Analysis of Flow of Control for Reverse Engineering of Sequence Diagrams

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Olga Nicole Volgin, B.S.

* * * * *

The Ohio State University

2005

Master's Examination Committee:

Atanas Rountev, Adviser

Sandra Mamrak

Approved by

Adviser

Department of Computer and Information Science

ABSTRACT

During software lifecycle, the design documentation and implementation often diverge. This is especially true in iterative development processes and in legacy systems. System enhancement based on inaccurate documentation may result in costly software design and implementation flaws. Consequently, reverse engineering of system design could be beneficial during enhancement and maintenance. Important aspects of software design are represented using UML sequence diagrams, which are considered to be one of the core design artifacts. Recent introduction of the next generation UML presents new challenges for reverse engineering of sequence diagrams. One such challenge is the mapping of the flow of control in the code to the UML sequence diagram primitives.

The work presented here addresses this problem through a static analysis algorithm for representing flow of control in sequence diagrams. We analyze control-flow graphs and map them to the UML primitives. We also propose a series of transformations on the resulting structure that are intended to improve readability and comprehension of the diagrams. These transformations reduce nesting while preserving the meaning of the diagrams. The simplification of the diagrams makes them easier to read and comprehend. We present an experimental study that evaluates the practicality of our analysis on several Java library components and the benefits of the transformations in reducing nesting in reverse-engineered sequence diagrams.

ACKNOWLEDGMENTS

I would like to thank my advisor Nasko Rountev for the opportunity to work on this project. His guidance throughout this project was invaluable and the feedback on the thesis was very helpful.

I also would like to thank Sandra Mamrak, who is my former advisor as well the other committee member, for the opportunity to be a part of the Acuity Project for the last two years.

Additionally, I would like to thank Miriam Reddoch for her contribution to this work with the implementation of the CFG generation and for being such a pleasant person to work with.

I am also very grateful to all of my family and friends for providing support and encouragement throughout this process. I would like to especially thank Mike Gibas for being so supportive and helpful.

VITA

June 1, 1977	Born - Moscow, Russia
Jannuary 1999 - December 1999	Engineering Co-op, Applied Innovation Inc., Columbus, OH
June 2000	B.S. Computer Science and Engineering, The Ohio State University, *Magna Cum Laude*
July 2000 - September 2002	Member of Technical Staff, Lucent Technologies, Columbus, OH
September 2002 - May 2004	Graduate Research Assistant, Acuity Project, The Ohio State University

FIELDS OF STUDY

Major Field: Computer Science and Engineering

TABLE OF CONTENTS

Page

Abstract ii

Acknowledgments iii

Vita iv

List of Figures vii

Chapters:

1. Introduction 1

2. UML 2.0 7

    2.1 Running Example 7
    2.2 UML 2.0 Overview 7

3. Representation of Intra-Method Flow of Control 15

    3.1 Data Structure Description 15
    3.2 Fragment Descriptions 18

4. Control Flow Analysis - Basics 24

    4.1 Phase I: Preprocessing 24
        4.1.1 Control Flow Graphs and Post-dominators 24
        4.1.2 Loops 28
        4.1.3 Branch Successors 30
        4.1.4 Loop Successors 33
    4.2 Phase II: Fragment Construction 39

5. Control Flow Analysis - Advanced Issues 47

5.1 UML Deficiencies 47
5.2 Proposed UML 2.0 Extensions 50
5.2.1 Notation for Multiple Returns 50
5.2.2 Exceptional Behavior 52
5.3 Data Structure Additions 53
5.4 Phase I: Preprocessing 53
5.4.1 Post-dominance 53
5.4.2 Control Dependence 55
5.4.3 Identifying Paths Leading to Exceptional Behavior 57
5.4.4 Processing of Multiple Method Exits 58
5.5 Phase II: Fragment Construction 60

6. Phase III: Fragment Transformations 64

6.1 Clean-up Transformations 65
6.1.1 Removal of Empty Alt, Opt, and Loop Fragments 65
6.1.2 Removal of Implicit Break Fragments 66
6.1.3 Replacing an Alt Fragment with an Opt Fragment 68
6.2 Readability Transformations 69
6.2.1 Moving of Nested Alt Cases 69
6.2.2 Moving of Fragments Surrounded by an Opt Fragment . . . 70
6.2.3 Removing an Opt Fragment Enclosed by a Case 72
6.2.4 Generalized Removal of Opt Fragment 73

7. Empirical Study 75

8. Related Work and Conclusions 83

Bibliography 87

LIST OF FIGURES

Figure **Page**

2.1 Sample classes based on package `java.text` 8

2.2 Java code for the running example 9

2.3 Sequence diagram for the running example 10

3.1 Fragment data structure for the running example 16

4.1 CFG for the running example 26

4.2 Fragment data structure for the running example, replica of Figure 3.1 for convenience 27

4.3 Post-dominator tree for the running example 28

4.4 Post-dominator tree for loop L in the running example 32

4.5 Java code illustrating the computation of loop successors 35

4.6 Control flow graph illustrating the computation of loop successors . . 36

4.7 Loop information for the three loops 37

4.8 Algorithm for fragment construction 45

4.9 Algorithm for fragment construction 46

5.1 Sample classes based on standard package `java.util.zip` 48

5.2 Java code illustrating multiple returns and exceptional behavior . . . 49

5.3 Sequence diagram illustrating the proposed UML extensions 51

5.4 Fragment structure generated for method `read` 54

5.5 Control flow graph illustrating computation of control dependence . . 56

5.6 Controlling edge information for method exits for CFG in Figure 5.5 . 56

5.7 Algorithm for fragment construction 62

5.8 Algorithm for fragment construction 63

6.1 Illustration of the removal of empty fragments 67

6.2 Illustration of the removal of implicit break fragments 69

6.3 Replacing an alt fragment with an opt fragment 70

6.4 Moving of nested alt cases 71

6.5 Moving of fragments surrounded by an opt fragment 72

6.6 Removing an opt fragment enclosed by a case 73

6.7 Generalized removal of opt fragments 74

7.1 Changes in average nesting depth from Table 7.2 79

CHAPTER 1

Introduction

As software evolves during its lifetime, the documented system design and implementation often diverge. As a result of the lack of complete and current design documentation, system understanding and enhancement may become challenging. This is especially true for complex legacy systems. The investments made into such systems during their lifetime make them valuable and difficult to replace. Consequently, the software continues to be enhanced and maintained for many years. In this process of evolution, the implementation may diverge from the documented system design making the documentation incomplete or outdated. Over the years, the enhancement and maintenance of such systems may become progressively difficult, especially in the absence of the original designers and developers.

System documentation and implementation may also deviate in the process of iterative development that is becoming more widely used. In an iterative process, system development is organized in short-termed iterations where the system is grown incrementally one iteration at a time. Each iteration involves analysis, design, coding, and testing phases. However, the design created in the beginning of an iteration often may change as the implementation raises new issues. As a result, the documentation created in the beginning of an iteration may not reflect the system design produced

at the end of an iteration. Because the design and implementation of each subsequent iteration are dependent on the previous ones, it is useful to extract the system design at the end of each iteration in order to ensure that the design and implementation of the following iteration are based on accurate design documentation from the previous iteration [7]. To overcome the difficulties in system development and enhancement caused by inadequate documentation, tools for extracting system design from code could be especially useful. Such tools would provide a consistent and cost-effective way of producing the latest design documentation based on existing implementation.

Software is often documented using the standardized notation provided in the Unified Modeling Language (UML). UML is used for specification, visualization, and documentation of the design of software systems. It provides the means for representing the system design using various types of diagrams. The type of diagram that is especially useful in modeling a system's dynamic behavior is the UML *sequence diagram*. Sequence diagrams are included in the UML as a notation for illustrating the requests of actors on the system and the operations that are initiated by them on system objects [7]. Sequence diagrams clearly depict the flow of control during object interactions in a visual manner. As a result, they are often used in analysis and design of software systems. Sequence diagrams abstract away intricacies of the code while showing the messages exchanged between objects and their temporal sequencing. Due to its capabilities to depict the sequence of events and to show the lifetime, creation, and destruction of objects, sequence diagrams are considered as one of the core UML design artifacts. An example of a sequence diagram is presented in Figure 2.3.

It is possible to forward engineer or reverse engineer sequence diagrams. During *forward engineering*, the sequence diagrams are generated during the system design stages. The skeleton of the code is then automatically generated from the sequence diagrams. It is also possible to *reverse engineer* sequence diagrams, in which case, the diagrams are automatically extracted from source code and consequently represent the existing design of the system. Reverse engineering of sequence diagrams for complex systems is useful because it provides insights into object interactions existing in the system. Utilization of tools for reverse engineering of sequence diagrams would provide a cost-effective way of automatically generating up-to-date design documentation with uniform level of detail presented in consistent notation. The resulting sequence diagrams would make it easier to understand the system and also would provide the latest design documentation useful for system maintenance and enhancement. Although existing tools provide the means for reverse engineering of UML class diagrams, they either do not have the capability to reverse engineer sequence diagrams, or such capabilities are limited.

The context of this work is the **RED** (**r**everse **e**ngineering of UML sequence **d**iagrams) tool. Incorporated in this tool are several analyses for reverse-engineering of sequence diagrams that utilize the latest UML 2.0 notation. In comparison with the earlier versions of the UML (1.x), the UML 2.0 standards utilize more expressive notation for the representation of system design. The new UML 2.0 specification introduces the notion of *interaction fragments*. These fragments represent different aspects of flow of control, and are used as the building blocks of the sequence diagrams. The latest UML notation makes it possible to express optional, alternative,

repetative, and breaking behavior in sequence diagrams. Figure 2.3 presents a sequence diagram that uses the UML 2.0 notation. Reverse-engineering of sequence diagrams is a complicated task requiring several analyses addressing various aspects of the problem. Our **RED** tool utilizes the UML notation and incorporates several such analyses including the *control flow analysis* presented in the scope of this work. This analysis focuses on method-level interactions and will later be incorporated into a larger inter-method analysis. The control flow analysis presented here focuses on the problem of mapping method-level flow of control to UML 2.0 primitives. In this work, the UML 2.0 primitives are represented in a form of a data structure that embodies the UML notation. Other analysis in **RED** along with visualization tools will use the data structure created as result of the control flow analysis to eventually generate sequence diagrams. The input to our control flow analysis is the method-level *control-flow graph* (CFG) and the output is the data structure representing the flow of control for the method.

The elements of the data structure correspond to the UML interaction fragments. The data structure is generated as the result of a multi-phase analysis of a method-level control flow graph. A control-flow graph consists of nodes corresponding to code statements and edges that represent the possible flow of control between the nodes. In first phase of our analysis, the CFG is analyzed to identify important aspects of flow of control such as loops and branches. This information is then utilized in the second phase of our analysis that uses an algorithm for mapping intra-method flow of control represented in the CFG to UML 2.0 fragments. As result of this step, we create the fragment data structure representing the method-level flow of control. An example of our data structure is shown in Figure 3.1.

The fragment structure produced by the control flow analysis may contain some redundant fragments not contributing useful information to the resulting sequence diagram. Additionally, the fragment structure may also contain deep nesting of fragments that reduces its readability and comprehension. In order to simplify the data structure and improve readability and comprehension, we introduce several *fragment transformations*. These transformations improve the fragment structure while preserving its meaning. The redundancies in the fragment structure are eliminated by *clean-up transformations*. These transformations remove any empty fragments that do not contribute useful flow of control information to the resulting sequence diagrams. The *readability transformations* improve the comprehension of the sequence diagrams by reducing the nesting of fragments.

The empirical study presented in this work uses 21 Java library components to evaluate the analysis cost, the occurrence and distribution of fragment nesting, and the effectiveness of the readability transformations on the reduction of nesting. The study demonstrates that fragment nesting is prevalent in real Java components and suggests that the readability transformations can successfully reduce the nesting and potentially improve the comprehension of resulting sequence diagrams. The running times of the analysis are low, which strongly indicates that it can be used to analyze complex real-world systems.

The contributions of this work include:

- The first general algorithm for mapping intra-method flow of control to UML 2.0 interaction fragments
- Transformations for improving diagram structure and readability

- Extensive experimental evaluation of the benefits of the transformations and the cost of our control-flow analysis

The rest of the thesis is organized as follows. Chapter 2 introduces the UML 2.0 notation. The details of the fragment structure are presented in Chapter 3. The algorithm for mapping intra-method flow of control to UML interaction fragments is described in Chapter 4. Chapter 5 addresses additional issues related to flow of control for methods with multiple returns and exceptional behavior. The transformations for simplifying and improving the readability of the fragment structure are detailed in Chapter 6. Chapter 7 presents the empirical study that evaluates the efficiency and effectiveness of our analysis. Chapter 8 concludes the thesis with the description of related work and conclusions on the contributions of this work.

CHAPTER 2

UML 2.0

2.1 Running Example

This section introduces the example illustrating the important aspects of our control flow analysis for reverse engineering of sequence diagrams. The example presented here will be referenced in this and subsequent chapters. The code introduced in Figure 2.1 and Figure 2.2 was selected because of its relevance to the key aspects of our analysis. It is based on several methods of class `MergeCollation` found in standard package `java.text`. However, it was modified to better illustrate ideas presented in this work. The sequence diagram corresponding to the code can be found in Figure 2.3.

2.2 UML 2.0 Overview

As described in Chapter 1, sequence diagrams abstract away details of the code and represent temporal sequencing of messages exchanged between objects. Historically UML sequence diagrams have been widely used as one of the core software design artifacts, but the UML specification lacked the capabilities to represent some of the

```
public class MergeCollation {
   public void example(PatternEntry e) { ...}
   private final void fixEntry(PatternEntry entry) { ...}
   private Vector patterns;
   private byte[] statusArray = new byte[8192];
   ...
}

public class PatternEntry {
   public String getChars() { return chars; }
   private String chars;
   ...
}
```

Figure 2.1: Sample classes based on package `java.text`

more complex design intricacies. The latest evolutionary UML 2.0 standard introduces new, richer control flow primitives, which allow to better document alternative, optional, repeating, and breaking behavior.

In order to represent this behavior, UML 2.0 introduces new primitives called *interaction fragments*. These entities are the building blocks of the diagram and they represent various types of interactions. The interaction fragments relevant in the scope of this work include *message, opt, alt, loop,* and *break*. An example of each fragment type can be found in Figure 2.3. The detailed description of each type of UML 2.0 fragment follows.

The *message* fragment represents a message exchanged between two objects where one of the objects sends the message and the other receives it. An illustration of a message fragment is `getChars()` in Figure 2.3. An object of type `MergeCollation` is the message sender and the object `e` of type `PatternEntry` is the receiver object. A

```
[1]  void example(PatternEntry e)
[2]  {
[3]      int i = -1;
[4]      String s = e.getChars();

[5]      if (s != null) {
[6]          i = s.charAt(0);
[7]      } else {
[8]          i = patterns.indexOf(e);
[9]      }

[10]     for (; i >= 0; --i) {
[11]        if (statusArray[i] !=0 ) {
[12]           PatternEntry e1 = (PatternEntry) patterns.elementAt(i);
[13]           if (e1 != null) {
[14]              fixEntry(e1);
[15]              break;
[16]           }
[17]        }
[18]        patterns.removeElementAt(i);
[19]     }
[20]     return;
[21] }
```

Figure 2.2: Java code for the running example

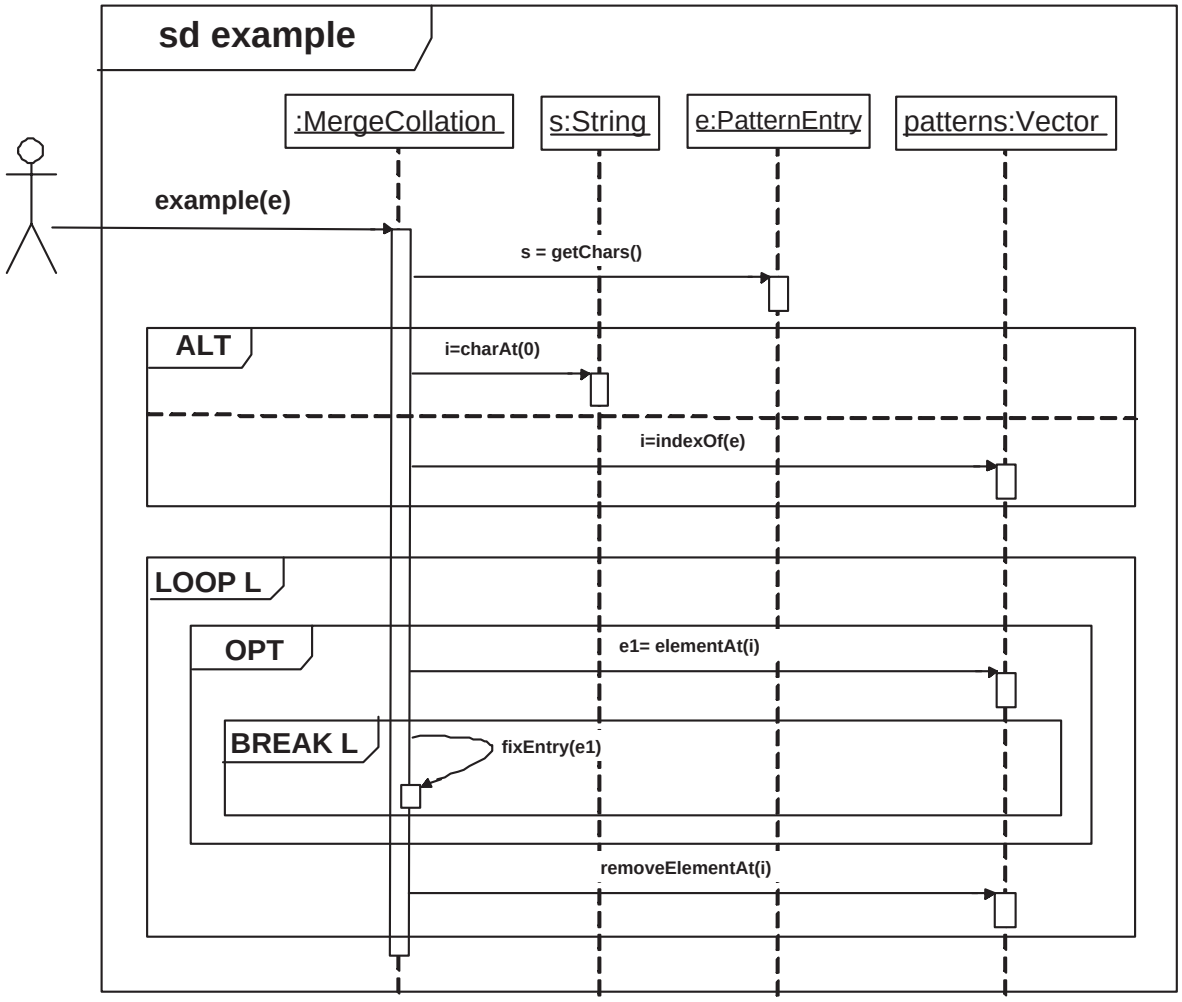

Figure 2.3: Sequence diagram for the running example

message fragment is represented by an arrow in a sequence diagram and corresponds to the appropriate method invocation in the code. Although the UML specification [10] treats the end points of the message as two separate fragments, in our approach this distinction is irrelevant and consequently a message exchanged between two objects is treated as a single message fragment.

The message fragment is the most primitive fragment type in the UML notation. Message exchanges are at the core of the UML sequence diagrams. The remaining fragment types are used to express the conditions guiding the possible paths of execution. The UML notation represents these paths by enclosing fragments affected by a condition inside the fragment affecting it. As a result, nesting of fragments is necessary to correctly represent the flow of control. Figure 2.3 provides an illustration of fragment nesting. Any non-message fragment can nest any number of fragments. Consequently, nested fragments may enclose other fragments thus resulting in multiple nesting levels of fragments. Neither the number of fragments enclosed by another fragment nor the depth of fragment nesting are limited.

The *opt* fragment was introduced in UML to represents optional behavior. The contents of the opt fragment are executed only when its guarding condition evaluates to true. Otherwise the execution of the opt operand is skipped. In most programming languages, the opt fragment corresponds to an `if` block. For instance, in Figure 2.3, the opt fragment is created to reflect the corresponding `if` block in the code, which is guarded by the condition `(statusArray[i] !=0)`. The fragments enclosed by the opt are executed only when the condition evaluates to true.

The *alt* fragment expresses two or more mutually-exclusive alternatives in behavior. Each one of the alternatives is represented by a separate path of execution and

an implicit or explicit guarding condition. Because each one of the paths corresponds to a different execution scenario, it is represented by a different sequence enclosed by the alt fragment. In the diagram the different alternatives, also referred to as cases, are separated by a dashed line. Figure 2.3 provides an illustration of an alt fragment. For a case of an alt fragment to be executed, its guarding condition must evaluate to true. If one of the alternative operands does not have a guard, an implicit true guard is implied. An alt fragment can be used to represent a `switch` statement found in most programming languages. Each one of the cases of the `switch` statement is normally guarded by a condition. However, there may also be the `default` case, which is executed when the guarding conditions of all other alternatives evaluate to false. Another example of code resulting in the creation of an alt fragment in the sequence diagram is a `if (c1) then ...else ...` statement. In this situation, the statement represents two distinct alternatives where one of the alternatives is guarded by the condition `c1` and the other guarded by the condition (!c1). An illustration of such alt fragment is presented in Figure 2.3. In this example, the first alternative is guarded by the condition `(s!=null)` and contains the message corresponding to the method invocation `charAt(0)` on a String object, while the second alternative is implicitly guarded by the condition `(s==null)` and encloses the method call `indexOf(e)` on a Vector object. An opt fragment described earlier can be considered as a special case of an alt fragment. In this case the opt fragment can be represented by an alt fragment with two alternatives. One of the paths of execution includes the contents of the opt fragment, while the other alternative skips the contents of the opt fragment and is therefore empty. Since in this case one of the alternatives is empty, it contributes no

flow of control information to the sequence diagram and thus opt fragment is used instead.

In order to represent the behavior of loops, UML 2.0 introduced the *loop* fragment. The contents of the loop fragment can be executed repeatedly until the guarding condition of the fragment evaluates to false. Additionally, the execution of the loop can be terminated by a *break*, which is described shortly. In Figure 2.3 the loop is guarded by the condition `(i>=0)` and contains an opt and a message fragments. The execution of the loop is terminated when either the loop guarding condition evaluates to false or the condition `(e1!=null)` for the break fragment nested inside the opt evaluates to true.

As mentioned earlier, the UML 2.0 notation also includes the *break* fragment, which represents a breaking scenario from one of the surrounding fragments. In Java, a break statement without a label transfers control to the innermost enclosing `switch, for, while,` or `do...while` statement. This statement then immediately completes normally [5]. The UML 2.0 break fragment is suited to represent this scenario. Similarly to an opt fragment, the operand inside the break fragment is executed only when its guarding condition evaluates to true. However, if the break fragment is entered, the execution of the remainder of the enclosing fragment is skipped. After completing the execution of the break fragment, the path continues immediately following the one of the fragments enclosing the break fragment. The illustration of a breaking scenario can be found in Figure 2.3, where the iteration of the enclosing loop is terminated by the break fragment. The contents of the break fragment are executed if the condition `(e1!=null)` evaluates to true. Then the execution proceeds

to execute the fragment immediately following the enclosing loop fragment, which in this case corresponds to the method exit.

CHAPTER 3

Representation of Intra-Method Flow of Control

3.1 Data Structure Description

The UML notation described in Chapter 2 provides a powerful way of documenting system design in the form of sequence diagrams. However, when creating a tool for reverse-engineering sequence diagrams, it is important to design a data structure that embodies the UML notation. This data structure would store the flow of control information discovered in the process of program analysis in a format easily convertible to the visual representation of the reverse-engineered diagram. We designed such a data structure and Figure 3.1 provides its visual representation for the example presented in Figure 2.2.

The building blocks of the data structure are entities corresponding to the interaction fragments in UML 2.0. Consequently, our data structure incorporates *message, opt, alt, break,* and *loop* fragments. Additionally, our data structure includes a convenience *top* fragment which is used to indicate the boundary for a method. Each one of the fragments mentioned above is described in detail in Section 3.2.

In order to accomodate fragment nesting necessary for representation of flow of control, the data structure is designed to be similar to a tree. All fragment types,

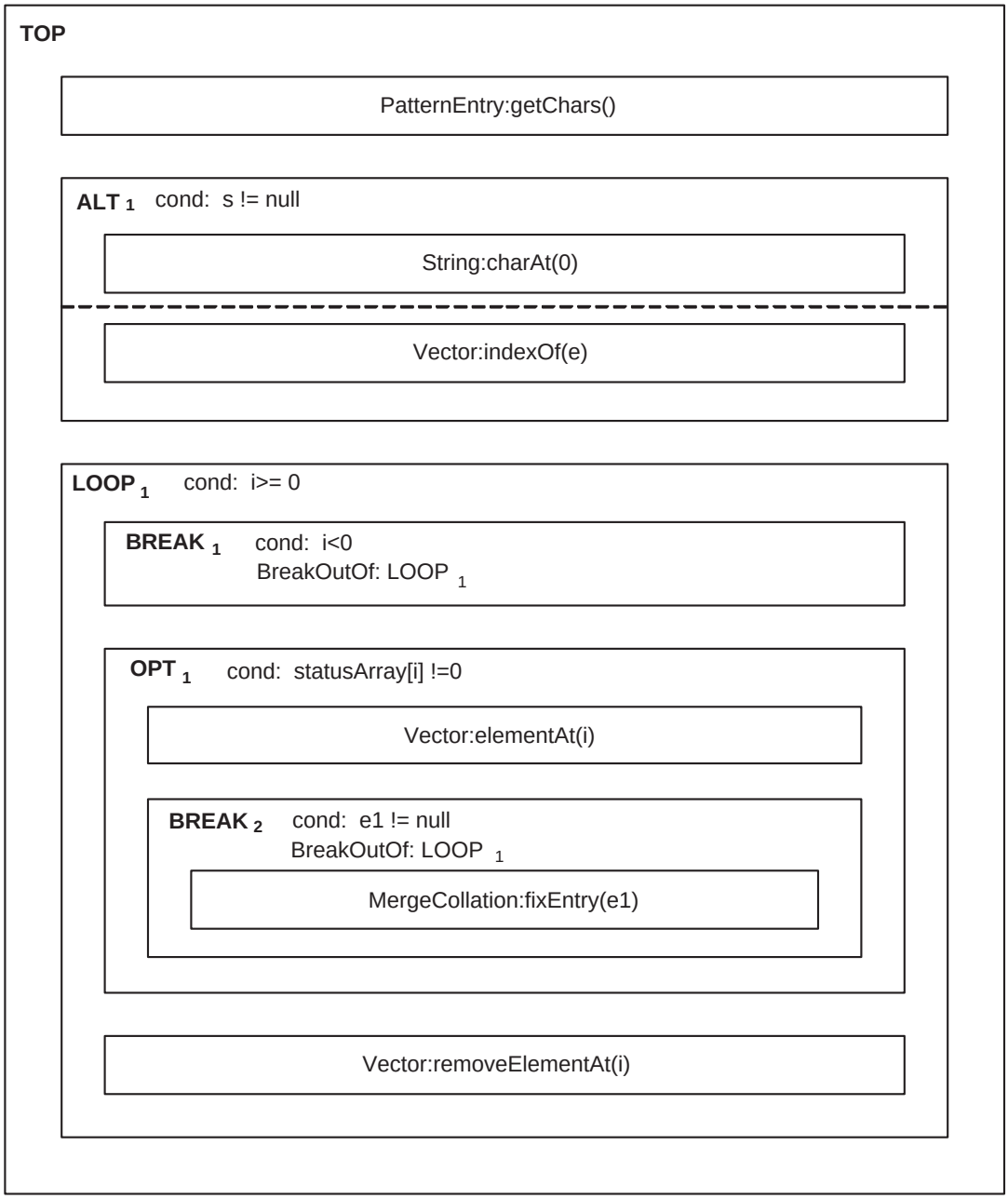

Figure 3.1: Fragment data structure for the running example

with the exception of message fragments, enclose one or more ordered sequences of fragments. For example, the opt fragment in Figure 2.3 encloses the message fragment corresponding to the invocation expression `patterns.elementAt(i)`, followed by the break fragment. The corresponding opt fragment in our data structure contains an ordered sequence of fragments, with the message fragment followed by the break fragment. It is important to note that the appropriate order of fragments is always preserved to correctly reflect the flow of control. The nesting depth of fragments is not limited because the nesting observed in existing code can often be quite deep. However, it is also possible for a fragment to not enclose other fragments. Certain such fragments that do not contain useful information will be eliminated during the transformations described in Chapter 6.

For each fragment, the sequence of its nested fragments represents all possible run-time paths of execution. The actual run-time patterns depend of the conditions of the execution, but the nested fragments represent all possibilities. These nested fragments can be considered as ordered children of the enclosing fragment in the data structure. For example, the opt fragment discussed above has two children. However, it is also possible for the fragments to have no children. This happens when the contents of the fragment do not correspond to important aspects of flow of control and therefore are not represented by fragments.

The nesting relationships seen in the data structure not only mirror the original flow of control, but also correspond to the structure of the resulting sequence diagram.

3.2 Fragment Descriptions

Each sequence diagram represents a scenario depicting interactions between various objects. The data structure for the scenario consists of the fragments that are enclosed by a container representing the method boundary. In order to represent the outermost boundary, a convenience *top* fragment was introduced. The top fragment is not directly present in UML 2.0 notation. However, for the purposes of understanding and consistency it was introduced as a special container fragment type. Due to intra-method nature of our analysis in the scope of this work, the top fragment always represents the method-level boundary. Although the top fragment does not represent a diagram element, it serves as the container for all the method-level fragments. In our tree-like data structure, the top fragment is always at the root and therefore it may never be nested inside another fragment. All the fragments at the top level of the method (i.e., not nested inside any other fragment) are children of the top fragment. For our running example, the ordered sequence of the top fragment's children includes the message fragment, the alt fragment, and the loop fragment. The top fragment is implemented by a TopFragment class which stores this information as an ordered sequence of fragments.

The *message* fragment type defines the simplest fragment type that represents a message sent from one object to another. In a sequence diagram, a message fragment would correspond to an arrow between the lifelines of two objects. As mentioned earlier, message fragments may not nest other fragments. However, they are themselves always nested inside another fragment. Although message fragments are the most primitive in their structure, they represent information that lies at the core of sequence diagrams. Message fragments are implemented by a CallFragment class

that stores information associated with a call such as the corresponding invocation expression and the compile-time target method for the call.

In our data structure, the *opt* fragment type corresponds to the opt fragment of the UML 2.0 notation introduced in Chapter 2. An opt fragment is used to represent optional behavior observed in the system. An opt fragment may be nested inside another non-message fragment and can also enclose any number of fragments. The implementation of opt fragments uses OptFragment class. This class stores the guarding condition of the opt as well as the ordered list of all the children nested inside the fragment.

Also introduced in Chapter 2 is the UML *alt* fragment, which is used to designate a choice of behavior from multiple alternatives. The UML alt fragment corresponds to the respectively named fragment in our data structure. Because of possibility of two or more alternative paths of execution, the alt fragment needs to store the sequence corresponding to each alternative separately. Consequently, this fragment stores a list of sequences, where each ordered sequence contains the fragments corresponding to the appropriate path of execution. Each one of the alternative sequences will be referred to as a *case*. Although the number of cases is not limited, it must be greater than one. An alt fragment containing just a single case is equivalent to an opt fragment. Each case may contain as many or as few nested fragments as the execution path dictates. In the running example presented in Figure 2.3, the `if` statement at line 5 defines two alternatives, where each one of the cases contains a single method call. The data structure for the running example presented in Figure 3.1 reflects this through an alt fragment with two distinct cases, each one containing a fragment sequence with the message fragment appropriate for the case. In our data structure,

alt fragments are implemented by an AltFragment class. Unlike all other non-message fragments (which contain just a single sequence of nested fragments) this class stores a list of fragment sequences, with each one representing one of the possible execution paths. For each one of the paths, the class also stores the corresponding guarding condition. Similarly to other fragments with the exception of the top fragment, each alt fragment is always nested inside another non-message fragment.

As mentioned in Chapter 2, the *break* fragment to UML 2.0 can represent a breaking scenario from the immediately surrounding fragment. However, this definition is overly restrictive and does not allow to represent some scenarios found in real code. For example, the Java language extends the notion of breaks by introducing labeled breaks. Labeled breaks represent a breaking scenario where the control is transferred to an enclosing labeled statement, which then immediately completes its execution. The enclosing statement could be any one of the surrounding statements, not necessarily the immediately surrounding statement.

Using the UML 2.0 definition of a break fragment, it becomes impossible to express scenarios utilizing the functionality of labeled breaks such as the one described above. In order to accommodate this situation, we propose an extension to the UML standards that generalizes the definition of the break fragment. The new definition expands the original concept and redefines the break fragment allowing it to break out of not only the immediately surrounding fragment, but also across multiple fragment boundaries. The UML notation can be easily modified to represent this more general scenario by labeling the corresponding enclosing fragments and using the label when displaying the break fragment. This generalization of the break fragments allows to represent the transfer of flow of control across one or multiple fragment boundaries.

Due to the representation benefits and ease of use, we have decided to use the generalized break fragment in our data structure. In order to make this possible, we store the flow of control information necessary to identify the successor of execution; this information is the outermost fragment from which the break exits. A break fragment may not break out of the top fragment. Instead, the method exit is represented using a return fragment, which will be described in Section 5.2. For that reason, top fragment may not be the immediate parent of a break fragment. In our running example, there is one explicit break fragment that terminates the execution of the loop when condition `e1!=null` evaluates to true. However, a break fragment is also used in our analysis to describe the implicit (or normal) loop exit. The implicit loop exit occurs when the guarding condition of the loop evaluates to false and the execution of the loop terminates. The loop exit can be either the very first or very last element in the loop's fragment sequence and must not have any nested fragments.

Due to this detail of our algorithm, the data structure for the running example presented in Figure 3.1 includes two break fragments, where BREAK_1 represents the normal loop exit and BREAK_2 represents the explicit loop exit.

In the data structure, the break fragments are implemented by a BreakFragment class. A break fragment is quite similar to an opt fragment because both of these fragments represent optional behavior. However, the difference between the two fragment types is more in the semantics than the representation. The opt fragment represents optional behavior after which the normal execution continues. The break fragment, on the other hand, represents the situation where once the contents of the fragment are executed, the execution continues at a point outside the boundary of the surrounding fragment. A break fragment also stores the guarding condition leading to execution of

the fragment contents. Because the break fragment can lead to exit from any of its enclosing fragments, with the exception of the top fragment, field BreaksOutOf in class BreakFragment stores the outermost fragment from which the break exits. For example, both break fragments in our running example lead to exit from $LOOP_1$, and we have BreaksOutOf($BREAK_1$)=$LOOP_1$ and BreaksOutOf($BREAK_2$)=$LOOP_1$. Similarly to other non-message fragments, a break fragment is always enclosed inside another fragment and internally may contain any number of nested fragments.

Chapter 2 also introduces *loop* fragments. The contents enclosed inside a loop fragment will be repeatedly executed until either the guarding condition evaluates to false, or the condition guarding some break fragment enclosed inside the loop evaluates to true. As discussed above, in our analysis the normal loop exit is represented by a break fragment. In case of either implicit or explicit break, the loop is terminated before the rest of its contents are executed. Therefore, both of these breaking scenarios can be classified as loop exits and represented as break fragments.

In our data structure, loop fragments are implemented by a LoopFragment class. Similarly to other fragments, the loop fragment implementation stores the ordered sequence of fragments that represents the fragments nested inside of the loop. A loop fragment is always nested inside another non-message fragment. An example of a loop fragment is presented in the data structure for the running example in Figure 3.1. The loop fragment contains in its fragment sequence the implicit break fragment followed by the opt and message fragments in that order.

The data structure described contains all the necessary building blocks for representing its contents as UML 2.0 sequence diagrams. This data representation, combined with the fragment construction algorithm described in Chapters 4 and 5,

and additional analyses to be implemented in the future, will eventually provide all the functionality necessary for complete reverse-engineering of sequence diagrams involving multiple methods. The design of such analyses is beyond the scope of this work.

CHAPTER 4

Control Flow Analysis - Basics

4.1 Phase I: Preprocessing

Our control flow analysis performs a traversal of the *control flow graph* (CFG) of the method under consideration. The analysis maps subgraphs of the CFG to the various interaction fragments described in Section 3.2. In order to make this possible, we define the concepts of *branch successor* and *loop successor* in Section 4.1.3 and Section 4.1.4, respectively. The definition of these concepts is based on common terminology defined in Section 4.1.1 and Section 4.1.2.

4.1.1 Control Flow Graphs and Post-dominators

A control flow graph (CFG) is a static representation of the flow of control of a procedure or a method. In a CFG, the nodes correspond to statements and the edges signify the possible flow of control between the nodes. Figure 4.1 illustrates the control flow graph for the running example. The CFG was chosen as input to our analysis due to the fact that the CFGs are language independent, which makes our analysis applicable to systems written in any programming language as long as the control-flow graph can be generated. Also, in the process of reverse-engineering, the

source code may not always be available. Since control-flow graphs may be generated from bytecode or object code, the reverse-engineering analysis may be performed even in the absence of source code.

For any given control-flow graph G, we assume that there exists exactly one *entry node*. An entry node of G is a node that does not have any predecessors. We also assume that G has at least one *exit node*, which is a node that does not have any successors. In the control-flow graph for the running example, node `1` is the entry node, and node `12` is the exit node. In Java programs, method exits correspond to either `return` or `throw` statements. Generally, a CFG may have multiple exits. However, for simplicity, this chapter considers control flow analysis of CFGs containing a single exit node which corresponds to a return statement. Chapter 5 builds upon the ideas described here and presents issues related to CFGs containing multiple exit nodes and paths leading to throwing of exceptions. For the purposes of the simplified analysis of this chapter, it is assumed that each CFG has a single entry node and a single exit node. Additionally, each CFG node must be reachable from the entry node and each node must reach the exit node.

Next, we describe the standard concept of *post-dominance*, which is necessary for identification of branch successors and loop successors. Consider a control-flow graph G containing a single exit node reachable from all nodes of G. *Node p post-dominates node n* if every path from n to the exit node must go through node p. A node does not post-dominate itself. *Node i immediately post-dominates node n* if i post-dominates n and any node n_1 such that $n_1 \neq n$ *and* $n_1 \neq i$ that post-dominates n also post-dominates i. For every node with the exception of the exit node, there exists a unique immediate post-dominator. The immediate post-dominance relationships can

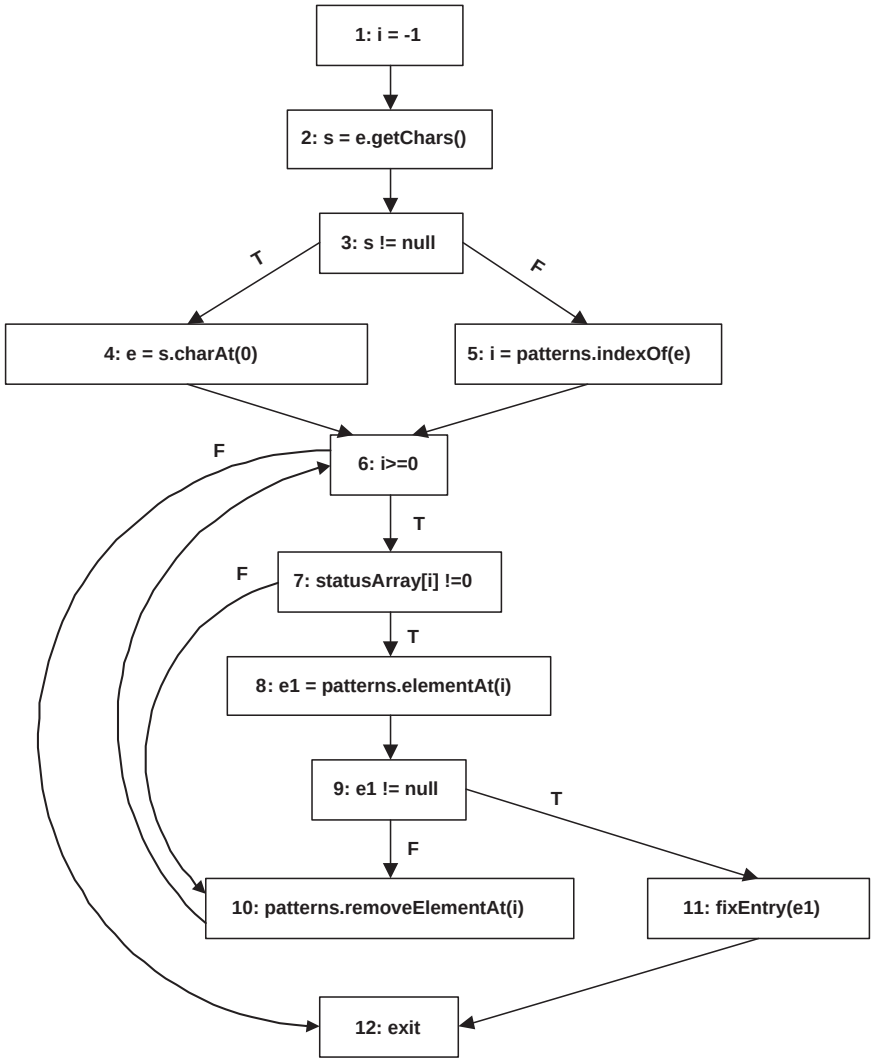

Figure 4.1: CFG for the running example

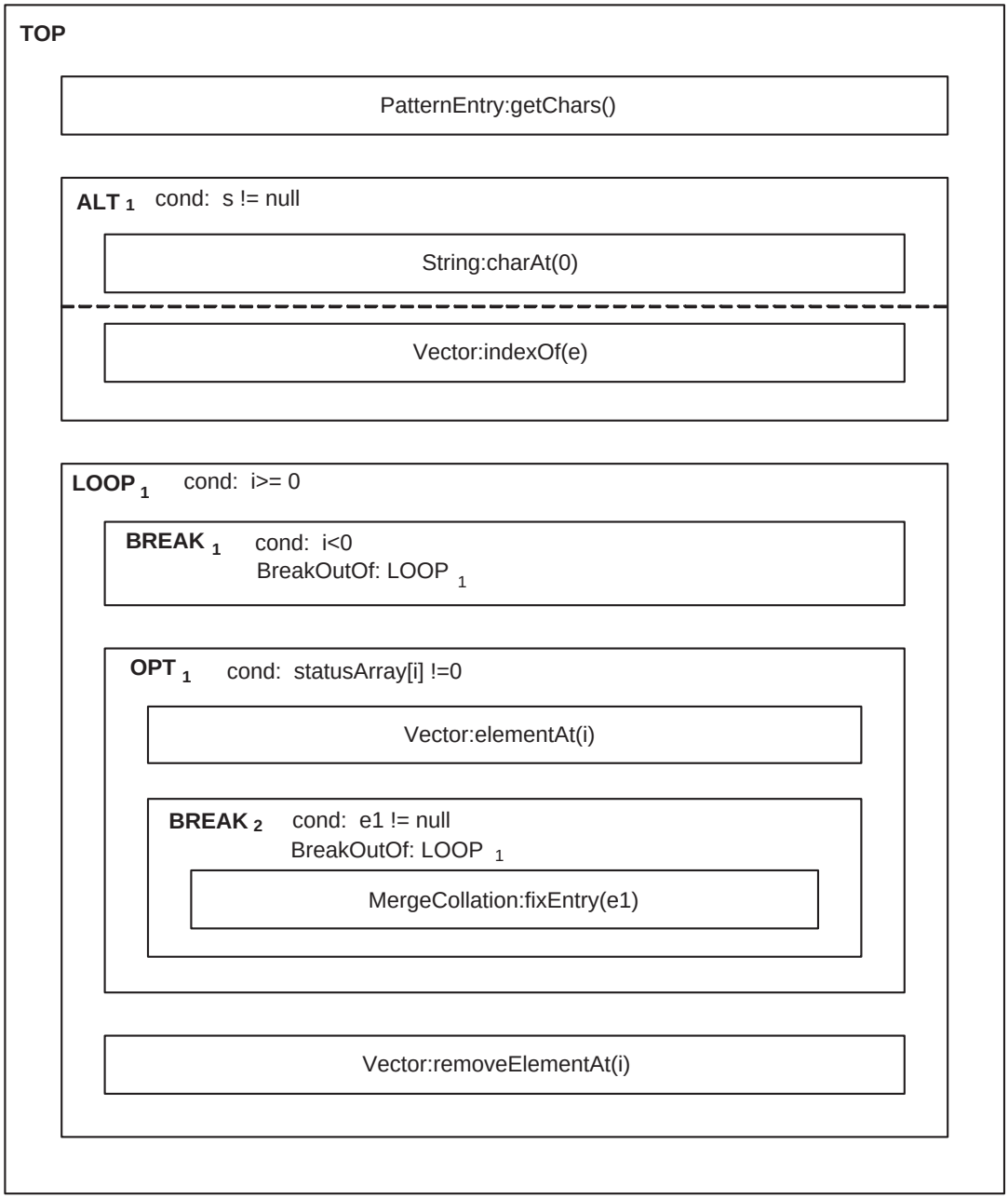

Figure 4.2: Fragment data structure for the running example, replica of Figure 3.1 for convenience

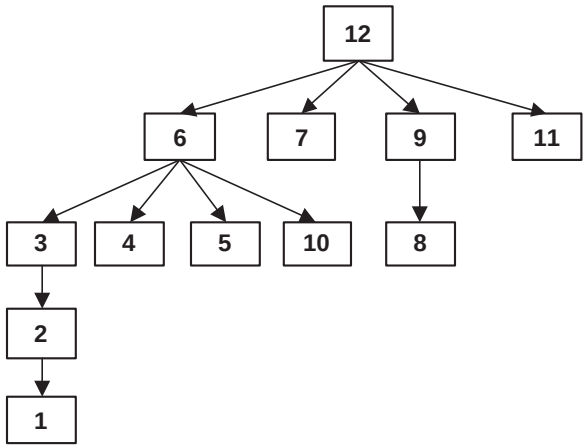

Figure 4.3: Post-dominator tree for the running example

be expressed in the form of a post-dominator tree where the exit node is the root of the tree and each parent node is the immediate post-dominator of its children.

Figure 4.3 represents the post-dominator tree for the running example. Node 12 (the exit node) is at the root of the tree because by definition the exit node post-dominates every other node in a control-flow graph. The immediate post-dominance relationship can be observed between any two nodes in the tree that have a parent-child relationship. For example, node 6 is the immediate post-dominator of node 4. There are various algorithms for computing post-dominator tree; our implementation uses the simpler of the two algorithms by Lengauer and Tarjan [8].

4.1.2 Loops

The Java programming language yields only *reducible* control-flow graphs, while some other programming languages allow non-reducible control flow structures. However, irreducibility is rare and its analysis is of secondary importance. For that reason, in our analysis, only reducible intra-method control-flow graphs are handled. The key

property of reducible CFGs is the absence of jumps into the middle of loops from the outside of the loop. A control-flow graph G is *reducible* if and only if its edges can be partitioned into two disjoint groups, the forward edges and the back edges. The forward edges form an acyclic graph in which every node can be reached from the entry node of G. An edge (x, y) is classified as a back edge iff node y is an ancestor of x in a depth-first spanning tree rooted at the entry node. The back edges can be identified using depth-first traversal of the CFG. In a reducible control-flow graph, a loop is a strongly connected subgraph L with the following properties:

- There is exactly one node $n \in L$ that has an incoming edge (n', n) and $n' \notin L$. In this case, node n is the header node for L, which will be denoted as *header(L)*.
- The set of nodes in L includes the exact set of CFG nodes that are reachable from *header(L)* and reach some node n' which is the source of a backedge $(n', header(L))$. This set of nodes is denoted as *body(L)*.

The definition of a header node implies that any given node can be the header node for at most one loop. Additionally, for any two loops L_1 and L_2, the sets *body(L_1)* and *body(L_2)* are either disjoint or one is a proper subset of the other. The second scenario represents the situation where one loop is nested inside the other loop.

In our implementation, we use the approach described in [1] to determine the loop structure of a reducible CFG. In order to identify loops and their nesting relationships from analysis of the CFG, it is first necessary to perform a depth-first traversal to identify the back edges. Targets of the back edges correspond to the loop header nodes. Next, for every loop L_i with the corresponding *header(L_i)*, it is necessary to determine the nodes belonging to the body of the loop. This process requires the

examination of all backedges whose target is *header(L_i)*. For a given header node h, consider all backedges *(n,h)*. Starting with n, we identify all nodes reaching n without going through h. Those nodes belong to *body(L)*. Additionally, node *header(L)* always belongs to the *body(L)*.

In addition to identifying all loops L_i, their *header(L_i)*, and *body(L_i)*, it is also necessary to identify the nesting relationships of the loops. A loop L_1 is said to be nested in a loop L_2 if there exists another loop L_2 such that *header(L_1)* $\neq$ *header(L_2)* and *header(L_1)* belongs to *body(L_2)*, yet *header(L_2)* does not belong to *body(L_1)*. For each loop, we identify additional information that is used in the analysis:

- Enclosing loop *encl_loop(L_i)* is the smallest loop $L_j \neq L_i$ such that *body(L_i)* $\subset$ *body(L_j)*. If L_i is a top-level loop, meaning it is not enclosed inside any other loop, then *encl_loop(L_i)=none*.

- For each CFG node n *encl_loop(n)* is defined as the smallest loop L_i such that $n \in$ *body(L_i)*. For n not enclosed inside any loop, *encl_loop(n)=none*.

4.1.3 Branch Successors

A CFG node is considered to be a *branch node* if it has two or more outgoing edges. During the analysis, either opt or alt fragment is created to represent the possible branching behavior. The details associated with the type of fragment created are described in Section 4.2. In both cases there is more than one possible alternative path of execution and at some "merge point" these paths come together and the execution continues. For a branch node n, the *branch successor* of n is the node from which the analysis will continue after processing the fragment created for n. From this point forward, the branch successor for node n will be denoted as *branch_succ(n)*.

An example of a branch node and its branch successor can be found in Figure 4.1, where node 3 has two outgoing edges. Both alternatives have distinct paths that come together at a merging point, in this case node 6. The node where the different alternative paths meet designates the end of the branching behavior and it is the starting point of the next fragment that will be constructed. Therefore, for branch node 3, *branch_succ(3)*=6. Using this information, our analysis will create an alt fragment for this branch node in Phase II of our analysis described in Section 4.2.

The identification of the branch successor varies slightly depending on whether or not the branch node is enclosed inside a loop. Consider a branch node n with outgoing edges *(n,n_i)* such that *encl_loop(n)=none*. Then, the branch successor for n is defined as the lowest common ancestor of all n_i in the post-dominator tree for the CFG. Generally, any common ancestor of nodes n_i corresponds to a merging point that each one of the nodes can reach in the CFG. The lowest common ancestor in the post-dominance tree is the merging point "closest" to n. For example, consider branch node 3, which has two outgoing edges (3,4) and (3,5) and its *encl_loop(3)=none*. By considering nodes 4 and 5 on the post-dominator tree depicted in Figure 4.3, it can be observed that node 6 is the lowest common ancestor of those nodes and therefore 6 is the branch successor for node 3.

The other possibility is that the branch node n is enclosed by a loop, i.e., *encl_loop(n)* $\neq$ *none*. For example, branch node 7 has two outgoing edges (7,8) and (7,10), and it is enclosed inside loop L with *header(L)* corresponding to the CFG node with the condition ($i >= 0$). In this case, the notion of the branch successor has meaning only in the context of flow of control inside the loop. For that reason, the method-level post-dominator relationships cannot be used to compute the branch successor. A loop

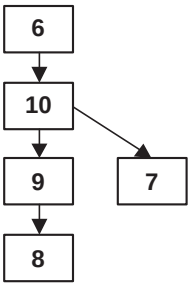

Figure 4.4: Post-dominator tree for loop L in the running example

may have any number of edges that lead to loop exit. These edges may include the normal loop exit as well as any number of exit edges resulting from breaks found inside the loop. Thus, these loop exit edges must not be considered when performing branch successor analysis for branch nodes inside a loop. To make that possible, it is first necessary to identify such edges for a loop L as *exit_edges(L)*. The set *exit_edges(L)* for a loop L is $\{(n_1,n_2) \mid n_1 \in body(L) \wedge n_2 \notin body(L)\}$. Each such edge e is associated with the loop from which it is breaking. Due to the presence of labeled `break` statements in Java, it is possible for a break edge to exit not only from the immediately surrounding loop, but also from any of its surrounding loops in cases where deep loop nesting is present. Therefore, an exit edge e belonging to set *exit_edges(L)* could be associated with both L and/or some loop enclosing L. For loop L in our example, *exit_edges(L)*=$\{(6,12),(9,11)\}$. For each loop exit edge e, we define *breaks_from(e)* as the largest loop L' such that e is in $exit\_edges(L')$. When determining branch successors for a branch enclosed in loop L, the edges in *exit_edges(L)* must be ignored. To formalize this, we introduce the notion of *post-dominance inside a loop*. Node n_2 post-dominates node n_1 inside loop L if every path from n_1 to *header(L)* that does not exit the loop goes through n_2. Using this definition, we can define the post-dominance

tree for loop L as the post-dominance tree for the subgraph containing the set of edges $\{(n,m) \mid n,m \in body(L)\}$, where *header(L)* is the subgraph's exit node. For any two nodes n_1,$n_2 \in body(L)$, if node n_2 post-dominates node n_1 in the loop, then during any iteration of L if n_1 is executed and eventually *header(L)* is reached, then n_2 is also reached after n_1 during the same iteration. The loop post-dominator tree for the loop in the running example is depicted in Figure 4.4. For a branch node n such that *encl_loop(n)*=L, only outgoing edges (n,n_i) that do not belong to *exit_edges(L)* are considered. This is the case because all edges belonging to *exit_edges(L)* break out of L and are represented as break fragments. If for a given node there are at least two outgoing edges that do not lead to an exit from L, then the branch successor for n is the lowest common ancestor of all such n_i in the post-dominator tree of L. If there is only a single edge $(n,n_i) \notin exit\_edges(L)$, then n is not considered as a branch node in reference to the flow of control in the loop and therefore there is no branch successor. The definition of branch successor for n ensures that the branch's merge point belongs to the same loop and the same loop iteration as n. Node 7 is an example of a branch node such that *encl_loop(7)* = L. The two outgoing edges for node 7 are (7,8) and (7,10). Both nodes 8 and 10 belong to *body(L)*. In order to determine *branch_succ(7)*, it is necessary to find the lowest common ancestor of nodes 8 and 10 on the post-dominator tree of loop L represented in Figure 4.4. Node 10 is the lowest common ancestor of nodes 8 and 10, and therefore *branch_succ(7)*=10.

4.1.4 Loop Successors

Loop fragments are created during our analysis when a loop header is encountered. In this section we use a more complicated example to illustrate the loop successor

computation. Figure 4.5 represents Java code for this example and Figure 4.6 shows the CFG corresponding to the code. From this point forward, the loops in the figure will be referred to as L_1, L_2, or L_3 corresponding to the outermost, middle, and innermost loop respectively. In order to correctly build the contents of the loop fragments and continue the analysis at the point following the loop, it is necessary to identify the node at which the construction of the next fragment should begin. Intuitively, the merge point of all the possible ways to exit a loop is the point where the construction of the fragment following the loop should begin. For example, for node 2 in Figure 4.6 the analysis will create a loop fragment corresponding to loop L_2. The fragment following the loop will be constructed starting from node 8. This node is the merge point of the three edges leading to the loop exit: (2,8), (3,8), and (6,8).

In order to identify the point following a loop in our fragment construction in Phase I we compute the *loop successor*, denoted by *loop_succ(L)*, for each loop L. First, we need to identify the set of edges for each loop L such that each edge e in the set breaks out of L, but e's target is in L's immediately surrounding loop, L'. These edges correspond to one of the following:

- Normal loop exit
- Regular `break` statements nested inside L, but not inside of one of L's nested loops
- A labeled `break` statement inside L or one of the loops enclosed by L such that the target of the label is L. Such `break` would terminate the execution

```
...
label1:
while(cond1)
{
  label2:
  while(cond2)
  {
    if (cond2_1)
      break;
    while (cond3)
    {
      if (cond3_1)
        break;
      if (cond3_2)
        break label2;
    }
    m1();
  }
  m2();
}
m3();
...
```

Figure 4.5: Java code illustrating the computation of loop successors

of L as well as the loops it may enclose and would continue the execution at *loop_succ(L)*

- A labeled `continue` statement inside L or one of the loops it encloses, where the target of the label is L'. This scenario terminates the iteration of L and continues with the execution of the next iteration of L'.

Figure 4.5 illustrates some of the possible types of edges described above. The set of loop exit edges whose target belongs to the body of the immediately surrounding loop is denoted as *jump(L)*. This set corresponds to the subset of exit edges e for L

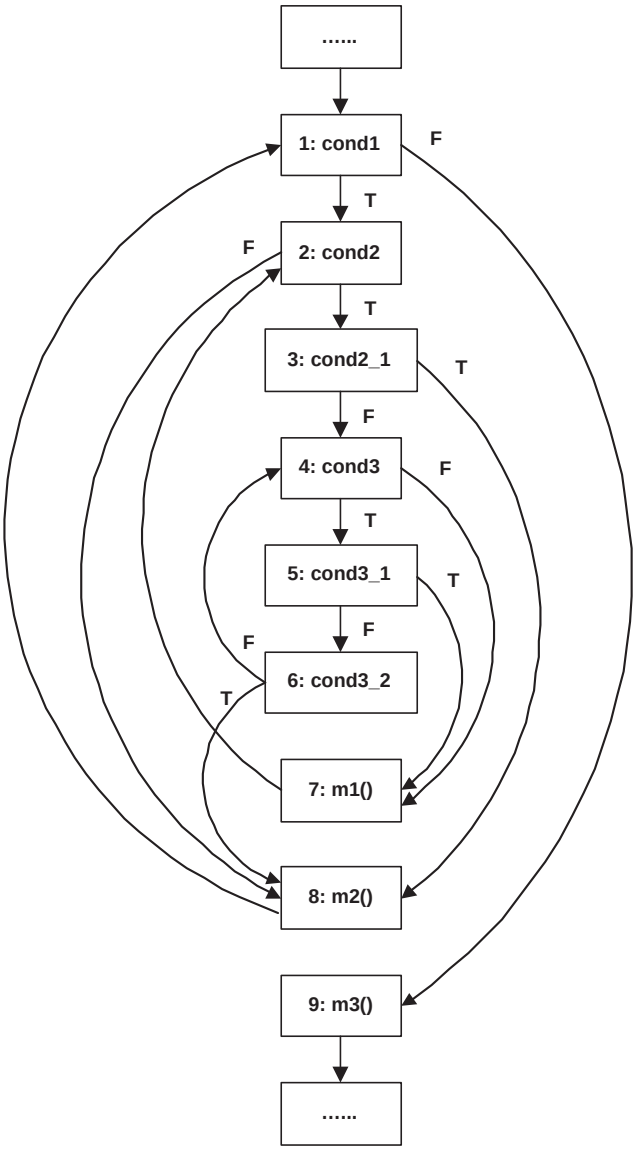

Figure 4.6: Control flow graph illustrating the computation of loop successors

```
header(L1)=1
encl_loop(L1)=none
exit_edges(L1)= {(1,9)}
jump(L1)= {(1,9)}

header(L2)=2
encl_loop(L2)=L1
exit_edges(L2)= {(2,8),(3,8),(6,8)}
jump(L2)= {(2,8),(3,8),(6,8)}

header(L3)=4
encl_loop(L3)=L2
exit_edges(L3)= {(4,7),(5,7),(6,8)}
jump(L3)= {(4,7),(5,7)}
```

Figure 4.7: Loop information for the three loops

such that *breaks_from(e)=L*. Only edges jumping to the immediately surrounding loop are considered because edges that cross boundaries of more than one loop represent the continuation of an iteration of some loop surrounding L one or more levels up and therefore are not of interest in the computation of L's loop successor. The set of jump edges for each one of the loops is listed in Figure 4.7. Note that for L_3 edge (6,8) jumps not only out of loop L_3, but also loop L_2 and therefore is included in L_3's set of exit edges, but not included in L_3's set of jump edges.

Similarly to the process of branch successor determination, two possible cases need to be considered when computing loop successors. One of the possibilities is that the loop L being considered is not nested inside another loop (i.e., *encl_loop(L)=none*). For this case, all the targets of edges in *jump(L)* are considered. The loop successor is determined by finding the lowest common ancestor for all such targets in the method-level post-dominator tree. In our example, L_1 is not enclosed by another loop and

the target of it jump exit edge is node 9. With just a single exit edge, the lowest common ancestor for node 9 in the method-level post-dominator tree is the node itself. Therefore, *loop_succ(L_1)*=9. If we consider the loop L in our running example, it also is not enclosed by another loop and its jump edges are (6,12) and (9,11). The targets of the two edges are nodes 11 and 12, whose lowest common ancestor in the post-dominator tree presented in Figure 4.3 is node 12. Therefore, *loop_succ(L)*=12.

The second possibility is that the loop is enclosed inside another loop (i.e., *encl_loop(L)*=L'). Similarly to branch successors, in this case only the flow of control that does not exit L' is considered in the loop successor computation. The loop successor represents the earliest possible merge point of all the paths leading to the exit from L such that the execution continues in the current iteration of L'. The loop successor for L is determined by considering the targets of all edges in *jump(L)*, and finding their lowest common ancestor in the post-dominator tree of L'. In our example, this case is illustrated by loop L_2. The targets of all the jump edges for this loop correspond to node 8. Therefore, *loop_succ(L_2)*=8. However, it is possible for the targets of edges in *jump(L)* to correspond to different nodes. In that case, the lowest common ancestor of those nodes in the loop post-dominator tree is identified as the loop successor.

In some rare cases, it is possible to have *jump(L)*= $\emptyset$. For example, this can happen in the case when the only way to exit out of a nested loop is through a labeled `break` that jumps across multiple loop boundaries. In such a case, *loop_succ(L)=none*. Although this case is highly unusual, it is handled by the fragment construction algorithm presented in the following section.

4.2 Phase II: Fragment Construction

The computation of branch and loop successors presented in the previous section provides the information necessary to perform the fragment construction by applying the algorithm presented in Figure 4.8 and Figure 4.9 to the control-flow graph of a method. The algorithm recursively traverses the control graph, creating interaction fragments corresponding to the control-flow structure. For example, during the traversal of the CFG corresponding to the running example, the algorithm would create a message fragment when it encounters method invocation *getChars()* on a *PatternEntry* object. The fragment structure produced by the algorithm for the running example is represented in Figure 4.2. This figure is a replica of Figure 3.1 and was duplicated in this chapter for convenience. Our algorithm for fragment construction is based on two important assumptions. One of the assumptions is that the control-flow graph for the method being analyzed is reducible. The notion of reducibility was discussed in detail in Section 4.1.2 and our analysis of loops is dependent on this assumption. The second assumption is that each control-flow graph node includes no more than one method invocation and that branch nodes do not correspond to method invocations. For example, method invocations where parameters themselves are calls such as *m1(m2())* need to be modified by introducing an auxiliary variable *aux* and two statments *aux=m2()* and *m3(aux)*. Similar strategy can be employed to branches containing method invocations. The second assumption can be trivially satisfied through such use of auxiliary variables and statements.

The fragment construction occurs in method *processSequence* where the control-flow graph is traversed starting from the *start* node. The traversal terminates when the *stop* node is encountered or when a CFG node with no successors is reached. To

illustrate this point, consider the running example. Initially, the method *processSequence* is called with the CFG entry node (i.e. node 1) as the *start* node and CFG exit node (i.e. node 12) as the *stop* node. The fragments corresponding to the method's flow of control are constructed as the algorithm traverses the CFG and identifies the subgraphs corresponding to the fragments. At some point during the traversal the algorithm will reach node 7 corresponding to a branch. To create the appropriate opt fragment, method *processSequence* will be invoked from method *processOpt* with *start*=7 and *stop*=10. The *stop* node here corresponds to *branch_succ(7)* determined in Phase I of our analysis (described in Section 4.1). The details of opt fragment construction are described later in this section. Each recursive call to *processSequence* corresponds to a new level of fragment nesting. Each fragment produced by the method is appended to the fragment sequence (*seq*) of the immediately surrounding fragment. The fragments at the top level belong to the fragment sequence of the top fragment. Each other fragment belongs to the fragment sequence of its immediately surrounding fragment, which could be either opt, alt, loop, or break fragment.

The main loop of the algorithm (lines 6-31) populates the sequence of the fragment under consideration with the appropriate nested fragments. When a new node n is encountered in the CFG traversal, it is first determined whether n belongs to a loop L' different from the currently enclosing loop L (L could be *none*). In case where *encl_loop(n)* $\neq L$, n must be the header node for loop L' that is either enclosed inside L or it is the top-level loop not nested inside another loop (when $L = none$). In order to create this loop fragment, the method *processLoop* is called. This method creates a new loop fragment and adds it to the current fragment sequence belonging to the fragment immediately enclosing the loop. The method then recursively populates the

contents of the new loop fragment by calling *processSequence* method with the nested loop's header node as the *start* node. As result of this processing, the new loop contains all the appropriate nested fragments is created. The algorithm then continues with the traversal starting from the successor of the newly created loop L'. Note that when *processSequence* is called to process the contents of a loop fragment, the *stop* node is not used. Instead, the traversal is stopped when the loop header node is reached via a backedge. This traversal termination is ensured by the algorithm through the checks performed at lines 11 and 30. During the CFG traversal for the running example, when node 6 is encountered as the next node n under consideration, it is determined that n=*header*($LOOP_1$) and that *encl_loop(start)=none* while *encl_loop*(n)=$LOOP_1$. In order to process the loop, *processLoop* is called, which in turn invokes the method *processSequence* with parameters *start*=6 and *stop=none*. Method *processSequence* then processes the CFG nodes belonging to the body of the loop and appends the resulting break and opt fragments to the loop's fragment sequence.

If it is determined that n is not a header a new loop, the next check at line 13 identifies whether n is a method call and therefore should be represented as a message fragment. A message fragment may not contain nested fragments and therefore it does not require recursive processing. Because a method call node may only have a single outgoing edge in a CFG by the assumptions stated earlier, the next node to be processed will be its successor, which is set at line 22 of the algorithm. In case of the running example, when method call *getChars()* is encountered, a new message fragment is created and appended to the sequence of the fragment immediately surrounding it, which in this case is the top fragment. Note that due to the presence of

polymorphism in object-oriented languages, it is often the case that there are several possible run-time targets for a method call. Consequently, during static analysis each method invocation could correspond to several polymorphic calls on different receiver objects. In the scope of this work, polymorphism is not addressed and each method invocation corresponds to a single message fragment.

For a node n enclosed inside a loop L, lines 16-19 of the algorithm identify its outgoing edges belonging to *exit_edges(L)*. These edges lead to the loop exit and therefore the algorithm will create corresponding break fragments. For each such edge (n, m), method *processBreak* is called, and a new break fragment is appended to the current fragment sequence. The break fragment is recursively built by calling *processSequence* with m as the *start* node. The *stop* node for processing of breaks is the loop successor for the loop L from which edge *(n,m)* breaks out (i.e. *breaks_from(n,m)*). For example, when outgoing edges for node 9 are analyzed, it is determined that edge (9,11) leads to exit from LOOP_1. In this case, *processBreak* is called in order to build the corresponding break fragment, which in turn invokes *processSequence* with *start=11* and *stop=12* since 12 is *loop_succ(*$LOOP_1$*)*.

Lines 20-30 of the algorithm process the remaining outgoing edges for n not leading to loop exits. At this point we also determine the next node at which the traversal will continue once the current fragment is built. Lines 23-25 of the algorithm address the creation of an opt fragment. If n has two outgoing edges and the target of at least one of the edges is *branch_succ(n)*, then optional behavior is encountered. This scenario is represented by an opt fragment which is created by *processOpt*. In this method, a new opt fragment is appended to the current fragment sequence and is populated with the contents of the path not leading directly to the *branch_succ(n)*.

It is possible for both outgoing paths for n to contain no control-flow information leading to the creation of fragments. Consequently, the fragment sequence of the opt fragment may be empty. Such an opt fragment would not contain any information valuable in the creation of a sequence diagram and therefore will be eliminated in Phase III of our analysis, which is described in Chapter 6. The opt fragment in the running example is created as the result of the analysis of node 7. For this node, the algorithm identifies that out of the two outgoing edges for this branch node, the edge (7,10) leads to the branch successor. In order to create the opt fragment, method *processOpt* is called, which in turn invokes *processSequence* with parameters *start=8* and *stop=10*, since $branch\_succ(7) = 10$.

In our analysis, the necessity to create an alt fragment is identified at lines 26-28 of the algorithm where the remaining edges not leading to loop exits are processed. If node n has two outgoing edges and the target of neither edge is *branch_succ(n)*, or if n has more than two outgoing edges, the alt fragment is created by calling *processAlt*. This method creates an alt fragment and processes each one of the outgoing edges for n. A separate fragment sequence corresponding to each *case* is created for each outgoing edge (n,m_i). Method *processSequence* recursively processes each *case* using the m_i as the *start* node and *branch_succ(n)* as the *stop* node. In the running example, branch node 3 leads to the creation of an alt fragment. Method *processAlt* creates a separate case for the two outgoing edges by calling *processSequence* twice with *start* corresponding to the target of the outgoing edges (3,4) and (3,5) and *stop* corresponding to *branch_succ(3)* (node 6).

An opt fragment can be considered as a special case of an alt fragment. The previously described process for the creation of an opt fragment is not a necessary part

of the algorithm. Instead, *processAlt* could be used to process the optional behavior with the resulting alt fragment having two cases with one them being empty. This special alt fragment could then be processed in Phase III (described in Section 6.1) and transformed into an opt fragment. Although both approaches yield the same result, the approach described above was chosen to explicitly identify opt fragments as part of the algorithm.

input Control-flow graph G and data described in Section 4.1
output Top fragment t constructed by *main*

proc *main*
[1] create empty top fragment t
[2] create empty fragment sequence s inside t
[3] *processSequence(s,G.start,G.exit)*

proc *processSequence(seq, start, stop)*
[4] L := *encl_loop(start)*
[5] n := *start*
[6] while $n \neq$ *stop* and $n \neq$ *none*
[7] if *encl_loop(n)* $\neq L$
[8] n must be header node of some loop L'
[9] *processLoop(seq,L')*
[10] n := *loop_succ(L')*
[11] if n = *start* then n := *none*
[12] continue with next iteration for [6]
[13] if n contains a call
[14] append a new message fragment to *seq*
[15] *breaks* := $\emptyset$
[16] if $L \neq$ *none*
[17] *breaks* := $\{m \mid (n,m) \in$ *exit_edges(L)* $\}$
[18] for each $m \in$ *breaks*
[19] *processBreak(seq,n,m)*
[20] *rest* := $\{m \mid (n,m) \in G \wedge m \notin$ *breaks*$\}$
[21] if *rest* = $\emptyset$ then *next* := *none*
[22] if *rest* = $\{m\}$ then *next* := m
[23] if *rest* = $\{m_1,m_2\}$ and (m_1 = *branch_succ(n)* or m_2 = *branch_succ(n)*)
[24] *processOpt(seq,n,rest)*
[25] *next* := *branch_succ(n)*
[26] else if *rest* = $\{m_1, \ldots, m_k\}$ for $k > 1$
[27] *processAlt(seq,n,rest)*
[28] *next* := *branch_succ(n)*
[29] n := *next*
[30] if n = *start* then n := *none*
[31] end while

Figure 4.8: Algorithm for fragment construction

proc *processLoop(seq,L)*
[32] append a new loop fragment f to *seq*
[33] create an empty internal fragment sequence seq_2 inside f
[34] *processSequence(seq_2,header(L),none)*

proc *processBreak(seq,n,m)*
[35] append a new break fragment f to *seq*
[36] create an empty internal fragment sequence seq_2 inside f
[37] L := *breaks_from((n,m))*
[38] *processSequence(seq_2,m,loop_succ(L))*

proc *processOpt(seq,n,rest)*
[39] append a new opt fragment f to *seq*
[40] create an empty internal fragment sequence seq_2 inside f
[41] for each $m_i \in rest$
[42] if $m_i \neq$ *branch_succ(n)*
[43] *processSequence(seq_2,m_i,branch_succ(n))*

proc *processAlt(seq,n,rest)*
[44] append a new alt fragment f to *seq*
[45] for each $m_i \in rest$
[46] add new alternative a_i to the alt fragment
[47] create an empty internal fragment sequence seq_i inside a_i
[48] *processSequence(seq_i,m_i,branch_succ(n))*

Figure 4.9: Algorithm for fragment construction

CHAPTER 5

Control Flow Analysis - Advanced Issues

5.1 UML Deficiencies

Although the second-generation UML standard provides a significant improvement in the expressive power of the language, the specification still presents some limitations in the representation of behaviors often found in existing systems. Particularly, UML 2.0 does not provide the explicit notation for representation of certain Java language features such as multiple method exits and exceptional behavior. Figure 5.2 and Figure 5.1 show an example of code illustrating these deficiencies throughout this chapter. This slightly modified code for method `read()` found in class `ZipInputStream` of the standard package `java.util.zip` represents code that can be found in real systems. By examining the code, it can be observed that there are multiple ways to terminate method execution. Three return statements and two throw statements in this method result in method exit. It is also not clear which one of the return statements represents the "normal" method exit. The inability of UML to express these language features is a problem for systems written not only in Java, but also other object-oriented languages. The addition of the notation for expressing this behavior in UML could prove to be beneficial in documenting system design.

```
public class ZipInputStream extend InflaterInputStream {
 public int read(byte[] b,int off, int len) throws IOException {..}
 private void readEnd(ZipEntry e) {..}
 private ZipEntry entry;
 private static final int STORED = ...;
 private static final int DEFLATED = ...;
 ...
}

public class InflaterInputStream extends FilterInputStream {
 public int read(byte[] b,int off, int len) throws IOException {..}
 private InputStream in;
 ...
}
public class ZipException extends Exception {
 public ZipException(String s) { ..  }
 ...
}
```

Figure 5.1: Sample classes based on standard package `java.util.zip`

```
public int read(byte[] b, int off, int len) throws IOException {
  if (entry == null)
    return -1;
  switch (entry.method) {
    case DEFLATED:
      len = super.read(b, off, len);
      if (len == -1)
        readEnd(entry);
      return len;
    case STORED:
      len = in.read(b, off, len);
      if (len == -1)
        throw new ZipException("unexpected EOF");
      return len;
    default:
      throw new InternalError("invalid compression method");
  }
}
```

Figure 5.2: Java code illustrating multiple returns and exceptional behavior

5.2 Proposed UML 2.0 Extensions

5.2.1 Notation for Multiple Returns

Similarly to other programming languages, Java allows for multiple exits from a method. In our analysis, each one of the returns is considered to be a "premature" method exit with the exception of a single "normal" exit. There is no methodology for systematically distinguishing between the "normal" exit and the "premature" exits. Instead, the distinction between the two can be performed by utilizing an arbitrary heuristic in the analysis. The particular heuristic used in the scope of this work is described shortly. An example of a premature exit as determined by our heuristic can be found in Figure 5.2. The return guarded by the condition `entry==null` is a premature return leading to premature termination of method execution.

Currently, there is no UML notation for representing premature method exits in sequence diagrams. Although it is possible to express the premature returns as break fragments breaking out of the method boundary, this representation would not follow the definition of breaks as described in [5]. Consequently, it would be beneficial to introduce new notation for accurate presentation of system design in the presence of multiple method exits. This could be accomplished by the addition of the new *return* fragment, making it possible to accurately represent the scenarios leading to premature method exits. The return fragment would be similar to opt or break fragments, since the execution of the contents of each one of those fragments depends on the value of the guarding condition. However, unlike other fragments, the return fragment would indicate termination of method execution and return to the caller. The proposed notation for return fragment is illustrated in Figure 5.3 and could be easily incorporated into the existing UML notation.

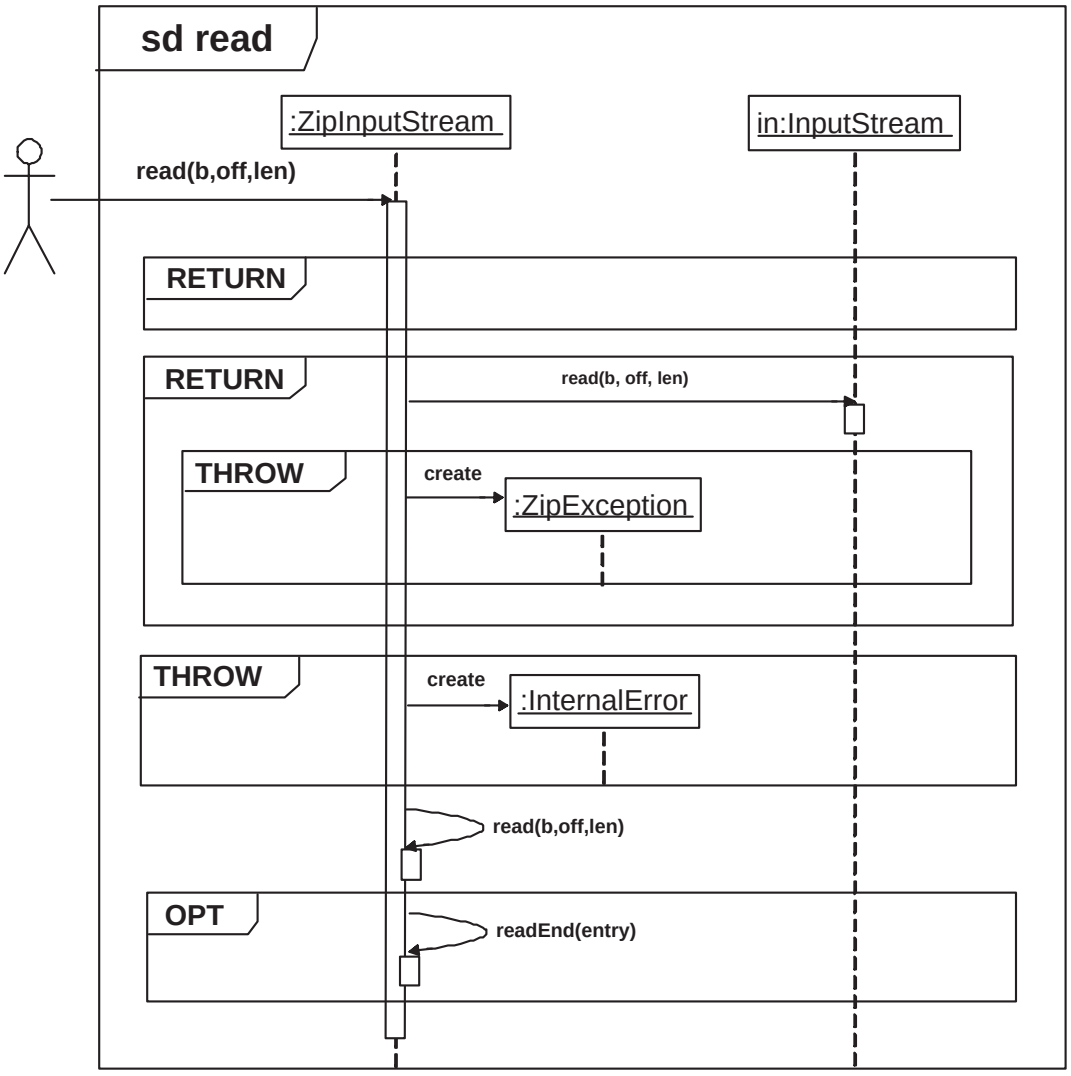

Figure 5.3: Sequence diagram illustrating the proposed UML extensions

5.2.2 Exceptional Behavior

The Java programming language also provides the exception handling mechanism for situations where a program either violates the semantic constraints of the language, or handles the consequences of exceptional conditions as defined by the programmer. When a program throws an exception, non-local transfer of control occurs from the point where the exception is thrown to the point where it is caught [5]. Representation of such exceptional behavior can be particularly difficult because in some cases a thrown exception could propagate several levels up the call chain prior to being caught. Figure 5.3 shows an example of `ZipException` being thrown, which occurs when the condition `len==-1` evaluates to true.

Currently, UML does not provide notation for representing the flow of exceptional behavior in sequence diagrams. Because of the challenges associated with the representation of exceptional behavior (in particular showing where the exceptions are caught), exceptional behavior is not represented in the scope of this work. However, with only minor changes, UML notation to show the throwing of exceptions in sequence diagrams could be introduced. A new *throw* fragment could be used to represent the behavior occurring between the point where the abnormal scenario is recognized and the point where the exception is thrown. Although this exception fragment is not part of our work, its introduction would require only trivial changes to our algorithm and data structure described in Chapters 3 and 4. The exception fragment shown in Figure 5.3 is an illustration of the possible UML representation for the throwing of exceptions.

5.3 Data Structure Additions

To accommodate the analysis and representation of multiple returns from a method in our analysis, return fragments were introduced to our data structure. A return fragment is similar to a break fragment, since both represent optional behavior that subsequently changes the flow of control. While the inclusion of this new fragment type in our analysis and data structure required only minor changes, the benefits of differentiation between breaks and returns in design documentation can be potentially significant. While break fragment indicates an exit from an enclosing fragment, return fragment denotes a method exit. We implement return fragments using a ReturnFragment class. Similarly to the break and opt fragments, the return fragment stores an ordered sequence of its nested fragments as well as the guarding condition. The return fragment is always nested inside another non-message fragment. Figure 5.4, based on example in Figure 5.2, provides an illustration of return fragments. One of the return fragments in the figure, RETURN_1, does not contain any nested fragments, while fragment RETURN_2 encloses a message fragment.

5.4 Phase I: Preprocessing

5.4.1 Post-dominance

In a control-flow graph containing multiple exit nodes such as the one presented in Figure 5.5, the notion of post-dominance is not defined. In order to compute post-dominance relationships within such a CFG, there must be a single exit node reachable from each node in the control-flow graph. In the presence of multiple method exits, it is necessary to add an artificial exit node that succeeds all other exit nodes. As a result of this minor change, all exit nodes in the CFG are post-dominated by this

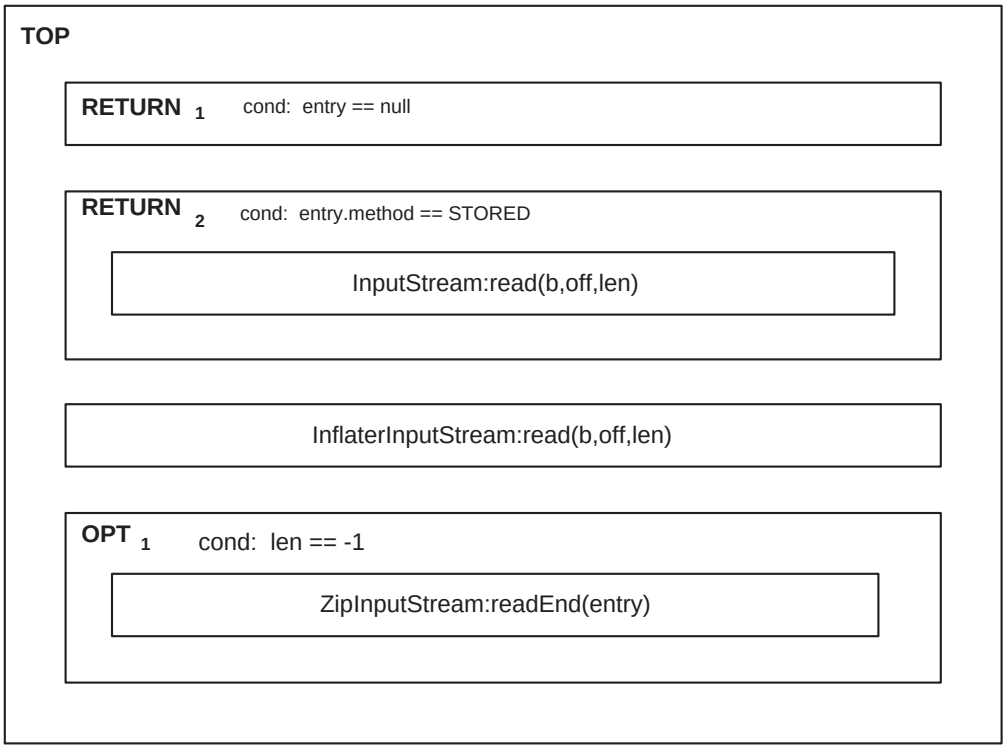

Figure 5.4: Fragment structure generated for method `read`

new exit node and the computation of the post-dominator tree can be performed in the same way as described in Section 4.1.1.

5.4.2 Control Dependence

To determine paths leading to "premature" exits in our analysis, it is necessary to identify the edges that once taken lead to the early exit. These controlling edges represent the decision point at a branch node where once such edge is taken, it leads to the method exit. Consider two nodes n_1 and n_2 of control-flow graph G. Node n_2 is *control-dependent* on n_1 if there exists a path from n_1 to n_2 such that n_2 post-dominates all the nodes in the path except n_1. By this standard definition, it is clear that n_1 must be a branch node. An edge (n_1, n_2) is a controlling edge for node n if both of the following requirements are satisfied:

- n is control-dependent on n_1
- n post-dominates n_2 or $n = n_2$

In order to identify controlling edges for an exit node x in CFG G, we traverse G searching for all nodes b such that $x = b$ or x *post-dominates* b. Then for each such node b we identify all edges (a, b) such that x *does not post-dominate* a. Each such edge is a controlling edge for the exit node x. This simple approach for identifying controlling edges is based on [4]. Figure 5.6 lists the sets of controlling edges for the exit nodes in the CFG shown in Figure 5.5. The set of controlling edges for a node n will be denoted by *contr_edges(n)*.

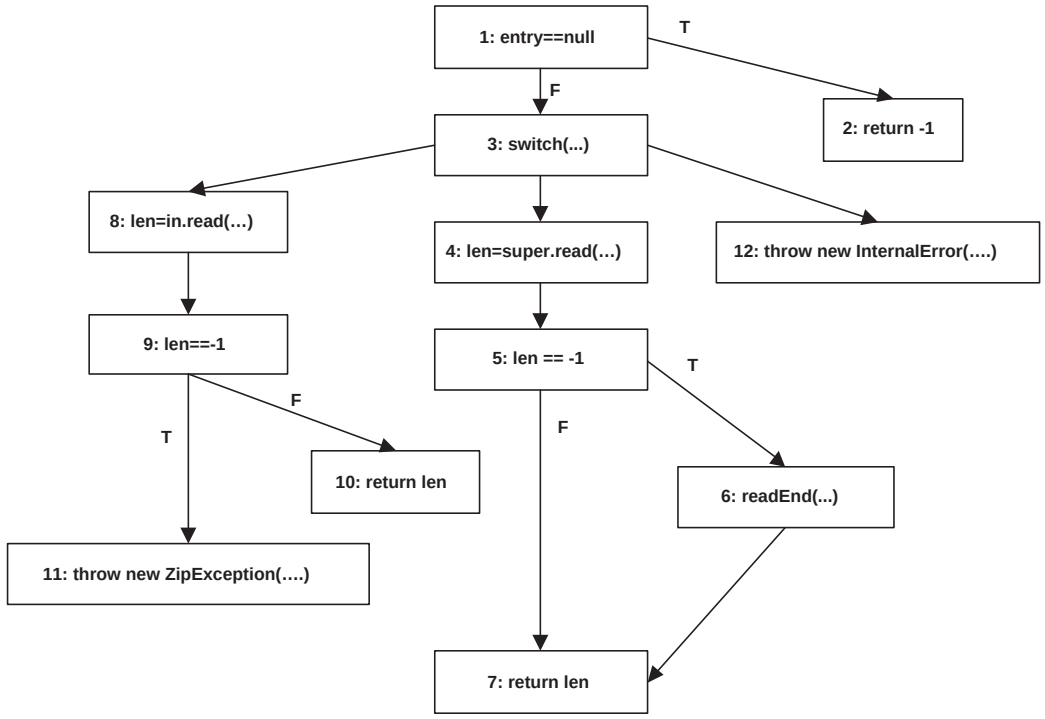

Figure 5.5: Control flow graph illustrating computation of control dependence

```
contr_edges(2)={(1,2)}
contr_edges(7)={(3,4)}
contr_edges(10)={(9,10)}
contr_edges(11)={(9,11)}
contr_edges(12)={(3,12)}

return_exit_edges= { (1,2),(3,8) }
throw_exit_edges= { (9,11),(3,12)}
```

Figure 5.6: Controlling edge information for method exits for CFG in Figure 5.5

5.4.3 Identifying Paths Leading to Exceptional Behavior

Our analysis does not consider exceptional behavior. Consequently, the paths leading to exceptional behavior do not contribute any control flow information in the analysis output: the control-flow subgraphs leading to an exception being thrown do not result in generation of fragments, and therefore these paths can be ignored. The controlling edges for `throw` statements are identified and stored in the set *throw_exit_edges*. This set of edges identifies the paths not resulting in fragment creation in the fragment construction phase of our analysis. This step is performed prior to processing of the returns (described in the next section) and the subsequent fragment construction, because the post-dominance relationships used there rely on the fact that paths leading to `throw` statements are ignored.

Next, we will describe the steps taken to eliminate each throw exit node from the CFG. These steps are followed for each such exit and result in a CFG with all paths leading to exceptional behavior eliminated for the purposes of our further analysis.

In order to distinguish a path leading to a `throw` method exit x, the controlling edges for each x must be identified using the algorithm described in Section 5.4.2. The controlling edges for x are then added to the set *throw_exit_edges* for the control-flow graph G. This process essentially eliminates the edges for the purposes of post-dominance relationships computation and the identification of subsequent controlling edges for the remaining throw statements and all premature returns statements. Note that the post-dominator information and controlling edges must be recomputed after the elimination of each one of the controlling edges leading to a throw exit.

The algorithm described above is be repeated for each throw exit node. As the result of this process, the controlling edges for `throw` statements are eliminated and

these exits become unreachable from the entry node of the control-flow graph. Essentially, the possible exceptional behavior of the code under analysis is eliminated. Figure 5.6 lists the controlling edges for throws exits ignored in our further analysis.

5.4.4 Processing of Multiple Method Exits

Once the controlling edges leading to throw statements are eliminated, it is necessary to identify the information needed for handling of multiple method exits. This information is used in Phase II of our analysis which is described in Section 5.5. There, return fragments are created in correspondence to "premature" method exits. It is often the case that a method may contain multiple method exits making it unclear which `return` statement should be designated as the normal exit node. Since there is no obvious way to determine the normal exit, a heuristic is applied to designate one of the exits as the normal exit, while others are assumed to be premature. In our approach, we identify the normal exit as the `return` statement in the CFG that is reached by greatest number of nodes containing method invocations. For each exit node, we traverse the CFG backwards counting the number of calls reaching the exit. The node with the maximum number of calls reaching it is designated as the "normal" exit. In case of a tie, the return node found later in the code is chosen as the normal return. In the CFG presented in Figure 5.5, node 7 would be identified as the normal exit because is it reached by two method calls. Return nodes 2 and 10 would be identified as premature exits because they are reached by 0 and 1 calls respectively.

Once the normal exit is identified, certain information necessary for processing of premature returns in the subsequent analysis is determined. This computation

uses the information regarding the controlling edges for throw statements. This is important because the computation of controlling edges for premature exits may differ depending on whether the paths leading to throw statements are considered. For example, consider Figure 5.5. If the edge leading to throw node 11 is ignored, the controlling edge for return node 10 is (3,8). However, if that path is considered, the controlling edge for node 10 is the edge (9,10). Since in our analysis the exceptional behavior is not considered, edge (3,8) should be controlling edge for the return node 10. For this reason, the processing described in Section 5.4.3 is a necessary preliminary step.

The computation of controlling edges for each premature return node requires the following steps. For each such return node, compute its controlling edge using the algorithm described in Section 5.4.2. Then, we add each such edge in the set *return_exit_nodes* associated with the CFG. Next, the post-dominator tree must be recomputed while ignoring the edges currently in *return_exit_nodes*. This is necessary because during each subsequent computation of controlling edges for the next return node, it must be ensured that those edges are identified properly. As result of the computations described, the information regarding controlling edges for premature exits is computed and stored in the set *return_exit_edges*. This information is then used in Phase II for return fragment construction.

5.5 Phase II: Fragment Construction

The changes described in the previous section effect the analysis of loops, loop successors, and branch successors. In a CFG with multiple returns and throw statements, these computations are based on the control-flow graph with the eliminated controlling edges for "premature" returns and for all throws.

The addition of the return fragment also requires modifications to the fragment construction algorithm described in Section 4.2. The modified algorithm is presented in Figure 5.7 and Figure 5.8 and it allows us to create return fragments whenever a controlling edge leading to a premature exit is encountered. Additionally, it handles the presence of `throw` statements by ignoring the paths leading to the exceptions being thrown. The line numbers that include modifications to the original algorithm are shown in bold.

At lines 15 and 16, sets *returns* and *throws* are identified. These sets correspond to outgoing edges for the current node n leading to return or throw exits respectively. At lines 17 and 18, for every outgoing edge for n leading to a premature exit, a new return fragment is created by calling method *processReturn*. There, the return fragment is added to the fragment sequence of the surrounding fragment and all nodes starting with node m are placed inside the return fragment. For example, consider the CFG for our example presented Figure 5.5. When the algorithm encounteres node 3, it identifies the edge (3,8) as the controlling edge for a premature return. Method *processReturn* is then called, which then invokes *processSequence* with $start = 8$ and $stop = none$. As result, nodes 8 and 9 will be processed. However, after the elimination of the path leading to the throw fragment, node 9 does not contain control-flow information of interest. Therefore, only the message fragment corresponding to

node 8 is enclosed in the new return fragment. The resulting return fragment is represented in the data structure in Figure 5.4.

Additional changes to the algorithm at line 21 ensure that edges leading to premature returns and throw statements are not processed inside the loop. The modification at line 24 excludes such edges from being counted as outgoing edges from the current node. Consequently, the newly created opt and alt fragments enclose proper fragments and the next node to be processed is identified correctly.

input Control-flow graph G and data described in Section 5.4
output Top fragment t constructed by *main*

proc *main*
[1] create empty top fragment t
[2] create empty fragment sequence s inside t
[3] *processSequence(s,G.start,G.exit)*
proc *processSequence(seq, start, stop)*
[4] L := *encl_loop(start)*
[5] n := *start*
[6] while $n \neq$ *stop* and $n \neq$ *none*
[7] if *encl_loop(n)* $\neq L$
[8] n must be header node of some loop L'
[9] *processLoop(seq,L')*
[10] n := *loop_succ(L')*
[11] if n = *start* then n := *none*
[12] continue with next iteration for [6]
[13] if n contains a call
[14] append a new message fragment to *seq*
[15] *returns:=* $\{m \mid (n,m) \in$ *return_exit_edges* $\}$
[16] *throws* := $\{m \mid (n,m) \in$ *throw_exit_edges* $\}$
[17] for each $m \in$ *returns*
[18] *processReturn(seq,m)*
[19] *breaks* := $\emptyset$
[20] if $L \neq$ *none*
[21] *breaks* := $\{m \mid (n,m) \in$ *exit_edges(L)* $\wedge$ $(n,m) \notin$ *returns* $\wedge$ $(n,m) \notin$ *throws* $\}$
[22] for each $m \in$ *breaks*
[23] *processBreak(seq,n,m)*
[24] *rest* := $\{m \mid (n,m) \in G \wedge m \notin$ *breaks* $\wedge$ $(n,m) \notin$ *returns* $\wedge$ $(n,m) \notin$ *throws* $\}$
[25] if *rest* = $\emptyset$ then *next* := *none*
[26] if *rest* = $\{m\}$ then *next* := m
[27] if *rest* = $\{m_1,m_2\}$ and (m_1 = *branch_succ(n)* or m_2 = *branch_succ(n)*)
[28] *processOpt(seq,n,rest)*
[29] *next* := *branch_succ(n)*
[30] else if *rest* = $\{m_1, \ldots, m_k\}$ for $k > 1$
[31] *processAlt(seq,n,rest)*
[32] *next* := *branch_succ(n)*
[33] n := *next*
[34] if n = *start* then n := *none*
[35] end while

Figure 5.7: Algorithm for fragment construction

proc *processLoop(seq,L)*
[36] append a new loop fragment f to *seq*
[37] create an empty internal fragment sequence seq_2 inside f
[38] *processSequence(seq_2,header(L),none)*

proc *processBreak(seq,n,m)*
[39] append a new break fragment f to *seq*
[40] create an empty internal fragment sequence seq_2 inside f
[41] L := *breaks_from((n,m))*
[42] *processSequence(seq_2,m,loop_succ(L))*

proc *processOpt(seq,n,rest)*
[43] append a new opt fragment f to *seq*
[44] create an empty internal fragment sequence seq_2 inside f
[45] for each $m_i \in rest$
[46] if $m_i \neq$ *branch_succ(n)*
[47] *processSequence(seq_2,m_i,branch_succ(n))*

proc *processAlt(seq,n,rest)*
[48] append a new alt fragment f to *seq*
[49] for each $m_i \in rest$
[50] add new alternative a_i to the alt fragment
[51] create an empty internal fragment sequence seq_i inside a_i
[52] *processSequence(seq_i,m_i,branch_succ(n))*

proc *processReturn(seq,m)*
[53] append a new return fragment f to *seq*
[54] create an empty internal fragment sequence seq_2 inside f
[55] *processSequence(seq_2,m,none)*

Figure 5.8: Algorithm for fragment construction

CHAPTER 6

Phase III: Fragment Transformations

The fragment structure produced as the result of Phase II of the analysis may not be the optimal representation for a given method. Redundant information or unnecessary nested fragments may make the resulting diagrams less readable and harder to understand. The *fragment transformations* introduced in this chapter were developed to improve the fragment structure resulting from the previous phases of the analysis. The purpose of these transformations is to simply the structure without altering its meaning. There are two main goals that are achieved by the transformations. First, the *clean-up tranformations* eliminate any redundancies present in the output of Phase II. Next, the *readability transformations* further simplify the fragment structure with the goal of reducing nesting of the fragments. Such reduction of nesting is intended to help with readability and comprehension of the reverse-engineered sequence diagrams.

The *clean-up* transformations, described in Section 6.1, eliminate any fragments not contributing meaningful information to the diagram. For example, an empty opt fragment does not provide any control-flow information in a sequence diagram due to lack of message exchanges in the context of the guarding condition of such a fragment. The clean-up transformations can be considered as a post-processing

phase of the algorithm that eliminates any redundant information from the fragment structure.

The *readability* transformations reduce the nesting of fragments by employing various techniques described in Section 6.2 to move the nested fragments one level up in the fragment structure. These techniques have proven to be successful in reducing nesting of fragments as described in Chapter 7. All the transformations in this chapter are semantics-preserving and our experimental results suggest that they could be very beneficial in improving readability and understanding of sequence diagrams.

The transformations are performed in the following sequence. First, the clean-up transformations are performed in the order in which they are described. If during any one of those transformations the fragment structure is changed, the entire sequence of transformations is repeated. This process continues until there are no longer any changes in the fragment structure. This concludes the clean-up transformation phase. Next, the readability transformations are performed in the order in which they are described later. The process is also iterative. If any one of the transformations causes a change in the fragment structure, the entire sequence of transformations is repeated. This iterative process continues until there are no changes in the fragments structure. All clean-up and readability transformations are described in detail below.

6.1 Clean-up Transformations

6.1.1 Removal of Empty Alt, Opt, and Loop Fragments

In the process of fragment construction, the algorithm often creates empty fragments that do not contribute any control flow information to sequence diagrams. The transformation described here removes some of the empty fragments generated

by the algorithm by recursively traversing nested fragment sequences. Top and message fragments are never removed. Alt, opt, and loop fragments are removed only if their sequences do not enclose other fragments. Loop fragments initially always enclose at least a single nested fragment, the implicit break fragment. However, if the break fragment itself does not enclose any fragments, in some cases it will be removed as a consequence of the the break removal transformation described next. If upon the removal of the break fragment the loop's fragment sequence becomes empty, the loop is removed as well. An alt fragment can be deleted only if all of its cases contain empty fragment sequences. If any one case of an alt is empty, the case will be removed as part of this transformation.

The recursive and iterative manner in which all the transformations are executed makes it possible for fragments at the upper level of the data structure to become empty as the empty nested fragments at the lower nesting levels are removed. Figure 6.1 provides illustration for such a scenario and this transformation in general. Note that when the fragment OPT_3 is removed, the fragment sequence of OPT_2 becomes empty making it a candidate for removal in the next iteration of the transformations. As seen from the example, this transformation could greatly simplify a seemingly complex data structure.

6.1.2 Removal of Implicit Break Fragments

The data structure generated by the algorithm creates loop fragments that contain not only the explicit, but also the implicit (normal) loop breaking scenarios represented in the form of break fragments. Break fragments can occur anywhere within the body of the loop and may signify a scenario that results in breaking out

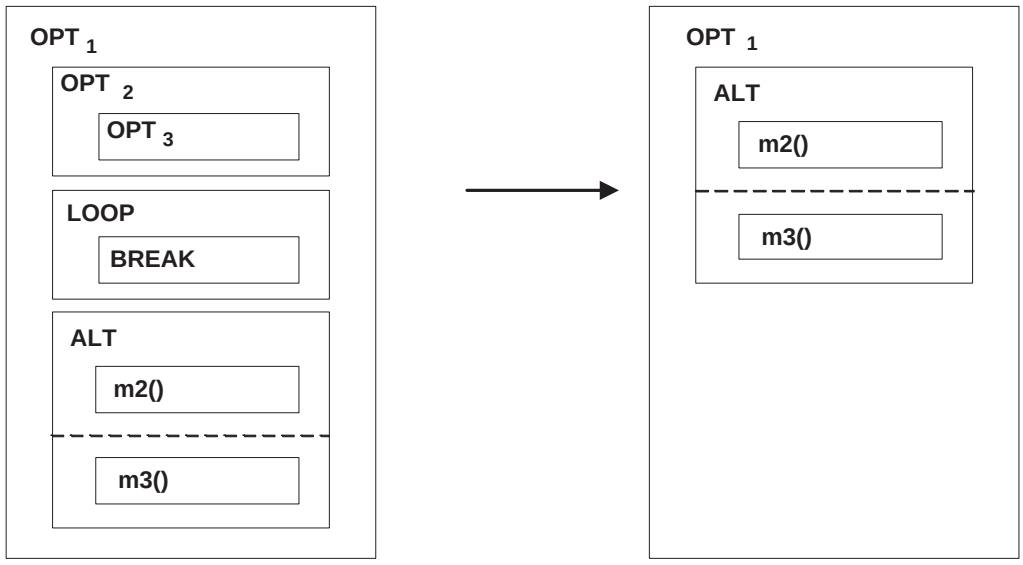

Figure 6.1: Illustration of the removal of empty fragments

of the current loop to one of the surrounding loops. The breaking scenario resulting in a jump over several loops is made possible, for example, by the labeled `break` and labeled `continue` statements in Java.

The transformation presented here considers break fragments that:

- Do not contain nested fragments.
- Are located as the first or last elements in the loop fragment's fragment sequence
- Exit out of the current loop and continue the execution at the point following the loop in the immediately surrounding fragment. This means that for the loop exit edge e that corresponds to the break fragment, $breaks\_from(e) = L$, where L is the immediately surrounding loop of the break fragment under consideration.

Such breaks can be categorized as implicit breaks because they result in the exit from the loop in the beginning or the end of a loop's iteration and therefore do not represent a breaking scenario where only part of the loop body is executed prior to the loop exit. Additionally, since we only consider breaks containing no other nested fragments, the execution of the contents of the break does not represent any additional flow of control information and therefore is not significant. An illustration of such a break can be found in Figure 3.1 for our running example. In this figure, BREAK_1 is an implicit break for LOOP_1.

Because the breaks satisfying the criteria described above do not contribute any meaningful information in terms of flow of control, they can be eliminated. To do so, the data structure for each method is recursively traversed in search of loop fragments. Once a loop fragment is encountered, all the break fragments satisfying the criteria for removal are eliminated. The loop fragment is analyzed repeatedly to ensure that once the breaks in the beginning and the end of the sequence are removed, there are no other breaks eligible for removal that appear in those positions. This transformation guarantees that the break fragments satisfying the conditions described above will not be found in any analyzed loop fragment. Figure 6.2 provides a demonstration of the result of this transformation on a sample fragment structure where fragments BREAK_1 and BREAK_3 are removed.

6.1.3 Replacing an Alt Fragment with an Opt Fragment

As the result of the analysis in Phase II, the fragment structure may contain an alt fragment with only a single case. However, an alt fragment containing a single case represents an optional behavior that occurs only when its guarding condition

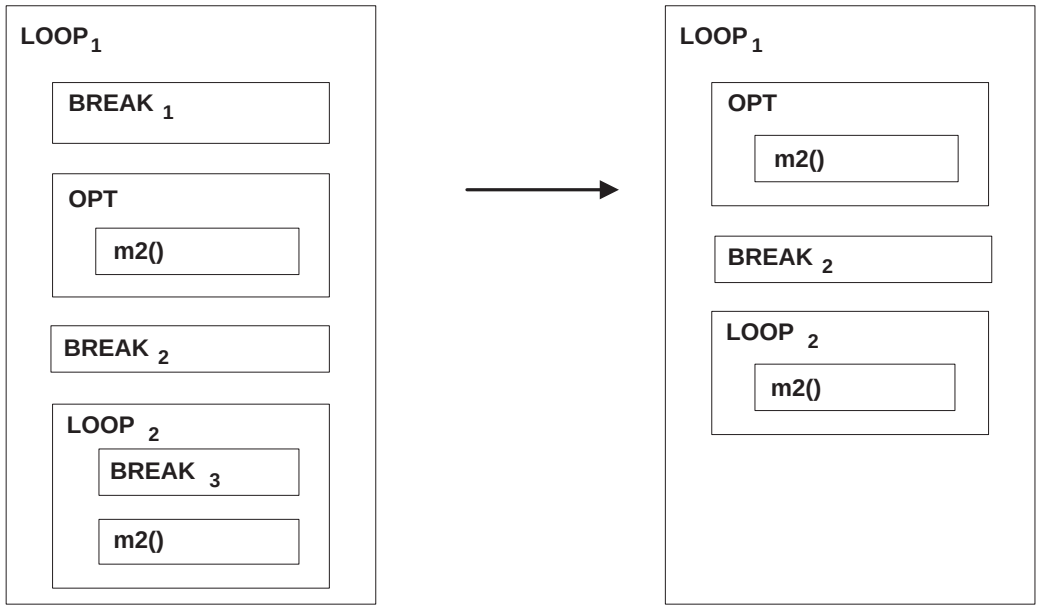

Figure 6.2: Illustration of the removal of implicit break fragments

of the case evaluates to true. Such fragment is semantically equivalent to an opt fragment with its guarding condition corresponding to the guarding condition of the alt fragment's case. An example of this transformation is represented in Figure 6.3, where the alt fragment containing a single case is replaced with an opt fragment. The corresponding opt fragment has the appropriate guarding condition and contains all the fragments that were nested inside the case.

6.2 Readability Transformations

6.2.1 Moving of Nested Alt Cases

This transformation applies when a case of an alt fragment contains a single nested element, which also happens to be an alt fragment. A case of the nested alt is

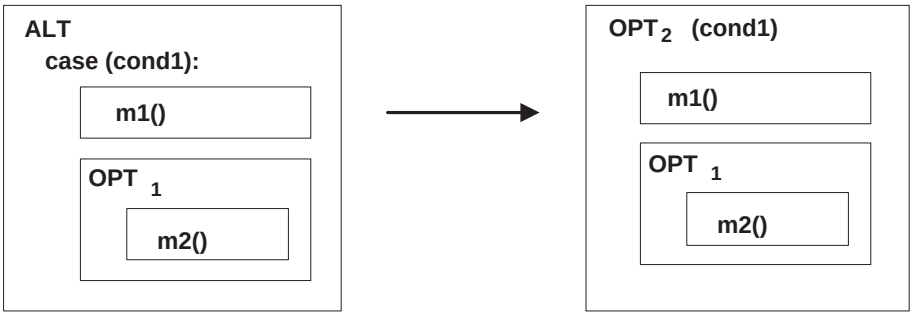

Figure 6.3: Replacing an alt fragment with an opt fragment

executed only when both the case-guarding condition of the surrounding alt and the appropriate case-condition of the nested alt evaluate to true. In the example presented in Figure 6.4, in order for the contents of the case with condition `x` to execute, both conditions `b` and `x` must evaluate to true. Therefore, it would be semantically equivalent to combine the two conditions and replace the original guarding condition of the nested case with the new condition. Additionally, since the two conditions of the nested and the outer case can be combined, the need for the guarding condition of the outer case is eliminated. As a result, the cases of the nested alt can become the cases of the outer alt fragment. Consequently, the nested alt fragment is eliminated as well as the case of the outer alt fragment containing it. Instead, the cases originally contained in the nested alt fragment are pushed up to the level of the surrounding alt and the modified guarding conditions ensure that the semantics are preserved.

6.2.2 Moving of Fragments Surrounded by an Opt Fragment

This transformation reduces nesting of a fragment by identifying opt fragments containing exclusively opt, alt, break, and return fragments. The transformation eliminates the surrounding opt fragment while pushing the fragments nested inside it

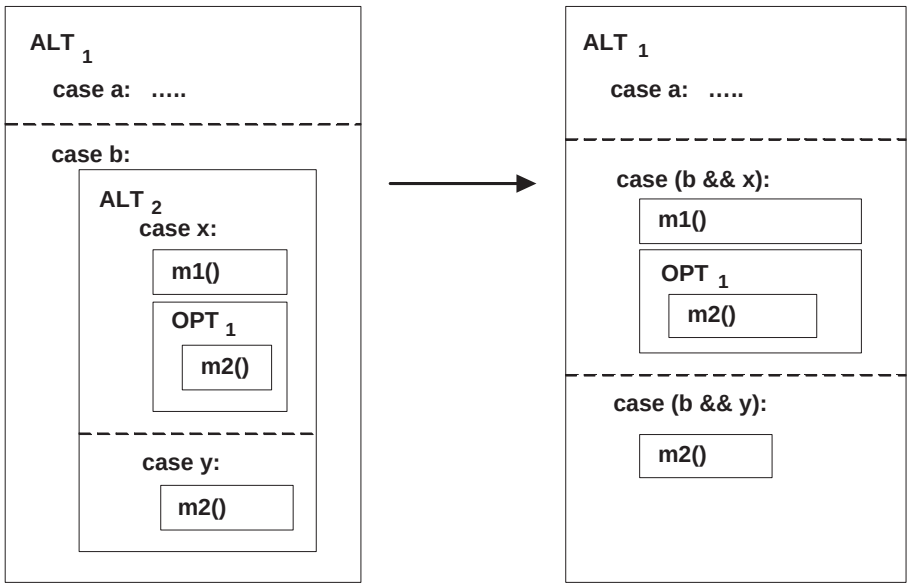

Figure 6.4: Moving of nested alt cases

one level up. This transformation combines the guarding condition of the surrounding opt with the conditions of each one of the nested opt, alt, return, and break fragments. As a result, the guarding condition of each one of the nested fragments accurately describes the circumstances under which the code inside of it will be executed. The merging of conditions of the opt and its nested fragments eliminates the necessity to have the opt fragment. Once the opt fragment is removed, all the alt, opt, break and return fragments are moved up in the nesting structure, and become enclosed in the fragment previously surrounding the original opt fragment. The nesting level of the fragments is reduced as the result. Since this transformation is performed recursively, the nesting level of many fragments can be significantly reduced. Figure 6.5 provides an illustration of this transformation.

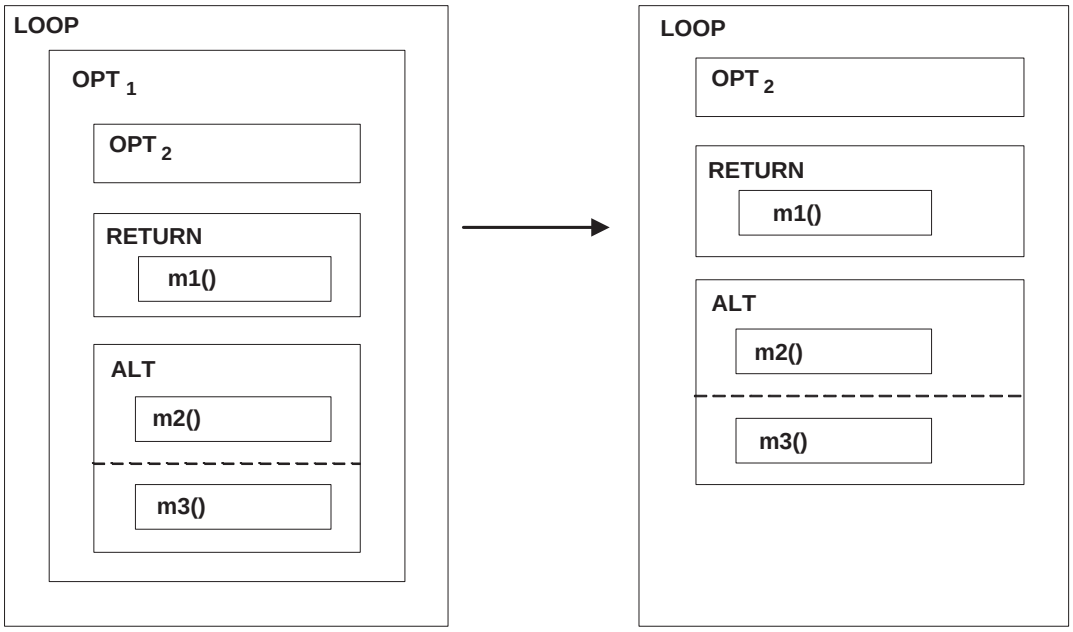

Figure 6.5: Moving of fragments surrounded by an opt fragment

6.2.3 Removing an Opt Fragment Enclosed by a Case

This transformation applies exclusively to alt fragments containing one or more cases enclosing only a single opt fragment. In this situation, the fragments enclosed by the opt fragment are guarded by the conditions of the opt fragment and the condition of the corresponding alt case. Therefore, it is possible to combine the conditions of the opt fragment and the case of the alt into a single condition. This new condition becomes the guarding condition of the alt case being transformed and the nested fragments of the original opt fragment become enclosed by the alt case. The completion of this transformation decreases the nesting level of the alt fragment cases containing a single opt fragment. The transformation is illustrated in Figure 6.6.

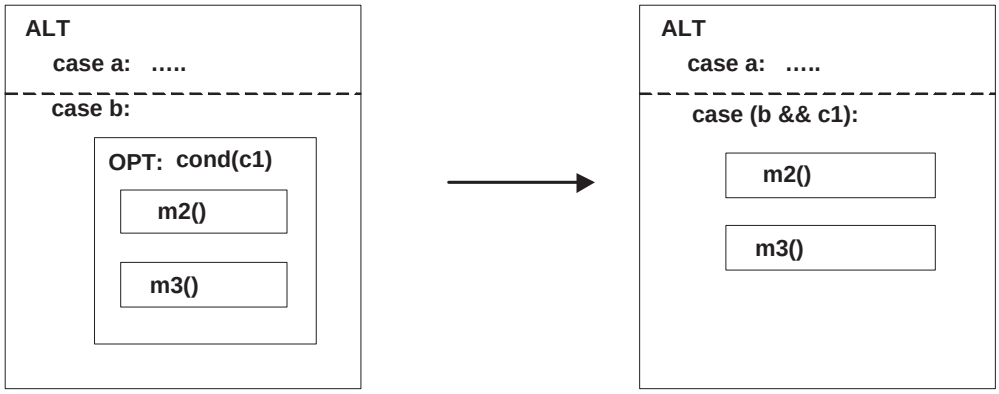

Figure 6.6: Removing an opt fragment enclosed by a case

6.2.4 Generalized Removal of Opt Fragment

This transformation reduces nesting of a fragment by identifying opt fragments containing opt, alt, break, and return fragments as well as a single subsequence of loop and message fragments. Similarly to other transformations described earlier, this transformation uses the technique of combining the guarding condition of the surrounding opt with the conditions of each one of the nested opt, alt, return, and break fragments. In order to preserve the semantics, it is also necessary to ensure that the subsequence of loop and message fragments only occurs under the guarding condition of the opt. Therefore, it is essential to enclose this subsequence by a newly created opt fragment with the guarding condition of the original surrounding opt fragment. The newly created opt fragment contains the loop and message fragments and replaces the original subsequence of loop and message fragments in the fragment sequence. As result of the transformation, all the alt, opt, break and return fragments are moved up in the nesting structure, becoming enclosed in the fragment

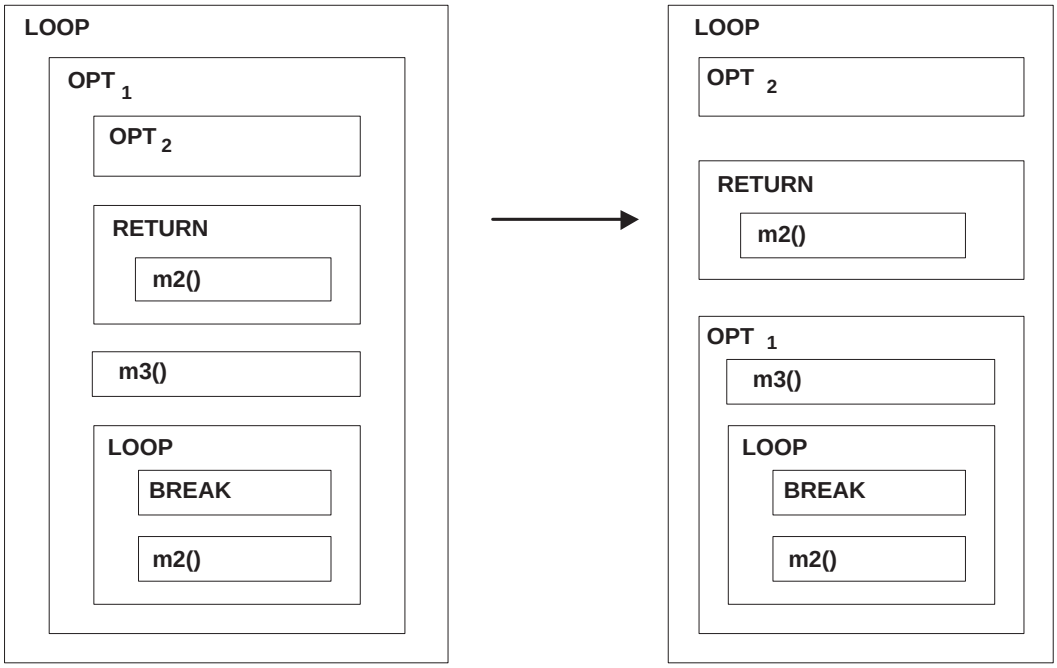

Figure 6.7: Generalized removal of opt fragments

previously surrounding the original opt fragment. The original loop and message fragment subsequence becomes enclosed inside an opt fragment, which is then placed in the appropriate location in the sequence of opt, alt, break, and return fragments. This transformation reduces the nesting level of the opt, alt, break, and return fragments, while keeping the nesting level of the loop and message fragments the same. Figure 6.7 provides an illustration this transformation.

CHAPTER 7

Empirical Study

In this chapter we present the experimental results from our evaluation of the control-flow analysis, which was implemented using the Soot framework [14]. Our experience suggests that as the nesting of non-message fragments increases, the comprehension of the resulting diagram becomes more difficult. Therefore, the reduction of the nesting in the diagrams could lead to significant improvements in its comprehension and readability. The goal of this study is to evaluate the efficiency of our analysis, to determine the prevalence of nesting in existing components and to identify the potential benefits of the readability transformations described in Section 6.2 in reducing the fragment nesting.

The set of components considered in this study is described in Table 7.1. These components typically come from reusable libraries. The components will be referred to as the components under analysis (CUA) from this point forward. Given a CUA, Phase I of the analysis is executed and the information necessary for the next phase is stored. Next, Phase II (i.e., fragment construction) is performed to generate the fragment structure. The redundant information present in the resulting structure is eliminated by performing the clean-up transformations described in Section 6.1. The

final step of our analysis is the execution of the readability transformations described in Section 6.2 and the assessment of their benefits in terms of nesting reduction.

(1) Component	(2) Classes	(3) Methods	(4) Time(s)	(5) Method Nesting		
				(a)	(b)	(c)
`collator`	12	157	4.84	56.10%	17.80%	26.10%
`date`	7	136	5.43	82.40%	5.10%	12.50%
`decimal`	7	136	0.77	81.60%	6.60%	11.80%
`message`	9	176	1.33	77.30%	5.70%	17.00%
`boundaries`	12	74	0.54	81.10%	13.50%	5.40%
`gzip`	6	41	0.21	68.30%	17.10%	14.60%
`zip`	14	118	0.54	72.00%	21.20%	6.80%
`math`	8	241	0.96	50.60%	33.20%	16.20%
`pdf`	24	330	0.74	78.20%	7.90%	13.90%
`mindbright`	60	488	2.08	69.10%	19.70%	11.30%
`sql`	18	60	0.32	63.30%	16.70%	20%
`html`	30	298	1.42	62.40%	18.50%	19.10%
`jess`	146	627	2.83	69.90%	8.50%	21.70%
`io`	21	86	0.34	74.40%	10.50%	15.10%
`jflex`	34	313	14.65	52.70%	21.70%	25.60%
`bytecode`	44	625	6.65	60.20%	19.20%	20.60%
`checked`	4	15	0.11	80%	13.30%	6.70%
`big`	1	33	0.24	57.60%	30.30%	12.10%
`vector`	4	38	0.18	60.50%	28.90%	10.50%
`cal`	6	152	0.87	67.80%	23.70%	8.60%
`push`	2	20	0.14	75%	20%	5%

Table 7.1: Analyzed components.

The details regarding the components used in this study are described in Table 7.1. Columns (2) and (3) provide the number of classes and methods in the CUA respectively. Column (4) presents the running time of the analysis in seconds for each respective component. This time includes all three phases of the analysis executed on a 900MHz Sun Fire 280-R machine. The maximum running time of 14.65 seconds

suggests that the cost of the analysis is practical. The performance of some of aspects of the current analysis implementation could be improved and the elimination of such inefficiencies could lead to even faster running times in the future.

For the purposes of this study, we classify each method into one of the following disjoint categories:

(a) Methods with either no fragments or only message fragments, such as simple get/set methods. The corresponding top fragment encloses either no fragments or only message fragments.

(b) Methods that have message fragments, but do not have any fragment nesting. In the corresponding fragment structure, only the top fragment would have nested non-message fragments.

(c) Methods with fragment nesting.

The last three columns in Table 7.1 show the percentage of methods in each one of these categories relative to the total number of methods. As the numbers demonstrate, the majority of the methods found in these components belongs to the first category, which is consistent with the typical object-oriented programming style. However, fragments are typically found in 20-50% of methods and many of those methods exhibit fragment nesting. We denote the set of methods in the third category as *Nested*. The goal of the readability transformations is to reduce fragment nesting. Since only fragments in methods belonging to *Nested* may be enclosed by other fragments, the readability transformations are only applicable to this category of methods.

We transform the methods in *Nested* using the transformations described in Section 6.2. The benefits of these transformations are analyzed using various metrics that evaluate the reduction of nesting in the fragment structure. One of the metrics used in our evaluation is the average nesting depth, $D(m)$, of non-message fragments for each method. The *nesting depth* of a non-message fragment is defined as the number of its enclosing fragments excluding the top fragment. Message fragments were not considered in this metric because they do not contribute to reduced diagram readability due to nesting. For this analysis, the methods in *Nested* were classified into three categories: (a) methods with $D(m) \leq 1$, (b) methods such that $1 < D(m) \leq 2$, and (c) methods with $D(m) > 2$. The larger values of $D(m)$ represent the more complicated fragment structures corresponding to greater levels of nesting. Note that prior to the readability transformations all methods in category (a) must have nested fragments and therefore $0 < D(m) \leq 1$. This is true because methods with $D(m) = 0$ do not belong to *Nested* and therefore are not transformed by the transformations. However, as a result of the transformations, it is possible that some of a transformed method from *Nested* may have $D(m) = 0$ if all the nesting in the method is eliminated. Column (2) in Table 7.2 shows the number of methods in *Nested* for each component. Note that this number corresponds to Column (5c) in Table 7.1 where it is shown as the percentage of the total number of methods for the component. Column (3) represents the distribution of methods with various average nesting depths prior to the readability transformations. Sub-columns (3a)-(3c) correspond to the appropriate categories of $D(m)$. Columns (4a)-(4c) show the partitioning of the categories after the transformations. Additionally, the numbers in parenthesis demonstrate the percent change of each set compared to the appropriate category

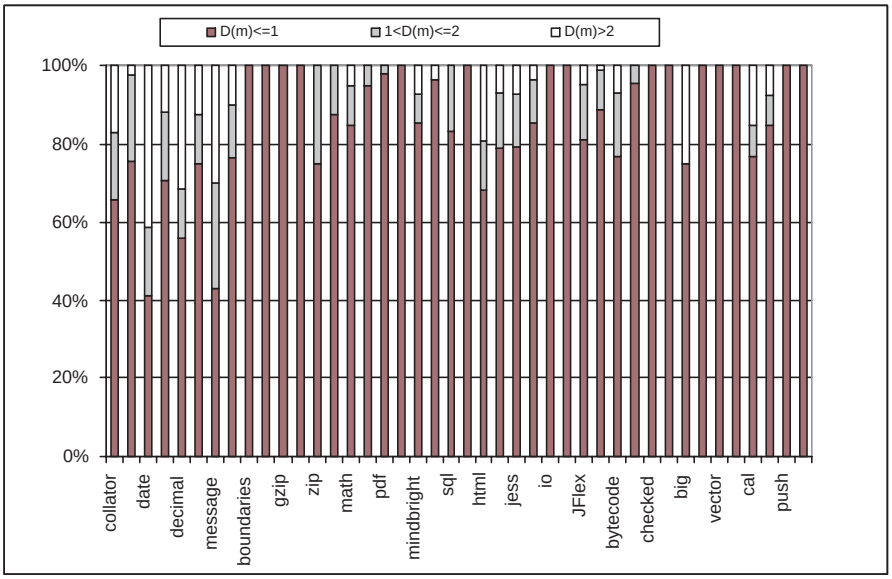

Figure 7.1: Changes in average nesting depth from Table 7.2

in Column (3). For example, for component `collator`, the number of methods in $D(m) > 2$ category decreased by 6 methods, which corresponds to 14.6 percent. As a result of the transformations, the deep nesting of some methods decreased, resulting in lower number of methods with $D(m) > 2$ and an increase in the number of methods in other categories. These results suggest that the transformations can successfully eliminate unnecessary nesting.

The results presented in Table 7.2 are shown in the form of a graph in Figure 7.1. For each component, there are two adjacent bars demonstrating $D(m)$ distribution before and after the readability transformations. Out of the 15 components that could be simplified, 11 demonstrated increase of around 10% or higher in the category where $D(m) \leq 1$. However, components `date` and `message` have shown the most significant improvement of around 30% in this category.

(1) Component	(2) # *Nested*	(3) Before			(4) After		
		(a)	(b)	(c)	(a)	(b)	(c)
`collator`	41	27	7	7	31(+9.8%)	9(+4.9%)	1(-14.6%)
`date`	17	7	3	7	12(+29.4%)	3(0%)	2(-29.4%)
`decimal`	16	9	2	5	12(+18.8%)	2(0%)	2(-18.8%)
`message`	30	13	8	9	23(+33.3%)	4(-13.3%)	3(-0.2%)
`boundaries`	4	4	0	0	4(0%)	0(0%)	0(0%)
`gzip`	6	6	0	0	6(0%)	0(0%)	0(0%)
`zip`	8	6	2	0	7(+12.5%)	1(-12.5%)	0(0%)
`math`	39	33	4	2	37(+10.3%)	2(-5.1%)	0(-5.1%)
`pdf`	46	45	1	0	46(+2.2%)	0(-2.2%)	0(0%)
`mindbright`	55	47	4	4	53(+10.9%)	0(-7.3%)	2(-3.6%)
`sql`	12	10	2	0	12(+16.7%)	0(-16.7%)	0(0%)
`html`	57	39	7	11	45(+10.5%)	8(+1.8%)	4(-12.3%)
`jess`	136	108	18	10	116(+5.9%)	15(-2.2%)	5(-3.7%)
`io`	13	13	0	0	13(0%)	0(0%)	0(0%)
`jflex`	80	65	11	4	71(+7.5%)	8(-3.8%)	1(-3.8%)
`bytecode`	129	99	21	9	123(+18.6%)	6(-11.6%)	0(-7.0%)
`checked`	1	1	0	0	1(0%)	0(0%)	0(0%)
`big`	4	3	0	1	4(+25.0%)	0(0%)	0(-25.0%)
`vector`	4	4	0	0	4(0%)	0(0%)	0(0%)
`cal`	13	10	1	2	11(+7.7%)	1(0%)	1(-7.7%)
`push`	1	1	0	0	1(0%)	0(0%)	0(0%)

Table 7.2: Changes in the average nesting depth D(m)

The data suggests that the transformations effectively reduce the unnecessary nesting. This is also confirmed by the reduction in the total number of non-message nested fragments. The metric used to express the decrease in the number of non-message nested fragments is the average number of such fragments per method, denoted as $A(m)$. The computation of $A(m)$ for each component is performed by calculating the total number of non-message nested fragments in *Nested* divided by the number of methods in *Nested*. Table 7.3 shows $A(m)$ for each component before and after the transformations as well as the resulting improvement. The results

show that the use of the readability transformations results in a significant reduction in fragment nesting. Out of 21 components analyzed, 13 exhibit 25% or greater reduction in the average number of nested non-message fragments per method. Some components (e.g. `math`, `bytecode`, and `big`) demonstrated over 50% reduction.

(1) Component	(2) $A(m)$ before transform	(3) $A(m)$ after transform	(4) Improvement
`collator`	5	3.6	27.7%
`date`	26.1	16.4	37.4%
`decimal`	20.6	13.2	35.8%
`message`	17.1	10.8	36.7%
`boundaries`	2	1.5	25%
`gzip`	2.2	2.2	0%
`zip`	2.5	2.2	10%
`math`	3.4	1.6	53.7%
`pdf`	1.6	1.4	12.5%
`mindbright`	2.1	1.6	25.6%
`sql`	1.8	1.2	31.8%
`html`	5.3	2.9	44%
`jess`	3.6	3	14.8%
`io`	1.4	1.3	5.6%
`jflex`	4	3.1	21.8%
`bytecode`	3.3	1.6	50.9%
`checked`	1	1	0%
`big`	2.8	0.8	72.7%
`vector`	1	0.8	25%
`cal`	9.1	5.2	43.2%
`push`	1	1	0%

Table 7.3: Changes in the average number of nested non-message fragments per method

The data presented here demonstrates that the use of readability transformations could significantly reduce the nesting in the fragment structure for most of the components in this study. These results are promising because they suggest that the

transformations presented in this work could be useful for improving the comprehension and readability of reverse-engineered sequence diagrams.

CHAPTER 8

Related Work and Conclusions

There has been a significant amount of effort in the area of reverse engineering of sequence diagrams or similar representations. Several of the existing approaches employ *dynamic analysis* of run-time program behavior. This technique collects information during the execution of the program and then uses the data to generate the UML diagrams. There are several drawbacks associated with this approach.

The diagrams generated as result of run-time behavior analysis can only represent a particular path of execution that occurred for certain input values. The execution path may represent only partial behavior of the program and the complete set of input values necessary to exercise all aspects of object interactions may not be known. This approach also does not permit reverse-engineering for incomplete systems or libraries that cannot be executed as stand-alone entities. Use of diagrams that are reverse-engineered from run-time information may be misleading to developers trying to understand the system because such diagrams may provide an incomplete picture of object interactions. This could result in incorrect code modifications since it is based on incomplete information.

The Jinsight tool by De Pauw et al. [3] visualizes program behavior based on trace information and provides various views of the run-time data. One of the views shows

information about invocation sequences which is represented similarly to sequence diagrams. Oechsle and Schmitt present the JAVAVIS tool that builds sequence diagrams by utilizing the Java Debug Interface [9]. Richner and Ducasse [11] use run-time analysis of Smalltalk programs to extract information about object collaborations. In this approach, execution sequences similar to each other are identified by pattern matching and the important collaborations between objects are identified with the assistance of the user. The collaboration patterns extracted by the technique are represented as sequence diagrams. All three of the above approaches do not represent the conditions guarding the execution of messages and the repetitive behavior resulting from loops.

Some of the existing work attempts to account for conditional and iterative behavior by using pattern matching techniques. Briand et al. [2] utilize run-time traces of program execution to generate sequence diagrams for C++ programs. They execute instrumented source code and analyze the trace information to identify conditions and loops that affect message sequences. The correct identification of the loops relies on pattern matching techniques. After completion of the analysis, the trace is represented as an instance of a trace meta-model, which is then transformed into an instance of a sequence diagram meta-model.

Dynamic analysis with pattern matching for identification of loops is also used by Systä et al. in the Shimba reverse-engineering environment for Java program [12]. They used a debugger to collect the trace data and provide line numbers for guarding conditions of the conditional behavior. As a result, the representation of the UML-like sequence diagrams generated by this tool contains loops and the line numbers corresponding to the conditional statements.

There is also existing work on reverse engineering of UML diagrams using static analysis techniques. Kollmann and Gogolla perform analysis of Java source code to extract collaboration diagrams, which demonstrate object interactions similarly to sequence diagrams [6]. Their approach involves the creation of a meta model that reflects the information relevant to collaboration diagrams, and a traversal of the model in order to generate the diagrams. This methodology results in the creation of collaboration diagrams in which conditional behavior and loops are not expressed.

Tonella and Potrich also use static analysis techniques to extract both collaboration and sequence diagrams from C++ source code [13]. They utilize points-to analysis techniques to generate a call graph. Their approach uses object naming based on the points-to analysis, and creates a unique diagram object for each `new` expression. However, these diagrams also do not reflect conditional or repetitive behavior.

Our approach significantly differs from the work done by other researchers in this area. The work presented here introduces the first static analysis algorithm for mapping general control flow to the latest UML notation, producing a data structure that could be easily used when generating sequence diagrams. Additionally, we introduce a set of transformations for improving readability and comprehension of sequence diagrams. Our empirical study demonstrates that the analysis is efficient and effective in creating and simplifying the UML fragment structure.

As part of our future work we will extend the analysis to allow representation of inter-method interactions. This approach will extend the analysis presented here to allow representation of sequence diagrams across multiple method boundaries. Introduction of additional transformations could further simplify the resulting diagrams and we would like to explore this aspect of our analysis. Additionally, currently there

are no tools available for visualization of sequence diagrams from textual input using UML 2.0 notation. We would like to extend existing tools to represent the flow of control using UML 2.0 notation, based on the output of our analysis.

BIBLIOGRAPHY

[1] A. Aho, R. Sethi, and J.Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] L. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Working Conference on Reverse Engineering*, pages 57–66, 2003.

[3] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualising the execution of Java programs. In S. Diehl, editor, *Software Visualization*, LNCS 2269, pages 151–162, 2002.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, 1987.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[6] R. Kollmann and M. Gogolla. Capturing dynamic program behavior with UML collaboration daigrams. In *European Conference on Software Maintenance and Reengineering*, pages 58–67, 2001.

[7] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2nd edition, 2001.

[8] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flow graph. In *ACM Trans. Programming Languages and Systems*, pages 1(1):121–141, July 1979.

[9] R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In S. Diehl, editor, *Software Visualization*, LNCS 2269, pages 176–190, 2002.

[10] OMG. *UML 2.0 Infrastructure Specification*. Object Management Group, `www.omg.org`, Sept. 2003.

[11] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *International Conference of Software Maintenance*, pages 34–43, 2002.

[12] T. Systä, K. Koskimies, and H. Muller. Shimba–an environment for reverse-engineering Java software systems. *Software–Practice and Experience*, 31(4):371–394, Apr. 2001.

[13] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *International Conference on Software Maintenance*, pages 159–168, 2003.

[14] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *9th International Conference on Compiler Construction (CC'00)*, pages 18–34, 2000.