both data and control dependency to cluster statements into segments. Segmentation is a process of graph shrinking.

The SDG is shrunk into a segment graph by repeatedly carrying out three activities enumerated below on each control block vertex in SDG. Each activity shrinks the SDG by means of contraction of specific edges. Contraction of an edge results in merging of two vertices, creating a new vertex. The new vertex is assigned the label as that of the segment Id (seg id, Section 3.2.3). As a result, a bigger subgraph eventually gets reduced into a vertex. Each of the three activities of segmentation is separately dealt with in the subsections that follow.

1. **Control Edge Contraction:** This activity is carried out on control blocks or inner control blocks in the program. Every such block is contracted to a single vertex by contracting each of the associated control edges. After contraction, such a block is further investigated for merger through associated data dependencies in next two steps. With each merger the corresponding control block expands and the SDG shrinks.

2. **Exclusive Source Contraction:** It identifies statements providing data exclusively to a control block and merges them into the block.

3. **Sequential Data Dependence Contraction:** Here, we identify chains associated with block resulted from step 2 (Chains are defined in Section 3.2.2). Those chain structures that are functionally coherent with the task implemented by a block are merged into it.

### 5.1 Control Edge Contraction (CEC)

This section presents a bottom up approach, which explores control blocks for contraction in reverse order of their appearance in the code. Such an approach is effective in extracting inner segments from the code containing multilevel nested control blocks. Processing the blocks in reverse order of appearance (bottom-up in source code, and inner first in nesting) makes it possible to first focus on the core (inner blocks) of the functionality, and then to expand it by including supporting subtasks such as data supplier (e.g. initialization) and chained activities.

The CEC activity aims at identifying and grouping statements (edge contraction) of a control block, which contribute to a distinct functionality. Such a functionally distinct control block is
identified the help of metrics lack of computational strength (LoCS) and parent affinity (PA). LoCS is applied to a control block to investigate if it contains a distinct functionality. Once a control block is qualified for extraction using LoCS metric, it is also investigated for merger with its parent control block, using PA. These metrics are defined below.

5.1.1 Lack of Computational Strength (LoCS):

Metric LoCS is used to rank control blocks considered by the CEC method. It measures the computational strength of a given control block in terms of the count of producer vertices that directly or indirectly contribute to relay vertices of the block. It may be noted, a producer vertex represents exchange of data among statements which is a crucial factor in identifying distinct functionality.

It can be noted that a producer vertex may not always contribute to a relay vertex, though it may drive a control block. For example, as shown in the code given in Figure 5.1, loop variable i drives the loop, but is not used in the computation of the result. Hence, in this example, producer vertex 1 does not contribute to relay vertex 4. However, vertices 0 and 3 are producers, which contribute to relay vertex 4. The formulation for LoCS is given below.

```
int Sum()
{
    int a, sum=0, i;
    for (i=0; i<5; i++)
    {
        scanf("%d",&a);
        sum += a;
    }
    printf("sum=%d\n",sum);
}
(a) Source Code
```

```
The formulation for LoCS is given below.
```

```
(b) Segment IR
```

```
(c) SDG
```

Figure 5.1: Non-contributing and Contributing Producers

Let $b_v$ be a block rooted at a primary control vertex $v$ in SDG $G$. Block $b_v$ is identified by CEC, and is a candidate opportunity. Let $Relay(b_v, G)$ be a set of all relay vertices in $b_v$. Boolean expression $PathD(u, r, b_v, G)$ indicates if there exists a path composed of data edges from vertex $u$ to vertex $r$ in $b_v$. Function $Producer(b_v, G)$ returns the set of all producer vertices in $b_v$. Function $RelayShare(b_v, r, G)$ provides a set of all producer vertices in $b_v$ from each of which, at least one relay vertex is reachable through a path composed only of data edges. In other words, $RelayShare(b_v, r, G) = \{ u : u \in Producer(b_v, G), \exists r \in Relay(b_v, G) \land PathD(u, r, b_v, G) \}$.

Even if a producer is not a member of $RelayShare(b_v, G)$, it is a part of computational unit of the block. The set of producer vertices that are not members of any $RelayShare$ is represented by function $NonRelayShare(b_v, G)$. Count $\#TotalRelayShare$ represents the aggregate count of all the data supplies to relay vertices. Data supplies are data edges, not supplier vertices, since every supply is independently counted. $Relay(b_v, G)$ represents the set of relay statements in $b_v$. Count $\#Relay$ is the cardinality of set $Relay(b_v, G)$.

Lack of Computational Strength (LoCS) is computed as the ratio of number of relay vertices in a block to the aggregate computational strength of the block. The denominator is the count
of total supplies associated (both internal and external) with the block. It is computed as the sum of \( \# \text{NonRelayShare} \) and \( \# \text{TotalRelayShare} \). Also, \( \# \text{TotalRelayShare} \) represents the count of total sharing from the block to its outer context. Heavy sharing implies higher computational strength, which indicates lower values of LoCS. The same argument applies to \( \text{NonRelayShare} \). These quantifications are listed in the box below.

\[
\begin{align*}
\text{AllRelayShare}(b_v, G) &= \bigcup_{r \in \text{Relay}(b_v, G)} \text{RelayShare}(b_v, r, G) \\
\text{NonRelayShare}(b_v, G) &= \text{Procedure}(b_v, G) \setminus \text{AllRelayShare}(b_v, G) \\
\# \text{TotalRelayShare}(b_v, G) &= \sum_{r \in \text{Relay}(b_v, G)} |\text{RelayShare}(b_v, r, G)| \\
\# \text{Relay} &= |\text{Relay}(b_v, G)| \\
\text{LoCS}(b_v, G) &= \frac{\# \text{Relay}}{\# \text{TotalRelayShare} + \# \text{NonRelayShare}}
\end{align*}
\]

The lower the value of LoCS, the better is the possibility of extracting block \( b_v \) as a separate function. A control block is accepted as a segment only if (i) value of \( \text{LoCS}(b_v, G) \) is less than a threshold (we use 0.41 as threshold which is obtained from experimentation), and (ii) its parent control block (if it exists) contains a significant separate computation. Such computations may share data with inner control block (i.e. \( b_v \)) in the form of parameters or return value.

**Example Use of LoCS for CEC:** An example SDG is shown in Figure 5.2(a). It contains three nested control blocks. In the figure dotted arrows represent control edges, and thick arrows represent data edges. It can be seen that the outermost block is rooted at vertex 1, middle block is rooted at vertex 3, and the innermost block is rooted at vertex 9. Table 5.1 lists the attributes associated with control block 9. Since the block has at least one relay statement (vertex 13), it is considered as a candidate for extract method opportunity and its LoCS is computed. As the table shows, block 9 has four vertices with one or more outgoing data edges (10-13), one exclusive source vertex (vertex 4) and one relay vertex (vertex 13). Therefore, LoCS for the block is \( \frac{1}{5} \), which is below 0.41, the default threshold used for this example. Hence, the block is contracted, and further, the exclusive source vertex and the outgoing chain (consisting of vertices 13, 15, 16) are merged to it. Figure 5.2(b) shows the SDG after contraction of the block at vertex 9 (consisting of vertices 4 and 9 to 16). After contraction of this control block, we need to investigate the parent block at 3 as per the bottom up approach of CEC. The contraction of parent block is explained next.

### 5.1.2 Parent Affinity:

It measures the functional difference between two nested overlapping control blocks, where the inner control block is identified as a candidate opportunity, leading to its contraction into a single vertex. It is used to determine whether the inner control block should be merged with the parent
Table 5.1: Analysis of Control Block 9 in Figure 5.2

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Vertex Set</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relay Nodes</td>
<td>{13}</td>
<td>1</td>
</tr>
<tr>
<td>Sink Nodes</td>
<td>{14}</td>
<td>1</td>
</tr>
<tr>
<td>Exc. Source</td>
<td>{4}</td>
<td>1</td>
</tr>
<tr>
<td>Producer Nodes</td>
<td>{4, 10, 11, 12, 13}</td>
<td>5</td>
</tr>
<tr>
<td>RelayShare</td>
<td>{10, 11, 12}</td>
<td>3</td>
</tr>
<tr>
<td>NonRelayShare</td>
<td>{4, 13}</td>
<td>2</td>
</tr>
<tr>
<td>Incoming Chains</td>
<td>{(5, 6), (6, 10)} &amp; {(7, 8), (8, 11)}</td>
<td>2</td>
</tr>
<tr>
<td>Outgoing Chain</td>
<td>{(13, 15), (15, 16)}</td>
<td>1</td>
</tr>
</tbody>
</table>

control block and be extracted together as a single functionality.

For this purpose, the distinct functionality implemented in the parent block is quantified in terms of distinct computations (computations which are not associated with inner control block via data exchange). Distinct computations are computations which do not supply or consume data to/from inner control block. Based on the degree of coherence of computations the inner block with the parent block, four cases of merger possibility arise. These are enumerated below. It can be observed that in some of them, the decision is evident due zero overlapping. But in the case of overlapping, we use this metric to decide the possibility of merger of the inner block into its parent block. The decisions taken in each of these four cases are as follows: (i) Parent block has a distinct relay statement, and all its computations are connected with inner control block: In this case they can be merged into one block due to its close association. (ii) Parent block has a distinct relay statement, and not all or none of its computations are connected with inner control block: In this case, their merger possibility is decided by PA.(iii) Parent block has no distinct relay statement, and it is either empty or all its computations are connected with the inner block: They can be merged into one block due to its close association. (iv) Parent block has no distinct relay statement, and none or all its computations are connected with inner block is present: Their merger possibility is decided by Parent Affinity (PA). The metric parent affinity (PA) is defined as follows.

$$PA(v, G) = 1 - \frac{\text{IndependentNodes}(b_p, b_v, G)}{\text{ParentDataNodes}(b_p, b_v, G)}$$

where, (i) $b_v$ and $b_p$ represent the control blocks respectively at an inner control vertex $v$ and its parent vertex $p$, (ii) $\text{ParentDataNodes}(b_p, b_v, G)$ is the count of those vertices in $b_p$ from which there is at least one outgoing data edge. (iii) $\text{IndependentNodes}(b_p, b_v, G)$ is the count of those vertices in $b_p$ which are not directly connected with $b_v$ by a data edge (after contraction of $b_v$).

Example Use of Parent Affinity for CEC: Figure 5.2 (b) shows the SDG after contraction of the inner block rooted at vertex 9. After the contraction of this block, its parent block rooted at vertex 3 still contains six vertices in addition to the contracted vertex 9 and vertex 3 itself. Two of these vertices (6 and 8) are directly connected with a data edge to block 9, whereas, vertices 5 and 7 are indirectly connected to block 9 via data edges. In block 3, only two vertices (17 and 18) that do not supply/consume data to/from block 9 remain. Thus, $\text{Producer}(b_p, G)$ is the vertex set $V = \{5, 6, 7, 8, 17\}$. Out of these, producer vertices 5, 7, and 17 are not directly connected with inner block 9. They form the set $\text{IndependentNodes}$. So, parent affinity (PA) for block 3 is $(1 - (3/5)) = 0.4$. Since it is higher than default threshold of 0.34, this parent block is not merged with inner block 9. Thus, only inner block 9 with vertices 4, 9-16 is extracted as a separate method.
5.1.3 Contraction:

Once a control block is qualified for extraction, all control edges originating from it are contracted, resulting in reduction of the subgraph to the control block vertex called target. The target represents a newly contracted control block constituting a distinct functionality. Remaining two activities of the segmentation investigate possibilities of merging direct contributing data dependencies into the target. Procedure 1 shows the identification and contraction of control blocks as extract method opportunity. Procedure $\text{EdgeContraction}(v, u, v, G')$ contracts the edge between vertices $u$ and $v$. Vertex resulting from the contraction is given the label $v$ (third argument).

**Procedure 1: Control Edge Contraction (CEC)**

**Data:** SDG $G$

$G' \leftarrow G$

$ctrList \leftarrow \text{GetPrimaryControlVList}(G')$

$v \leftarrow \text{GetLast}(ctrList)$

**while** $\text{HasElement}(ctrList)$ **do**

$\text{Del}(v, \text{List})$

$count \leftarrow \text{GetRelayCount}(v, G') > 0$

**if** $count > 0$ **And** $CA(v, G') < \text{THRESHOLD}$ **then**

$v \leftarrow \text{GSI}(v, G', \text{List})$

$G' \leftarrow \text{SDDC}(v, \text{ESC}(v, \text{CCB}(v, G'))}$

**end**

$v \leftarrow \text{GetLast}(\text{List})$

**end**

**return** $G'$

**Procedure 2: Get Segment Id (GSI)**

**Data:** Vertex $v$, SDG $G$, PCL List

$G' \leftarrow G$

$p \leftarrow \text{getControlParent}(v, G')$

**while** $\text{IsPrimaryCtrlVtx}(p)$ **And** $PA(p, v, G') < P\text{Threshold}$ **do**

$\text{Del}(v, \text{List})$

$v \leftarrow p$

$p \leftarrow \text{getControlParent}(v, G')$

**end**

$\text{Del}(v, \text{List})$

**return** $v$

5.2 Exclusive Source Contraction

Once a control block (target) is contracted, the next activity involves merging of contractible initializations and input statements, which are those present in the same control region as that of the target. Each of the contractible input/initialization statements provides data to exactly one vertex in the graph produced by control edge contraction. Such statements in the SDG can hence be identified by tracking the source vertices with a singular outgoing data edge. Thus, an edge $\langle u, v \rangle$ originating from a source vertex $u$ is contracted only if its outdegree is one. The new vertex
Procedure 3: Contract Control Block (CCB)

Data: SDG $G$, Vertex $v$

$G' \leftarrow G$

while $HasControlEdge(v, G')$ do

$\langle v, u \rangle \leftarrow GetNextCtrlEdge(v, G')$

$G' \leftarrow EdgeContraction(v, u, v, G')$

end

return $G'$

created after contraction of the edge $\langle u, v \rangle$ is assigned the label $v$, thus merging loosely hanging source vertices with the main component to which they are data sources.

It is possible that the newly created vertex is also a source vertex for some other vertex. In such cases, the same process is repeatedly applied till the data source chain merges into the target. The merger of the source chain into the target results in a vertex with label of the target. Figure 4.2(g) shows the output of exclusive source contraction applied to the output of control edge contraction.

Procedure 4 outlines the process of exclusive source contraction. The procedure uses procedure $Source(G)$, which selects and returns the set of all source vertices in graph $G$. Procedure $EdgeContraction()$ is described in the previous section.

5.3 Sequential Data Dependence Contraction (SDDC)

This step groups statements involved in sequential data dependence. For example, consider $k$ statements $s_0, s_1, ..., s_{(k-1)}$ in a program such that each statement $s_i$ is data dependent on statement $s_{i-1}$ where $1 \leq i \leq (k-1)$. Such statements can be identified in the SDG by tracking chains of data dependency. These statements are grouped by contracting edges present in chains. A chain structure may cross beyond the control region of the target block. So, a truncation of the chain is required. Such a chain is truncated at the border edge connecting two of its vertices present in different control regions. Thus, only those vertices of the chain that are in the same control region are considered for merger. A chain can be associated with the target in two ways, as incoming or as outgoing. It can be observed that an incoming chain into the target supplies data to the block, whereas, chain outgoing from the target consumes data produced by the block. Each chain structure represents a subtask. But the challenge lies in identification of chain structures that keep the resulting block functionally coherent. The selection criteria for chain structures is given below.

5.3.1 Chain Structure Selection:

If there is only one chain incoming to the target vertex, it is merged into the target block, which is extracted as a method. The same applies to the lone outgoing chain from the target vertex. Such chain structures show strong association of data dependence with target block. Presence of more than one incoming or outgoing chains can be of two types based on size. These cases are handled as follows:

1. *Unit chains*: These chains are of length one. If incoming, such a chain represents an assignment of a variable or an input call such as `scanf()`. If outgoing, it represents a return statement, or an output statement or a statement which consumes but does produce a data value as shown in edge $\langle 4, 6 \rangle$ in Figure 5.1 representing use of variable `sum` in a printf statement outside the control block corresponding to the for loop.
All such chains can be merged to the target block. An added benefit of merger of incoming unit chain is that the definitions (of variables) required for the functionality in the block will be at one place. If the chain head has two or more predecessor then merger of the unit chain increases the parameter list of the extracted method. Similarly, an outgoing chain that is of length one and which does not extend outside of control region of target block, either represents an output statement or a data sink statement. A sink statement uses the result but does not create a data dependency on any succeeding statement. An added benefit of merger of outgoing unit chain is that it avoids extra efforts in managing more than one return arguments.

2. **Longer chains**: These chains are of length more than one. All such chains represent subtasks. In this case, after merging all one length chains if only one long chain remains, it is merged with the target block. However, if there are more than one long chains associated with the block, user intervention may be required to rank functional closeness of the chains with the target block. Hence, at present, segmentation does not automatically merge such chains with the target block.

Figure 4.2(h) shows the output of *sequential data dependence contraction* applied on graph produced by exclusive source contraction in Figure 4.2(g). Sequential data dependence contraction is implemented in Procedure 5.

### 5.4 The Segmentation Algorithm

The complete algorithm is listed as Algorithm 6 in the form of nested function calls nesting the contraction operations in segmentation described above. The input is SDG $G$, and the output is segment graph $S$. 
**Procedure 4:** Exclusive Source Contraction (ESC)

**Data:** $G'$: The graph produced by CEC, `target`: newly contracted control block Id  
$G'' \leftarrow G'$

SourceList $\leftarrow$ `SourceAt($G''$)`

while SourceList $\neq \emptyset$ do
  $u \leftarrow$ Next(SourceList)
  if $\text{Outdegree}(u) = 1$ And $\text{ControlRegionDiff}(\text{target}, u) = 0$ then
    $w \leftarrow \text{Adj}(u, G'')$
    $G'' \leftarrow \text{EdgeContraction}(\langle u, w \rangle, w, G'')$
    SourceList $\leftarrow$ `Source($G''$)`
  else
    SourceList $\leftarrow$ SourceList $- u$
  end
end

return $G''$

**Procedure 5:** Sequential Data Dependence Contraction (SDDC)

**Data:** $G''$: The graph produced by SESC, `v`: Newly contracted Block  
$G''' \leftarrow G''$

incoming $\leftarrow$ `ChainsAt($v, G''$)`

outgoing $\leftarrow$ `ChainFrom($v, G''$)`

count $\leftarrow 0$

while incoming $\neq \emptyset$ do
  $c \leftarrow$ Next(incoming)
  if length($c$) == 1 then
    EdgeContraction($\langle u, v \rangle, v, G'''$)
  else
    count $\leftarrow$ count + 1
    tmpchain $\leftarrow c$
  end

  incoming $\leftarrow$ incoming $- c$
end

if count == 1 then
  foreach $\langle u, w \rangle \in c$ do
    EdgeContraction($\langle u, w \rangle, u, G'''$)
  end
  EdgeContraction($\langle u, v \rangle, v, G'''$)
end

if length(outgoing) == 1 then
  $c \leftarrow$ Next(outgoing)

  foreach $\langle u, w \rangle \in c$ do
    EdgeContraction($\langle u, w \rangle, u, G'''$)
  end
  EdgeContraction($\langle u, v \rangle, v, G'''$)
end

return $G'''$
Algorithm 6: Segmentation

Data: SDG G
return (CEC(G))
Chapter 6

Evaluation

As discussed previously, the proposed approach aims at suggesting minimal set of extract method opportunities while maximizing functionally relevant suggestions. Table 6.1 lists the chosen sources for the evaluation case studies, mentioning the sizes of the programs and opportunities pre-marked by developers.

<table>
<thead>
<tr>
<th>Case Studies</th>
<th>#Methods (Program Size)</th>
<th>#Extract Method opportunities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>JUnit [12]</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>JHotDraw [12]</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>XData [13]</td>
<td>32</td>
<td>110</td>
</tr>
</tbody>
</table>

The evaluation of the approach consists of two parts. In the first part, we discuss the process of tuning segmentation for finding promising threshold values on one program in C that was manually mapped to intermediate representation. The program chosen was a non-refactored synthetically created version of our own implementation of the segmentation algorithm in this paper. The results obtained matched with manually refactored version. These results are discussed later in this section.

In the second part, we apply a tool based on the proposed technique on three open source Java programs and present the results, comparing them with another approach taken by the JDeodorant [10] tool. The results obtained after refactoring and also a feedback obtained from a developer of one of the programs are discussed. Two of the three chosen Java-based OSS case studies, JUnit and JHotDraw, were created by silva et al. [12] by inlining the callee methods inside the caller method. Both have been used earlier in the literature [12], [9] as benchmarks for long method extraction. The third OSS project chosen is the XData grading system for queries [13], which contains method sizes of sufficiently varied lengths (30-200 LOC) for our experimentation.

6.1 Performance metrics used for comparison

To measure the performance we use precision, recall and F measure metrics. Extract method opportunities listed by the our approach and JDeodorant tool are compared with pre-marked opportunities, and the shares of True Positives, False Positives, and False Negatives are identified as mentioned below.
- **True Positive (TP):** opportunity identified by both the developer and the algorithm
- **False Positive (FP):** opportunity identified only by the algorithm
- **False Negative (FN):** opportunity identified only by the developer

\[
\text{Precision} = \frac{\#TPs}{\text{Retrieved opportunities}} = \frac{\#TPs}{(\#TPs + \#FPs)}
\]

\[
\text{Recall} = \frac{\#TPs}{\text{Relevant opportunities}} = \frac{\#TPs}{(\#TPs + \#FNs)}
\]

\[
\text{Fmeasure} = \frac{2 \times (\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})}
\]

**Match Tolerance:** We have analyzed the effect of applying *match tolerance* values of 1 to 3 statements on the identification of opportunities by our algorithm. A deviation of a match within the tolerance range in both direction is acceptable. The same tolerance value is used for the JDeodorant tool for the purpose of comparison.

### 6.2 The Process of Tuning

The proposed segmentation approach primarily uses three parameters to control the number of extract method opportunities (suggestions), namely, LoCS, PA as discussed earlier, and a flag `NoRelayExtract`. This flag is used as follows. By default, segmentation computes LoCS metric for blocks containing at least one variable (relay vertex), which is assigned inside and is used outside the block. However, in some cases, a method may not contain a relay vertex, but may still represent a distinct functionality. To account for such methods, this flag can be set to value `True`, which allows the segmentation algorithm to compute values of LoCS for such blocks also. This is achieved by setting the numerator value `#Relay` to one in equation 5.1.1. Blocks for which LoCS is not computed are not considered into output suggestions. To find the appropriate threshold values for aforementioned parameters, we performed a study on an older version of segmentation approach and analyzed the extract method suggestions against varying thresholds for LoCS and PA. The values of LoCS and PA found in the case study discussed in this subsection are used for three other case studies discussed in the next section. Now, we discuss the method of tuning LoCS and PA.

An older *modularized base code* of segmentation implemented in C was first transformed into *non-modularized code*, by unfolding a few functionalities into their caller functions. New functions of varying sizes were formed by unfolding functions in the source. The non-modularized code for our evaluation that is generated from the above method consisted of a total of 28 distinct functionalities spread over 13 functions. Ten out of these 13 functions contain a single distinct functionality, resulting in three large functions containing 18 functionalities all in all. The original modularized base code was considered for comparing the results. Table 6.2 shows the precision and recall values for the transformed code. It is observed that threshold values of 0.41 and 0.34 respectively for LoCS and PA, provided the best results for this case study. Flag `NoRelayExtract` was set to `False`. These parameter settings for segmentation are used as default seed values/setting for next three applications, which are tuned as per the need of the application.
Table 6.2: Precision and Recall obtained after tuning

<table>
<thead>
<tr>
<th>Methods to be refactored</th>
<th>#LoC</th>
<th>#Relevant Retrieved</th>
<th>TP FP FN</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 FindExcSourceNodes</td>
<td>55</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>#2 InitializeSourceList</td>
<td>23</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>#3 FindChain</td>
<td>137</td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

6.3 Application to OSS code

JUnit and JHotDraw projects provide already marked extract method opportunities, whereas for XData grading system, we requested one of the developers to manually identify the extract method opportunities. The developer in this case was a research scholar with some industry experience. A brief description about our approach was provided to the developer before marking. Table 6.3 shows the settings for which we obtained satisfactory results for all three studies. We find that segmentation with default setting is not able to extract any extract method opportunity for JUnit, whereas in case of JHotDraw, it provides better precision than JDeodorant with a low recall.

Table 6.4 shows the number of opportunities suggested by both approaches for each study. As it can be observed that segmentation and JDeodorant are comparable except for XData system, for which the latter suggested far more opportunities than marked by the developer.

Tables 6.5, 6.6, and 6.7 present a detailed comparison of our approach with JDeodorant over the three applications. The experiments were conducted for three match tolerance values of 1, 2 and 3. It can be observed that JDeodorant performed better than segmentation for JUnit, whereas in case of JHotDraw and XData, segmentation performed better. The results are now analyzed with pointers to future work.

- **Accuracy in suggestions**: Segmentation outperforms JDeodorant in precision as well as recall except in case of JUnit as shown in Table 6.5, 6.6 and 6.7. We observed that segmentation has suggested a few marked opportunities in JUnit, but they were wrapped with an additional associated if-else block or a try-catch block. Such cases resulted in lower precision. In future, these features can be explored as alternate suggestions or improvements, assisting the method of manual refactoring.

- **Effect of NoRelayExtract Flag**: This flag can be set to True if the opportunities of interest do not have a relay vertex in the SDG. It can be seen from Tables 6.5 and 6.7 that in the case of JHotDraw and JUnit, the performance of the segmentation approach improved when the flag was set to True. The same is not observed for XData, in which, there are large functionalities. In this case the flag set to False provides better overall performance as shown in Table 6.6.

Table 6.3: Tuning segmentation for best results

<table>
<thead>
<tr>
<th>Projects</th>
<th>LoCS value</th>
<th>PA value</th>
<th>'NoRelayExtract'</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUnit</td>
<td>0.41</td>
<td>0.34</td>
<td>True</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>0.41</td>
<td>0.34</td>
<td>True</td>
</tr>
<tr>
<td>XData</td>
<td>0.41</td>
<td>0.34</td>
<td>False</td>
</tr>
</tbody>
</table>
In future this selection can be incorporated through an application specific decision making.

- **Performance over long methods**: To measure the performance of both approaches over long methods, we selected methods with more than 150 LOC. Table 6.8 tabulates the results for the segmentation approach. JDeodorant did not suggest any matching suggestions in these cases. Thus, segmentation provided significantly better performance over JDeodorant in terms of precision and recall. To note, the recall crossed over 50. We haven’t observed the performance of the approach over very large method such as in range 200+ LOC. In such cases although the programmer may a set of functionality as one unit, the tool may suggest a break-up. The approach can be assessed for identification of such large units to assist in refactoring.

- **Comparing with expert’s markings**: One of the primary motivation behind choosing XData system was to apply the approach over non-synthetic long method, and also to compare the result with expert’s markings. We note that both the tools have identified and suggested a similar set of opportunities that were forming blocks bigger than marked opportunities crossing the marking tolerance. Hence they were not classified as matches. However, we did not perform further analysis of scrutinizing the results of manual markings. This line can be further investigated for using the tool in assistance with the expert or to evaluate manual decisions.

- The effect of programming language features: The proposed algorithm identified majority opportunities present in the considered applications compared to JDeodorant. However, we noticed that a few of the opportunities that were not identified were Java specific, especially anonymous inner classes. Inclusion of support for language specific plugins to the SDG model may further improve the performance of the approach.

### Table 6.4: Extract method opportunities for all methods

<table>
<thead>
<tr>
<th>Tools/OSS</th>
<th>JUnit (25)</th>
<th>JHotDraw (56)</th>
<th>XData (110)</th>
<th>Total(191)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>11</td>
<td>72</td>
<td>117</td>
<td>200</td>
</tr>
<tr>
<td>JDeodorant</td>
<td>14</td>
<td>63</td>
<td>202</td>
<td>279</td>
</tr>
</tbody>
</table>

31
Table 6.5: Comparison for JHotDraw

<table>
<thead>
<tr>
<th>Tools</th>
<th>Tolerance</th>
<th>Precision</th>
<th>Recall</th>
<th>F measure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>15.27</td>
<td>19.64</td>
<td>17.18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>23.61</td>
<td>30.35</td>
<td>26.56</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>30.55</td>
<td>39.28</td>
<td>34.37</td>
</tr>
<tr>
<td>Segmentation</td>
<td></td>
<td>7.83</td>
<td>8.92</td>
<td>8.40</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11.11</td>
<td>12.5</td>
<td>11.76</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>20.63</td>
<td>23.21</td>
<td>21.84</td>
</tr>
<tr>
<td>(<code>NoRelayExtract</code>= True)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDeodorant</td>
<td>1</td>
<td>12</td>
<td>5.35</td>
<td>7.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>20</td>
<td>8.92</td>
<td>12.34</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>28</td>
<td>12.5</td>
<td>17.28</td>
</tr>
<tr>
<td>Segmentation</td>
<td></td>
<td>51.28</td>
<td>54.54</td>
<td>52.86</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>55.55</td>
<td>59.09</td>
<td>57.26</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>57.26</td>
<td>60.09</td>
<td>59.03</td>
</tr>
<tr>
<td>(<code>NoRelayExtract</code>= False)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Comparison for XData grading system

<table>
<thead>
<tr>
<th>Tools</th>
<th>Tolerance</th>
<th>Precision</th>
<th>Recall</th>
<th>F measure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>51.28</td>
<td>54.54</td>
<td>52.86</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>55.55</td>
<td>59.09</td>
<td>57.26</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>57.26</td>
<td>60.09</td>
<td>59.03</td>
</tr>
<tr>
<td>Segmentation</td>
<td></td>
<td>2</td>
<td>3.23</td>
<td>2.53</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.46</td>
<td>6.36</td>
<td>4.48</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.46</td>
<td>6.36</td>
<td>4.48</td>
</tr>
<tr>
<td>(<code>NoRelayExtract</code>= False)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDeodorant</td>
<td>1</td>
<td>9.09</td>
<td>4</td>
<td>5.55</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9.09</td>
<td>4</td>
<td>5.55</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>18.18</td>
<td>8</td>
<td>11.11</td>
</tr>
<tr>
<td>(<code>NoRelayExtract</code>= True)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.7: Comparison for JUnit

<table>
<thead>
<tr>
<th>Tools</th>
<th>Tolerance</th>
<th>Precision</th>
<th>Recall</th>
<th>F measure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>35.71</td>
<td>20</td>
<td>25.64</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>42.85</td>
<td>24</td>
<td>30.76</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>42.85</td>
<td>24</td>
<td>30.76</td>
</tr>
<tr>
<td>Segmentation</td>
<td></td>
<td>38.29</td>
<td>50</td>
<td>43.37</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>40.42</td>
<td>52.77</td>
<td>45.78</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>40.42</td>
<td>52.77</td>
<td>45.78</td>
</tr>
<tr>
<td>(<code>NoRelayExtract</code>= False)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8: Performance over long methods (LOC>150)

<table>
<thead>
<tr>
<th>Tools</th>
<th>Tolerance</th>
<th>Precision</th>
<th>Recall</th>
<th>F measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
<td>1</td>
<td>38.29</td>
<td>50</td>
<td>43.37</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>40.42</td>
<td>52.77</td>
<td>45.78</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>40.42</td>
<td>52.77</td>
<td>45.78</td>
</tr>
<tr>
<td>(<code>NoRelayExtract</code>= False)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7

Conclusions and Future Work

Segmentation, a novel graph-based approach for extract method refactoring was presented. The segmentation algorithm uses a language independent intermediate representation called segment IR. A structure dependence graph (SDG) is extracted from the segment IR. The algorithm transforms this graph into a segment graph, which is obtained by contracting subgraphs through edge contraction. The approach was evaluated on code from three different open source software projects, and promising results were obtained. The approach was found to have worked well on long methods. The SDG based framework developed in the paper is open to tuning. The approach can be used as a refactoring assistant to aid human expert in identifying modular functionalities through extract method opportunities. Further work in this area includes extension to cover inter-procedural analysis, and merger of segments across procedures. Impact of language specific features on modularization can also be investigated.
Bibliography


