University of Alberta

Transactional Pointcuts for Aspect-Oriented Programming

by

Seyed Hossein Sadat Kooch Mohtasham

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

©Seyed Hossein Sadat Kooch Mohtasham
Spring 2011
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Examining Committee

H. James Hoover (Advisor), Computing Science

Kenny Wong, Computing Science

José Nelson Amaral, Computing Science

John C. Bowman, Mathematical and Statistical Sciences

Gregor Kiczales, Computer Science, University of British Columbia

To my wife,
and my family, especially Mom and Dad.

Abstract

In dynamic pointcut-advice join point models of Aspect-Oriented Programming (AOP), join points are typically selected and advised independently of each other. That is, the relationships between join points are not considered in join point selection and advice. But these inter-relationships are key to the designation and advice of arbitrary pieces of code when modularizing concerns such as exception handling and synchronization. Without a mechanism for associating join points, one must instead refactor (if possible) into one method the two or more related join points that are to be advised together. In practice, join points are often not independent. Instead, they form part of a higher-level operation that implements the intent of the developer (*e.g.* managing a resource). This relationship should be made more explicit.

We extend the dynamic pointcut-advice join point model to make possible the designation, reification, and advice of interrelated join points. The Transactional Pointcut (transcut), which is a realization of this extended model, is a special join point designator that selects sets of interrelated join points. Each match of a transcut is a set of join points that are related through control flow, dataflow, or both. This allows transcuts to define new types of join points (pieces of computation) by capturing the key points of a computation and to provide *effective* access for their manipulation (*i.e.* advice). Essentially, transcuts almost eliminate the need for refactoring to expose join points, which is shown by others to have a significant negative effect on software quality.

The transcut construct was implemented as an extension to the AspectJ language and integrated into the *AspectBench* compiler. We used transcuts to modularize the concern of exception handling in two real-world software systems. The results show that transcuts are effective in designating target join points without unnecessary refactorings, even when the target code is written obliviously to the potential aspectization.

Acknowledgements

I would like to thank my PhD advisor, H James Hoover, who supported me throughout all the stages of this thesis. His advice, encouragement, and sense of humour enabled me to keep the spirit up and stay focused even at difficult times.

Also, I would like to thank my supervisory committee, especially Ken Wong and José Nelson Amaral, for their invaluable advice and comments during the development of this project. I am also grateful to Gregor Kiczales for his guidance in my early explorations of Aspect-Oriented Programming. Additionally, I would like to show my gratitude to Natural Sciences and Engineering Research Council of Canada for supporting this work.

Lastly, I am indebted to all the people who encouraged me and helped me in any way during the completion of this work, especially Daqing Hou and Xin Li for sharing their thoughts and providing feedback on my work.

–Hossein Sadat-Mohtasham

Table of Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Thesis . . . 3
- 1.3 Contributions . . . 3
- 1.4 Organization . . . 4

2 Background: Aspect-Oriented Programming **6**
- 2.1 Modularizing Cross-Cutting Concerns . . . 6
- 2.2 Join Point Model . . . 10
- 2.3 Pointcut-Advice Dynamic Join Point Model . . . 10
  - 2.3.1 Join Points . . . 11
  - 2.3.2 Pointcuts . . . 12
  - 2.3.3 Advice . . . 14
  - 2.3.4 Aspect Association . . . 15
- 2.4 Inter-Type Declarations (Static Cross-Cutting) . . . 15
- 2.5 An Example . . . 16

3 Transactional Pointcuts **20**
- 3.1 Motivating Example . . . 20
- 3.2 Transactional Pointcuts . . . 23
  - 3.2.1 Nested Transcuts . . . 26
  - 3.2.2 Looped and Conditional Pointcuts . . . 26
  - 3.2.3 Dependent Pointcut . . . 27
  - 3.2.4 Dataflow Pointcuts and Context Exposure . . . 29
  - 3.2.5 Overlapping Transcuts . . . 30
- 3.3 Program Dependence Graph: the Join Point Representation . . . 30
  - 3.3.1 Region Analysis . . . 33
  - 3.3.2 Join Point Shadows . . . 35
  - 3.3.3 PDG-based Matching Algorithm . . . 36
- 3.4 Applications . . . 40
  - 3.4.1 Modularizing Exception Handling . . . 42
  - 3.4.2 Transaction Management . . . 45
  - 3.4.3 Synchronization . . . 48
  - 3.4.4 Parallelization . . . 49
  - 3.4.5 Data and Control Context Designation . . . 50
  - 3.4.6 Static Verification of API Usage . . . 51
- 3.5 Implementation . . . 53
- 3.6 Limitations . . . 54
  - 3.6.1 Fragility . . . 54
  - 3.6.2 Interaprocedural Limits, Escaping Objects, and Buried Effects . . . 55
  - 3.6.3 Continuation vs. History Semantics . . . 56

4 The Semantics of Transcuts **58**
4.1 Continuation-based Semantics for Dynamic Join Points 58
4.2 Extending the Continuation-based Semantics to Support Transcuts 62
4.2.1 Defunctionalized CPS Interpreter: a Running Example 63
4.2.2 AOP-Enabled CPS Interpreter 66
4.2.3 Enabling Transcuts in CPS Interpreter 68

5 Evaluation **71**
5.0.4 Evaluation Strategy 73
5.0.5 Case Study 1: Tactical Separation Assisted Flight Environment . . . 75
5.0.6 Case Study 2: Eclipse Instant Messenger Plugin 80
5.0.7 Weaknesses of Transcuts 82
5.0.8 Summary of Results 86

6 Related Work **87**

7 Conclusions **91**
7.1 Contributions 91
7.2 Future Work 92

Bibliography **94**

A Construction of the Program Dependence Graph in the Presence of non-Normative Control Flow **98**
A.1 Background 99
A.1.1 Control Flow Graph 99
A.1.2 Dominator and Post-Dominator Trees 101
A.2 Program Dependence Graph 102
A.2.1 Region Analysis 103
A.2.2 Constructing the Region Hierarchy 104
A.2.3 Implementation and Usage 105
A.3 Non-Normative Control Flow 106
A.3.1 Abrupt Loop Exit and Continuation 107
A.3.2 Exceptional Flow 107
A.3.3 Multiple Exits 114
A.4 Conclusion 114

List of Tables

2.1 Pointcuts corresponding to various kinds of join points (adapted from [34].) 12

5.1 Four different treatments of exception handling cases in the experiment. . . 74
5.2 Results of the exception handling modularization in TSAFE system. 75
5.3 Results of the exception handling modularization in EIMP system. 80

List of Figures

2.1 A class from the database layer (left) and the corresponding security aspect (right) . . . 7
2.2 Code tangling in a module . . . 9
2.3 Code scattering of security concern . . . 9
2.4 General form of a named pointcut. . . . 12
2.5 ObserverProtocol aspect (from [24]) . . . 17
2.6 Observer instance: ColorObserver (from [24]) . . . 17
2.7 Observer instance: CoordinateObserver (from [24]) . . . 18
2.8 Observer instance: ScreenObserver (from [24]) . . . 18

3.1 A simple program that uses the standard Logger class to log messages. . . . 21
3.2 An aspect that guards the log calls and their argument evaluation code . . . 22
3.3 An example of three different join points: *call* and *execution* join points as well as a region join point that could potentially be a match for f(), g() sequence. . . . 23
3.4 A simple transcut composed of 3 pointcuts. . . . 24
3.5 A transcut that relates the join points based on their target object. . . . 25
3.6 An example of transcut nesting . . . 26
3.7 Using *looped()* pointcut in a transcut . . . 26
3.8 A match for *readloop()* (left) and a non-match (right). . . . 27
3.9 Using *conditional()* pointcut in a transcut . . . 27
3.10 A match for *readConditionally()* (left) and a non-match (right). . . . 27
3.11 Composition of *conditional()* and *looped()* pointcuts. . . . 28
3.12 A match for *loopedConditionalRead*. . . . 28
3.13 An advice that targets a transcut and simply executes the original join point. 28
3.14 Three different matches for the same transcut. . . . 29
3.15 A transcut using *dependent()* pointcut. . . . 29
3.16 Two overlapping transcuts . . . 30
3.17 A program and its CFG . . . 31
3.18 Dominator (left) and post-dominator (right) trees . . . 32
3.19 PDG of the program in Figure 3.17 . . . 33
3.20 A region and its potential shadows . . . 36
3.21 Transcut Matching Algorithm . . . 37
3.22 Transcut Matching Algorithm (matchPointcutInRegion) . . . 39
3.23 Transcut Matching Algorithm (matchConditionalInRegion) . . . 41
3.24 Handling an exception is an aspect . . . 43
3.25 A file operation within method boundaries that could throw exceptions . . . 43
3.26 An aspect that implements recommended practice for handling resources . . 44
3.27 A typical database client code . . . 45
3.28 Transaction handling transcut . . . 46
3.29 Transcut to handle resources . . . 47
3.30 Critical sections in writer (left) and reader (right) threads. . . . 49
3.31 Transcut to synchronizes reader's critical section. . . . 49
3.32 Transcut for open/close usage . . . 51
3.33 *declare error* advice in AspectJ . . . 52
3.34 Static API usage verification using transcuts. . . . 52

5.1 Original code (left), after removing exception handling code (middle), possible handler aspect (right). 73
5.2 The original code of a method from TSAFE. 77
5.3 After removing exception handling code. 78
5.4 Exception handling aspect for method in Figure 5.3. 79
5.5 A sample case in EIMP system before separating exception handling. . . . 81
5.6 Sample code after removing exception handling concern code. 81
5.7 Exception handling aspect corresponding to code in Figure 5.6. 82
5.8 Unsupported boundary join point 83
5.9 Exception handling advice would need to "return" from the method containing the target join point. 83
5.10 The complex case of "return" from region 84
5.11 The complex case of "return" from region (left), its corresponding CFG (middle), and PDG (right) 85
5.12 A simple case (left), its corresponding CFG (middle), and PDG (right) . . . 85

A.1 A program and its CFG 100
A.2 Dominator (left) and post-dominator (right) trees 101
A.3 PDG of the program in Figure A.1 103
A.4 An example of construction and usage of a PDG 106
A.5 UML class diagram for part of the PDG package 107
A.6 A program and its CFG 108
A.7 Dominator (left) and Post-dominator (right) trees 108
A.8 The PDG of program in Figure A.6 109
A.9 Control flow out of a node 109
A.10 Control flow inside a try-catch-finally 110
A.11 Brief CFG 110
A.12 Adding a dummy node to the brief CFG 111
A.13 The enhanced CFG corresponding to the brief CFG 113
A.14 The enhanced CFG of typical exception-handling code 113
A.15 The PDG for program in Figure A.14 113

Chapter 1

Introduction

Almost all software development methodologies and programming paradigms have tried to address the problem of software complexity by decomposing a software system into manageable units that can be handled in isolation, which is generally referred to as the *separation of concerns*[1] and it is achieved through modularization. All important software quality attributes are, directly or indirectly, influenced by the way the software is modularized; that is, the criteria that are used to decompose a system and the modularization constructs provided by the programming language used for implementation. In the early years of programming, programs were small and machine language was the primary programming tool. Flowcharts were used to model the processing steps in programs and these processing steps were the main factor in decomposing systems. As systems became more complex, the old decomposition criteria and techniques were impractical. Parnas is his classic paper shows that "information hiding" is the right criterion to be used in decomposing systems into modules [44]. Information hiding means that each module should hide a design decision (that is most likely to change) and expose as little of that decision as possible to other modules through an interface. Therefore, when one design decision changes, other parts of the system will most probably not be affected.

The advances in software development and programming languages have always been mutually dependent. New programming abstractions and constructs were needed to realize ideas such as Parnas's information hiding. In fact, one could say that Object-Oriented Programming (OOP) is a well-known realization of many ideas, one of which is information hiding. OOP brought in abstraction and composition mechanisms that enabled the development of very large systems with reasonable cost that would otherwise be very hard

[1]Informally, a concern is anything that any stakeholder thinks of at any given time [31]. Examples of typical concerns in a software system include security, computing a mathematical function, printing status of an operation, and so on.

to achieve. OOP was a major evolutionary step of programming languages; yet, not long after its wide acceptance, as demands for larger and more complex software were rising, researchers and engineers noticed new problems, that is cross-cutting concerns, that could not be elegantly dealt with using the object-oriented techniques.

Kiczales *et al.* [32] target these problems and propose yet another abstraction and composition mechanism that is fully compliant to the previous methods (*i.e.* OOP), but supports the modularization of concerns that could not be cleanly modularized using the previous methods. These concerns are referred to as *cross-cutting concerns* and the new programming model is called *Aspect-Oriented Programming (AOP)*.

AOP mechanisms are characterized by their join point models. As defined in [32], a join point model has three components: join points, which are elements of language semantics; “a means of identifying join points”; and “a means of affecting the behaviour at those join points.” In dynamic join point models [56, 41], join points are well-defined points in the execution. AspectJ, which is the state-of-practice aspect-oriented language, supports a dynamic join point model that is referred to as the *Pointcut-Advice* model.[2] *Pointcuts* select a set of join points of interest and *advice* affects the semantics of the selected join points.

1.1 Motivation

In a pointcut-advice model, each join point is selected and advised individually and independently. That is, the relationships between join points are not taken into account in join point selection and advice, except in limited predefined ways, such as the *cflow* pointcut in AspectJ. For example, one cannot designate a call join point whose target object is returned by another join point in the same control flow region.

In fact, AspectJ designers made a decision to make join points as context-insensitive as possible so as to make pointcuts more predictable to the programmer. Consequently, there are two things that are not well supported. First, join points cannot be selected based on their relationship to other join points (context). For instance, one can designate all the calls to method *f()*, but not the subset of those that are preceded and succeeded by a call to method *g()*. Secondly, join point types are predefined (by the target language) and therefore computation patterns that are in fact formed by a set of interrelated join points cannot be designated and advised together. This limitation forces programmers to refactor the target

[2]AspectJ also supports a static join point model which is out of the scope of this work. We work with the AspectJ’s pointcut-advice model as a real-world realization of dynamic join point model; however, the concepts in this work are not specific to AspectJ.

code into methods to make them selectable by pointcuts, which is shown to negatively affect software quality [8].

Other researchers have recognized the above limitations and provided languages or extensions to address them. Region pointcut [2] is an independent work that addresses the same problems as we do even though important differences exist. Trace-based aspect mechanisms (*e.g.* [3]) aim at addressing the first limitation (*i.e.*, join point selection based on their relationship with other join points), to some extent, through making past contexts accessible to the programmer but suffer from inherent limitations (no *around/before* advice and low performance) of trace-based models. The *Ptolemy* language addresses the second limitation by allowing the programmer to explicitly specify arbitrary pieces of code as instances of declared typed events [46]. The main problem is that the event announcement mechanism in Ptolemy is explicit; that is, the events are announced in the base code (see Section 6 for details of related work).

1.2 Thesis

We claim that the dynamic pointcut-advice join point model can be extended to take join point interrelationships into account and to allow the designation, abstraction, and advice of arbitrary but well-formed pieces of computation as join points. This extended model simplifies the separation of extended concerns (such as transactions). It also helps avoid refactorings that are primarily aimed at exposing join points that can be handled by the original dynamic pointcut-advice join point model.

1.3 Contributions

We propose[3], design, and implement Transactional Pointcuts (transcuts) [50] as a realization of the new model in the AspectJ language. A transcut is a special join point designator that selects sets of interrelated join points. Each match of a transcut is a set of join points that are related through control flow, dataflow, or both. The basic observation behind transactional pointcuts is that join points do not occur in isolation, but rather, are parts of a higher-level computation that can in turn be regarded as a join point. Transcuts define new kinds of join points by capturing the key points of target computations and providing *effective* access for their manipulation.

We present a new join point representation based on the Program Dependence Graph

[3]First presented in [49] and later in [50].

(PDG). PDG is a program representation that incorporates both control flow and data flow, as well as region hierarchies that are necessary for implementing transcuts. *looped()*, *conditional()*, and *dependent()* pointcuts have been added to transcuts to allow expressing complex join point dependencies. Also, nested transcuts and their interaction with other elements of transcuts are presented.

The major contributions of this thesis are as follows:

- A new join point model, based on the pointcut-advice model, is defined. This new model makes designation and advice of interrelated join points possible.
- A new construct (transcut) is designed and implemented. Transcut is a (partial) realization of the above model and can seamlessly be integrated with the existing languages, specifically AspectJ.
- A join point representation based on the Program Dependence Graph (PDG) is presented. This representation is the backbone of the transcut matching algorithm.
- Some pseudo pointcuts that can be used in transcuts are designed: (*looped()*, *conditional()*, and *dependent()*.)
- Some applications of transcuts are presented.
- A continuation-based semantics for transcuts is presented.
- The results of modularizing exception handling using transcuts in two real-world software systems are reported.
- An implementation of PDG that can deal with non-normative control flow, such as exceptional flow and loop continuation/exit, contributed to the Soot [53] compiler optimization framework.
- Weaknesses and limitations of transcuts, whether in their conceptual grounds or in the design and implementation, are examined and possible directions are proposed for resolving them.

1.4 Organization

The rest of the thesis is organized as follows: Chapter 2 presents some background material on AOP, and AspectJ in particular. We present the new join point model along with transcuts in Chapter 3. Also, more specifically in that chapter, Section 3.3 presents the necessary program representations and concepts to understand transcuts' semantics and the join point model. A set of applications are presented in Section 3.4 to show how transcuts can be utilized. The implementation details are explained in Section 3.5 and some of the limitations are presented in Section 3.6. The semantics of transcuts is explained based on the continuation-based semantics in Chapter 4. Chapter 5 discusses the concern of exception handling and the results of applying transcuts in the modularization of exception handling in two real-world systems. The important related work is discussed in Chapter 6, and, finally, we conclude the thesis in Chapter 7.

Chapter 2

Background: Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [32] introduces the necessary concepts and mechanisms that allow modularizing cross-cutting concerns. Procedural programming introduced functional abstraction and composition and OOP introduced object abstraction on top of it. OOP enabled programmers to handle the complexity of large-scale software systems and has been the dominant programming model for the last two decades. However, OOP cannot help in the modularization of those concerns (behaviours) in a system that span many modules (the implementation of other concerns.) As it will be explained in this chapter, using OOP to implement such concerns results in *code scattering* and *code tangling*, which imply low-quality code [34]. AOP allows abstraction and composition of cross-cutting concerns.

2.1 Modularizing Cross-Cutting Concerns

Most of software development methodologies rely on decomposition techniques to break a system into smaller and more manageable functional units. Traditional programming language models provide the facilities needed to abstract these functional units and compose them in different ways in different contexts. These units were called procedures in procedural programming and objects in object-oriented programming (OOP) both of which are considered to be some forms of *generalized procedure* (GP) [32]. The programming language models whose main abstraction and composition constructs are generalized procedures are referred to as GP languages. In fact, in all GP languages units of decomposition are usually units of functionality. Therefore, the features that are not functional (*i.e.*, do not fit well in the decomposition hierarchy) cannot be put in a single module; the result is an implementation code that is tangled up and hard to maintain. It should be mentioned that there are also functional units that cannot elegantly be modularized. Some of the common features that might not be easy to modularize are security, tracing, profiling, performance, and so on.

Informally, a concern is anything that any stakeholder thinks of at any given time [31]. Two different concerns, when implemented, crosscut each other if they must compose differently and yet be coordinated [32]. GP languages only provide one composition mechanism, hence, only those concerns that can be abstracted and composed using generalized procedures can be cleanly modularized. The implementation of the other concerns is scattered over other modules because they need to coordinate with the other concerns at various points in the code.

Several methodologies have been developed to address the problems related to cross-cutting concerns (*e.g.* Adaptive Programming [42], [35], Composition Filters [7], Subject-Oriented Programming [26], Intentional Software [51], Multi-0 Separation of Concerns [43], Metaobject Protocols [33], and Generative Programming [13].) Aspect-Oriented Programming has become the most widespread methodology to deal with cross-cutting concerns.[1]

In order to illustrate some of AOP concepts, we present a simple example of a cross-cutting concern. We consider an online e-tailer application through which users can search for and purchase products, as well as manage their accounts. In such applications, there is usually a data-tier that consists of classes that access the database for retrieving or modifying information. The classes and their methods could be structured in various ways, however, for the purposes of this example, we assume that any method in any class that has a name starting with "db" is a method that accesses the database (Figure 2.1, on the left.)

```
class UserAccount {

  ...

  public boolean dbChangeAddress(Address a) {
      /*get db connection
      make a query
      execute the query
      return true if successful,
      false otherwise.*/
  }
  ...
  public boolean dbAdd2Wishlst(Product p){
    ...
  }
}
```

```
aspect  Security {

pointcut  dbaccess():
  execution(boolean
    UserAccount.db*(..));

  boolean around() : dbaccess() {
    /*check user login status
    if signed in, continue with
    method execution. else do
    not execute and simply return
    false.*/
  }
}
```

Figure 2.1: A class from the database layer (left) and the corresponding security aspect (right)

[1]Many of these methodologies can be considered to be "aspect-oriented", if they have a join point model.

Each one of these methods implements part of a *functionality*. Assume that a new feature is requested that requires all database methods to check login status of the user before accessing the database. One way of accommodating this change is to go through all the code and find and change all the methods that access the database. Not only is this approach cumbersome and error-prone but it would also lead to less maintainable code (*e.g.*, imagine a system with 40 classes in a data-tier, each containing on average 10 methods, that access the database.) To make things worse, this new security feature might change (evolve) as other features evolve in the system. What makes adding this new feature difficult is its cross-cutting nature: the implementation of this feature, using traditional modularization techniques, requires change in various places in the code. In other words, the feature's code cannot be packaged in one place and is scattered throughout the code.

To resolve this situation a mechanism is needed that not only does allow expressing the new cross-cutting features without breaking the existing decomposition, but also helps to do it in a modularized fashion. Figure 2.1 (right) shows a module that can implement the added security feature in the above example.[2] The *Security* aspect implements the required security feature in a modularized way. It first gives a name (*i.e. dbaccess*) to the points in the execution that correspond to the *execution* of the methods in the *UserAccount* class, whose names begin with "db" and return a *boolean* (lines 3-4.) These points are the places where security should be checked (join points) and *dbaccess()* is a pointcut that identifies them.

Having identified the points in the execution (*i.e.*, join points) where the security should be checked, the only thing that remains is to bind the security behaviour to the join points so that it is executed when the join points are activated at runtime (*i.e.*, when the control enters the corresponding methods.) This behaviour can be implemented using an *around advice*, which is a construct that can change the behaviour of a set of selected join points: it runs instead of its target join points and can execute the join points once, or 1 (if at all.) The security aspect, in the example, takes the control from the target methods and does the necessary operations (*e.g.* check if the current user is authenticated) and then, it can then either continue with the original execution or dismiss the database access (*e.g.* in case the user has not signed in yet.)

Figure 2.2[3] shows how a module may look like in a system in which cross-cutting concerns are not modularized. The module is implementing a functional feature in the

[2]This is AspectJ syntax, which is a popular aspect-oriented language based on Java.
[3]Adapted from [34].

business logic, however, there are pieces of code that belong to other concerns in the system, such as security and tracing. In other words, the module implements pieces of multiple concerns. This presence of pieces of implementation of multiple concerns in a module is referred to as *code tangling.*

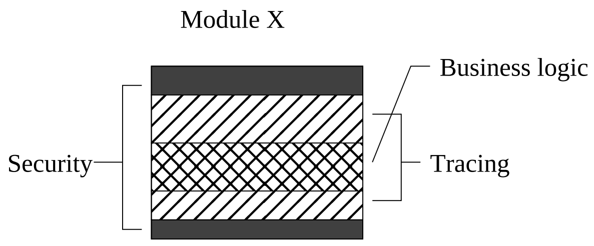


Figure 2.2: Code tangling in a module

Code scattering is illustrated in Figure 2.3. The implementation of a cross-cutting concern like security can be present in many other modules, even in other cross-cutting modules. Code scattering can occur in two different ways. It occurs when a piece of implementation repeatedly appears in many modules (like in the example in previous section.) It can also be the result of complementary pieces of implementation (of a single concern) appearing in various modules. For instance, security concern includes user authentication and authorization which are checked and enforced in different modules in a system.

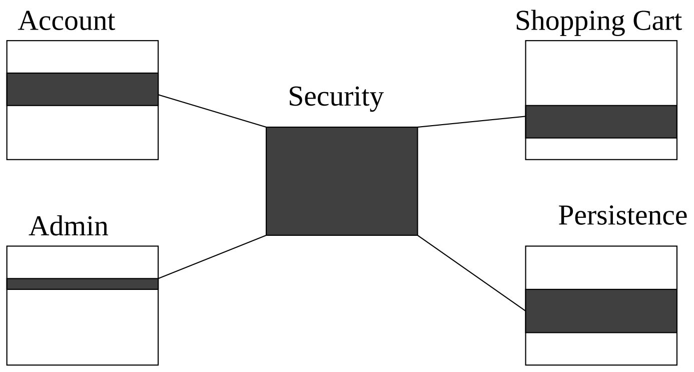


Figure 2.3: Code scattering of security concern

Code tangling and code scattering negatively affect software design and development by causing poor traceability, lower productivity, lower code reuse, poor quality, and harder evolution [34].

2.2 Join Point Model

AOP mechanisms are characterized by their join point model. A *join point model* defines three elements: *join points*, *pointcuts*, and *advice*. Join points are meaningful elements of the programming language semantics. Pointcuts are predicates that select join points based on their properties and context. Advice is a way of changing the semantics at the selected join points. In the above example, join points are method executions. *dbaccess()* is a pointcut that selects those join points (method executions) whose signature pattern matches *boolean *.db*(..)* (*i.e.* all methods that start with “db”, take any number parameters with any type, belong to any class, and return a boolean value.)

There are various join point models in AOP. AspectJ, for instance, supports two join point models: *introductions* and *pointcut-advice* join point model. The *introductions* join point model handles static cross-cutting: join points are class declarations to which new member variables and methods can be added; also, the inheritance hierarchy can be changed through adding new parent classes and interfaces. This static join point model complements the *pointcut-advice* join point model in which join points are points in the execution of a program.

Contrary to the common mentality, cross-cutting is a three-party relationship [40]: a base program (in language A), an aspect program (in language B), and a common representation (X). The base program and the aspect program cross-cut each other with regard to X, that is the common representation, which can be (but not necessarily) a lower-level code: neither contains the other, and they are not disjoint in the common representation.[4] Join points are element in X; and A and B have constructs to refer to the join points in some way to express some behaviour about them. A weaver then takes X, A, B, the join point description, the base 0, and the aspect program, and composes the base program and aspect program at the common join points to get a single representation in X. X does not have to be different than A and B.

In this work our focus is the dynamic *pointcut-advice* join point model, as realized in AspectJ, however, the concepts introduced in this thesis are not limited to AspectJ.

2.3 Pointcut-Advice Dynamic Join Point Model

The dynamic pointcut-advice join point model is a join point model in which join points are run-time entities and not static elements in a program. Nonetheless, a specific implementa-

[4]It should be noted that there is no generally-agreed definition of cross-cutting concerns.

tion of a dynamic join point model might transform the static representations of these join points to affect their semantics (*i.e.* weave advice.) Therefore, in the semantical sense, it is not correct to use verbs such as “injecting”, “inserting”, *etc.* to refer to weaving; however, these words might be used in the context of a specific implementation of weaving. In this section, we introduce the AspectJ’s pointcut-advice join point model which is used in the rest of this thesis.

2.3.1 Join Points

A dynamic join point in AspectJ is any identifiable point in the program execution, such as a method call, a method execution. Not all join points, however, are exposed by AspectJ to prevent unreliable and fragile aspects. For instance, the point where a local variable is set is not a join point; nor is a loop inside a method. There are several kinds of exposed join points in AspectJ:

- Method call: this is a point in execution where a method is to be dispatched.
- Method execution: this is when the body of a method is to be executed.
- Field access: when a field of a class is read or written.
- Constructor call and execution: these join points are similar to method call and execution but represent the creation of an object.
- Object initialization and pre-initialization: object initialization join point starts from the return of a parent class’s constructor until the end of the first called constructor. Object pre-initialization, which is rarely used, is the code from the beginning of the first called constructor to the beginning of its parent constructor.
- Exception handler execution: represents the execution of a handler block of an exception type in an exception-handling block.

Method call and execution are the most important kinds of join points because they represent the points in execution where some meaningful behaviour (which a designer has abstracted) occurs. These join points are referred to as “points” but some of them are represented by not just a point but a block of code (*e.g.* method execution, handler.)

Join points can have some related context available to be exposed to aspect programs. For example, a call join point can have a corresponding target object, executing object, and list of arguments.

2.3.2 Pointcuts

Pointcuts are constructs that select and abstract a set of join points and expose their contexts. There is a pointcut designator corresponding to each kind of join point. For instance, *call(* *.* (..))* selects all the call join points and *execution(* *.* (..))* selects all the execution join points in a program. Pointcuts can be named to make it easier to reuse them at multiple places. Figure 2.4 shows the general form of a named pointcut and Table 2.1 shows the pointcuts used to identify various join point categories.

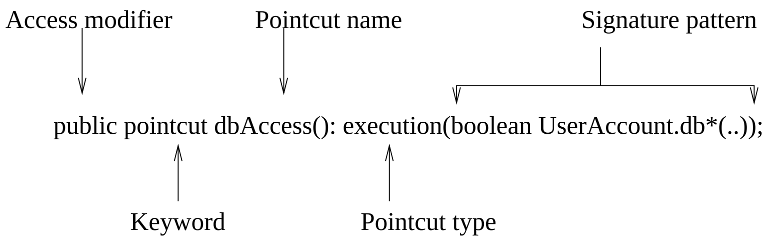

Access modifier
Pointcut name
Signature pattern
public pointcut dbAccess(): execution(boolean UserAccount.db*(..));
Keyword
Pointcut type

Figure 2.4: General form of a named pointcut.

Join Point Category	**Pointcut Syntax**
Method call	call(MethodSignature)
Method execution	execution(MethodSignature)
Constructor call	call(ConstructorSignature)
Constructor 0	execution(ConstructorSignature)
Field read access	get(FieldSignature)
Field write access	set(FieldSignature)
Class initialization	staticinitialization(TypeSignature)
Object initialization	initialization(ConstructorSignature)
Object pre-initialization	preinitialization(ConstructorSignature)
Exception handler execution	handler(TypeSignature)
Advice execution	adviceexecution()

Table 2.1: Pointcuts corresponding to various kinds of join points (adapted from [34].)

TypeSignature is a pattern that identifies a set of types. For instance, *java.util.** identifies all types directly defined under *java.util* package, *UserAccount* identifies the type with the name *UserAccount*, and *java..*Statement+* specifies all the types in java package and its direct or indirect sub-packages (the ".." operator) that have a name ending in "Statement" (the "*" operator) and their subtypes (the "+" operator.) Binary operators can be used, as set union and intersection, on type signatures to combine type signatures (*e.g.* "*java.util.* || java.io.**".)

MethodSignature is a pattern that identifies a set of methods. For example, the following pattern identifies all the methods in class Model whose names begin with "get" and take a single argument with any type:

```
* Model.get*(*)
```

To identify only those methods that take an argument of type $String$ and return a $boolean$ one can write:

```
boolean Model.*(String)
```

Similarly, to identify all the methods in Model and its subclasses that take any number of arguments with any type one can write:

```
* Model+.*(..)
```

Other signature properties, such as access modifiers and list of thrown exceptions, can also be specified in method patterns.

ConstructorSignature is similar to *MethodSignature* with the exception that instead of a method name, new is used and no return type is specified. For instance, the following pattern specifies all the constructors of class $Model$:

```
Model.new(..)
```

FieldSignature is used to identify a set of member fields is just like a field declaration with the possibility to use wild card characters in place of the filed type, the declaring type, and the field's name. For example, to select all the public fields in class $Model$ with any type and a name that starts with "parent", one can write the following:

```
public * Model.parent*
```

Lexical Pointcuts

within() and *withincode()* pointcuts can be used to restrict join point selection to those that lexically occur in a specific class (set of classes) or a specific method (set of methods), respectively. For example, *within(Model)* selects all the join points that lexically occur in the $Model$ class, which when combined with other pointcuts, can be used to narrow join point selection to a desired set. For example, to select all the method call join points within the $Model$ class, one can write:

```
call(* *.*(..)) && within(Model)
```

The other lexical pointcut is *withincode()* which is similar to *within()* but takes a method/0 signature instead. For instance, to select all the $Model$'s field-write join points in the $Model$'s *load()* method the following can be used:

```
set(* Model.*) && withincode(* Model.load())
```

Control-flow Pointcuts

Control-flow-based pointcuts select join points based on the control flow of the join points selected by another pointcut. For example, if one desires to identify all the join points that do not occur as a result of the execution of the advice in aspect $ErrorHandler$, a control-flow pointcut can be used:

```
cflowbelow(execution(* ErrorHandler.*(..)))
```

The *cflowbelow(pc)* pointcut takes another pointcut (pc) as argument and selects the join points that occur in the control flow of the join points selected by pc excluding the join points selected by pc.

cflow(pc) is similar to *cflowbelow(pc)* with the exception that it selects the join points selected by pc as well.

Context Exposure Pointcuts

Join points might have some relevant context that can be used by advice. For example, a method call join point may have a target object, an executing object, and a list of arguments. Context exposure pointcuts are used to bind the context available at identified join points and expose them to advice. For example, the following pointcut exposes all the above values at all call join points:

```
pointcut AllCalls(Object thisobj, Object targetobj, Object argobj):
        call(* *.*(..)) && this(thisobj) && target(targetobj) && args(argobj);
```

2.3.3 Advice

There are three kinds of dynamic advice:

- *before()*: which executes before the execution of the target join points.
- *after()*: which executes after the execution of the target join points.

- *around()*: executes in place of the target join points and can call the original join point (using *proceed()* expression) with a possibly modified context, call them multiple times, or not call them at all.

2.3.4 Aspect Association

By default, only one instance of an aspect exist per virtual machine which is shared by all the advised join points. There are situations, however, that one desires to associate an aspect with each object or method execution in such a way that a new instance of the aspect is created and used per object or per execution of a method. For instance, the *TrackStatusAspect* aspect below is associated with each object executing a method with signature pattern ** *.execute*(..)*. This association is necessary because $status$ member variable needs to be associated with one single object so that different objects will not overwrite other object's status.[5]

```
aspect TrackStatusAspect perthis(excuteCommand()) {

  private int status;

  pointcut executeCommand(): execution(* *.execute*(..));

  /*
    set status based on the results of command execution ...
  */
}
```

In addition to *perthis(pc)*, *pertarget(pc)*, *percflow(pc)*, and *percflowbelow(pc)* association pointcuts can be used, all of which take a pointcut as argument. *pertarget(pc)* associates an aspect instance with each target object of the join points selected by pc. *percflow(pc)* and *percflowbelow(pc)* associate an aspect instance with each control flow (execution) matching the pointcut. The following shows an example of the *percflow(pc)* association:

aspect TransactionManager percflow(transacted())

pointcut transacted(): execution(* DataObject+.update(..));

/* Manage transactions */

2.4 Inter-Type Declarations (Static Cross-Cutting)

In addition to the dynamic pointcut-advice join point model, AspectJ supports a static join point model that makes it possible to introduce new member fields and methods associated with a class in an aspect. It also allows changing class hierarchies by declaring new

[5]Once could use a hash table to implement aspect association.

parent-child relationships among classes and interfaces. For example, the following example shows how to add a new field $name$ and its access method to class $Model$:

```
aspect Naming {
  private String Model.name;

  public String getName() {
    return name;
  }
}
```

When the access modifier in a member introduction is $private$, the field will only be visible to the defining aspect.

To change the class hierarchy the *declare parent* advice can be used. For instance, to make the $Model$ class implement the $Viewable$ interface, the following advice can be used:

```
declare parent: Model implements Viewable;
```

There are two other useful static advice that use *declare* pattern, *i.e. declare error* and *declare warning*. They can be used to generate compile-time error and warning messages when the presence of some identified join points is detected. For example, to issue a compile-time error when some kind of factory pattern should be enforced, the following can be used:

```
aspect EnforceFactory {
  pointcut newInClass() : within(SomeClass) && call(*.new(..));
  declare error : newInClass(): "Must only use factory methods to create objects!";
}
```

2.5 An Example

To put everything together, an example is presented, from [24], to show how aspects can be used to modularize and reuse the observer design pattern [23]. The observer pattern is used when a list of observer entities are notified to be updated when a subject entity's state changes. For details about this example and its design rationale see [24].

Figure 2.5 shows the *ObserverProtocol* aspect. The design is based on two roles: $Observer$ and $Subject$ realized by two interfaces that only help in adding strong typing to the related methods. The aspect keeps a mapping between each subject and its observers. Also, the aspect has methods to add and remove an observer to and from the list of observers of a subject.

The more interesting part of the aspect is the abstract pointcut *subjectChange()* that is to be an abstraction of all the join points that are considered to be (causing) a "change"

```
public abstract aspect ObserverProtocol {
  protected interface Subject { }
  protected interface Observer { }

  private WeakHashMap perSubjectObservers;

  protected List getObservers(Subject s) {
  if (perSubjectObservers == null)
    perSubjectObservers = new WeakHashMap();

  List observers = (List)perSubjectObservers.get(s);
  if ( observers == null ) {
    observers = new LinkedList();
    perSubjectObservers.put(s, observers);
  }
  return observers;
  }

  public void addObserver(Subject s,Observer o) {
    getObservers(s).add(o);
  }
  public void removeObserver(Subject s,Observer o) {
    getObservers(s).remove(o);
  }

  abstract protected pointcut: subjectChange(Subject s);

  abstract protected void updateObserver(Subject s, Observer o);

  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() )
      updateObserver(s, ((Observer)iter.next()));

  }
}
```

Figure 2.5: ObserverProtocol aspect (from [24])

```
public aspect ColorObserver extends ObserverProtocol {
  declare parents: Point implements Subject;
  declare parents: Line implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
              (call(void Point.setColor(Color)) || call(void Line.setColor(Color)))
              && target(s);
  protected void updateObserver(Subject s, Observer o) {
    ((Screen)o).display("Color change.");
  }
}
```

Figure 2.6: Observer instance: ColorObserver (from [24])

```
public aspect CoordinateObserver extends ObserverProtocol {
  declare parents: Point implements Subject;
  declare parents: Line implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
                     (call(void Point.setX(int))
                     || call(void Point.setY(int))
                     || call(void Line.setP1(Point))
                     || call(void Line.setP2(Point)) ) && target(s);

  protected void updateObserver(Subject s, Observer o) {
      ((Screen)o).display("Coordinate change.");
  }
}
```

Figure 2.7: Observer instance: CoordinateObserver (from [24])

```
public aspect ScreenObserver extends ObserverProtocol {
  declare parents: Screen implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s): call(void Screen.display(String))
                                    && target(s);

  protected void updateObserver(Subject s, Observer o) {
    ((Screen)o).display("Screen updated.");
  }
}
```

Figure 2.8: Observer instance: ScreenObserver (from [24])

in the subject; it is abstract and protected because it is to be defined by concrete observer patterns. However, nothing prevents one from advising the abstract pointcut. An *after()* advice is used to implement the update behaviour for the list of observers of the changed subject (lines 30-35.)

Figures 2.6, 2.7, and 2.8 show three different concrete instances of the observer pattern in the context of a figure package, with classes $Point$, $Line$ (as subjects), and $Screen$ (as both subject and observer.)

Chapter 3

Transactional Pointcuts

3.1 Motivating Example

In this section, we present a natural situation that the existing constructs either cannot address or where they are a hassle in practice. The example is a simple but important application of transcuts to a problem that was described on AspectJ's user mailing list.[1] These examples compile and run with our extension to AspectJ; we present these examples in full detail so as to leave little to the reader's imagination.

In dynamic join point models, a method-call join point does not include the evaluation of the method parameters: all the parameters in a call are evaluated first, then the corresponding call join point is activated. Consider the following scenario, adapted from the AspectJ users mailing list:

An application contains 5000+ calls to a logging method *void Logger.log(Level, String)* whose first parameter is the level of importance of the log and the second parameter is a log string formed by several string concatenations. The standard Logger class provides methods for getting and setting the current level of logging to one of several severity levels (*e.g.* SEVERE, ERROR, INFO, *etc.*). A message and its severity are passed to the logger, which logs the message only if its severity level is higher than the logger's current severity level. Figure 3.1 shows a simple Java program that uses a standard Logger to log a severe (lines 23-26) as well as an informational message (lines 18-21).

In the application scenario, most of the log messages are only informational messages that are formed by assembling various strings. String concatenations are expensive, therefore a large number of logging calls degrades performance. One might think that performance could be boosted by setting the severity level to Error to disable the logging of informational messages. The problem is that the severity level of a message is checked inside

[1]http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg06225.html and the follow-ups.

```
class LoggingExample{
  Logger m_logger = null;

  public static void main(String[] args) {
    LoggingExample logExample = new LoggingExample();
    logExample.doSomeOperation();
  }

  public LoggingExample() {
    m_logger = Logger.getLogger(””);
    m_logger.setLevel(Level.SEVERE);
  }

  public void doSomeOperation() {
    /* Assume that  ‘‘getStatus()” is a
    method that returns status string.*/

    m_logger.log(Level.INFO,
      ”this should NOT be logged:” +
      getStatus() +
      (new Date()).toString());

    m_logger.log(Level.SEVERE,
      ”this should be logged: ” +
      getStatus() +
      (new Date()).toString());
  }
}
```

Figure 3.1: A simple program that uses the standard Logger class to log messages.

the *log()* method and, as a consequence, the expensive corresponding string concatenations (caused by eager parameter evaluation) are executed. Performance still degrades regardless of whether the message is logged or not.

The first solution that comes to mind is to guard each *log()* call with a conditional that checks the logging level and calls the *log()* only if the message is worth logging. However, that requires placing guards at all 5000+ places in the code, therefore the obvious tactic is to use an AspectJ *around* to implement the guard at calls to the logger method: *void around(): call(* Logger.log(Level, String))*. But in AspectJ, a call to *log()* excludes the respective parameter evaluation. Therefore, any around advice would be executed after the expensive string concatenations. In other words, one incurs the price of the parameter evaluation regardless of advice. AspectJ exposes only the most important (principled) points in the control flow of a program as join points. In cases such as this, refactoring is required to restructure the code into exposable join points.

How transactional pointcuts (transcuts) can help in this situation is shown in Figure 3.2. The *logCut* transcut is essentially a pointcut that matches only when all of its constituent pointcuts match sequentially against a well-defined region of code (see Section 3.2 for precise definitions). The transcut designates the region of code that begins with creating

```
 1
 2 aspect LoggingAdapterAspect{
 3
 4   transcut logCut(Logger logger,
 5         Level level,
 6         StringBuffer sb,
 7         String s) {
 8
 9     pointcut createbuf: call(StringBuffer.new(..))
10             && return(sb);
11     pointcut append: call(* StringBuffer.append(..))
12             && target(sb);
13     pointcut tostring: call(String StringBuffer.toString())
14             && target(sb)
15             && return(s);
16     pointcut logcall: call(* Logger.log(Level, String))
17             && target(logger)
18             && args(level, s);
19   }
20
21   void around(Logger logger,
22         Level level,
23         StringBuffer sb,
24         String s):    logCut(l, level, sb, s) {
25
26     if(logger.getLevel().intValue() <= level.intValue())
27       proceed(logger, level, sb, s);
28   }
29 }
```

Figure 3.2: An aspect that guards the log calls and their argument evaluation code

a *StringBuffer* followed by at least one *StringBuffer.append()* method call, followed by at least one *StringBuffer.toString()* method call, and ends with a *Logger.log()* method call.[2]

The pointcuts that form the transcut are traditional AspectJ pointcuts, but their parameters are shared — the union of the context parameters for each pointcut forms the shared context parameter list of the transcut. Also, the semantics of the pointcuts that rely on run-time type information (*e.g. args(), target(), etc.*) has changed to rely on data flow information instead. For instance, the *StringBuffer* object that was created, in the example, is tracked to the point where its embedded string is retrieved through a call to *toString*. The *logCut* transcut matches the log calls in Figure 3.1, lines 18-21 and 23-26. Lines 21-28 in Figure 3.2 show the advice around *logCut* that checks the level of the logger against that of the message to decide whether or not to evaluate the parameters and proceed. Therefore, the informational log call and its parameter expressions are not evaluated at all.

[2]In Java, String objects are immutable, therefore, to do string concatenation, *StringBuffer*, which is a mutable string, should be used. Strings can be appended to the buffer and in the end, the result can be retrieved by a call to *toString()*. Java provides the + operator to make things easier.

3.2 Transactional Pointcuts

The basic observation behind transactional pointcuts is that join points do not occur in isolation, but rather, are parts of a higher-level computation that can in turn be regarded as a join point. For instance, when a file object is acquired at a constructor-call join point it is most probably going to be used for data input/output and then released by closing. This file operation pattern consists of a set of key related join points that can define the operation at some abstraction level. The traditional dynamic join point model cannot elegantly define, designate, and advise a set of interrelated join points.

AspectJ designers made a decision to make join points as context-insensitive as possible to make pointcuts more predictable to the programmer. We believe that in some situations suitably managed contexts can deliver more powerful aspects with an acceptable loss of reliability, hence, transcuts bring some context into the join point model.

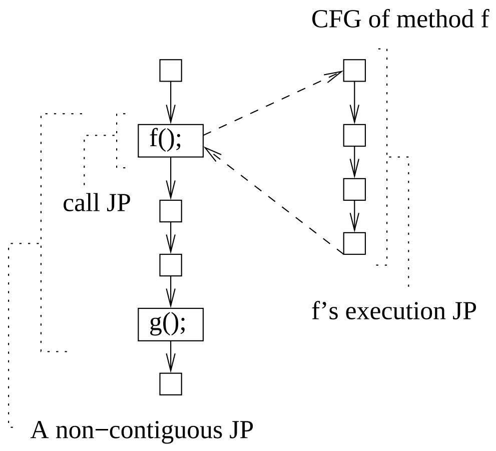


Figure 3.3: An example of three different join points: *call* and *execution* join points as well as a region join point that could potentially be a match for f(), g() sequence.

To specify a dynamic join point model, three elements need to be defined: join points, pointcuts, and advice. Join points in our new model are arbitrary pieces of computation whose shadows belong to well-defined single-entry-single-exit regions of the control flow graph (see Section 3.3.1 for the definition of region). Figure 3.3 shows a control flow graph that contains calls to methods *f()* and *g()* along with other nodes. *call* and *execution* join points are probably the most important and well-understood join points in a dynamic join point model. As it is annotated in the figure, the *call* join point is a "point" in the CFG[3]; however, the execution join point is not a point but a set of points between the entry of the CFG to the exit. This observation is interesting because at first, the idea of extending the

[3]Dynamic join points are execution-time concepts, while the compiler works with join point shadows [28].

definition of join point to arbitrary (single-entry-single-exit) regions of a control flow graph seemed a little controversial. However, the existing *execution* join point is one realization of this notion. In Figure 3.3, the region that begins with a call to method *f()* and ends with a call to method *g()* can be designated and advised as a join point. We refer to an arbitrary (possibly) non-contiguous segment of a region in the control flow graph as a *Region Shadow* and the corresponding join point as a *Region Join Point*.

A transcut is a special join point designator that selects sets of interrelated join points. Each match of a transcut is a set of join points that are related through control flow, dataflow, or both. The transcut definition begins with the *transcut* keyword followed by the transcut identifier and a list of parameters. A transcut is defined by a sequence (or pattern in general) of pointcuts that match individually almost similar to a traditional pointcut. A transcut only matches a piece of computation when all of its constituent pointcuts match some corresponding key join point in the computation. When a transcut contains only one pointcut, it does exactly what the contained pointcut does. Transcuts are useful when they contain more than one pointcut, the simplest form of which is a linear sequence of pointcuts. Figure 3.4 shows a transcut that would match computations that are generated by regions of code that have a call to *a()* followed by a call to *b()* followed by a call to *c()*: [4]

```
transcut abc() {
  pointcut a: call(* *.a());
  pointcut b: call(* *.b());
  pointcut c: call(* *.c());
}
```

Figure 3.4: A simple transcut composed of 3 pointcuts.

This is a very simple transcut that only captures the implicit control-flow relationship among the join points. It might seem that the transcut matches when a trace of events matches an *a(), b(), c()* sequence. That is the behaviour expected from trace-based mechanisms (see Section 6 for comparison), but that is not the meaning of this transcut.

The *abc()* transcut identifies pieces of computation that are known to the programmer with three points of reference: the beginning of the computation, *a()*, its end, *c()* and some point in between, *b()*. A piece of computation is a match only if it is possible to find out (at the beginning of its execution, at least) that its key points of reference (join points) will occur. Consequently, there can be a computation that has these join points in it but is not a match. For example, consider the following piece of code:

[4]Most of the examples in this section are simple and their purpose is to illustrate the concepts or function of the constructs. More realistic examples and applications are presented in Section 3.4.

```
a();
if(c()) {
  b();
}
c();
```

In some control flow paths, the generated computation has in fact all three key join points in it, however, it will not be known until the control is inside the computation, at which point, only *after()* advice can be applied, which is basically what trace-based mechanisms can do. Transcuts match a piece of computation only if it can be determined, by looking at the continuation of the head join point of the computation, that the rest of the join points will be present in the continuation.

The transcut reifies these identified computations as join points, and consequently, can be advised in the same way as traditional join points (all three major types of advice are supported). The points of reference in computations, which are join points themselves, are in reality related. In the current design, these join points have to be in the same control flow graph (method) to be visible to the matching algorithm.

Transcuts do not match against the source code; neither do they match against the execution trace. Transcuts match at the level of the control flow graph. Understanding the semantics of transcuts requires the concept of a region of control dependence which is explained in Section 3.3.

The pointcuts that constitute the transcut are defined almost similar to AspectJ pointcuts with one difference: they do not have separate parameter lists for exposing context but all share the transcut's list of parameters.

This shared context is used to establish dataflow relationships among various join points. For example, the *abc()* transcut would capture a sequence of *a(), b(), c()* even if they are called on different objects. While this behaviour might be useful in some cases, many situations require that the target of the designated join points be the same object. This modified version captures the *a(), b(),* and *c()* call join points only when they have the same target (Figure 3.5.)

```
transcut obj_abc(Object obj) {
  pointcut a: call(* *.a()) && target(obj);
  pointcut b: call(* *.b()) && target(obj);
  pointcut c: call(* *.c()) && target(obj);
}
```

Figure 3.5: A transcut that relates the join points based on their target object.

The context variable *obj* establishes a dataflow relationship between the three designated join points. The *target()* pointcut in the context of transcuts does two things: exposes the context values (which is what it does in AspectJ) and enforces a must-alias relation on the target object of the participating join points.

From another perspective, transcuts make it possible to define new join point types. Traditionally, the set of selectable join points is predefined by the language (*e.g.* method call, field get/set, *etc.* in pointcut-advice join point model.) Transcuts define new types of join points by composing them. In some sense, transcuts define new interfaces into code which advice can affect.

3.2.1 Nested Transcuts

A transcut can be used within another transcut which makes the composition and reuse of transcuts possible. Figure 3.6 shows an example.

```
transcut openClose() {
  pointcut open: call(* *.open(..));
  pointcut rw: readWrite();
  pointcut close: call(* *.close(..));
}
transcut readWrite() {
  pointcut read: call(* *.read(..));
  pointcut write: call(* *.write(..));
}
```

Figure 3.6: An example of transcut nesting

3.2.2 Looped and Conditional Pointcuts

There are situations in which a transcut should be sensitive to looped or conditional join points. For instance, if one needs to capture join points that begin with an *open()*, contain a loop in which a *read()* is called, and end in *close()*, the transcut in Figure 3.7 can be used.

```
transcut readloop() {
  pointcut open: call(* *.open(..));
  pointcut read: looped(call(* *.read(..)));
  pointcut close: call(* *.close(..));
}
```

Figure 3.7: Using *looped()* pointcut in a transcut

looped() is a new pointcut that matches a sub-region in a region of control flow if the sub-region is a loop and the given pointcut matches somewhere in the body of the loop. The above transcut matches the join point shadow in Figure 3.8 on the left but does not match the one on the right because *looped()* pointcut finds the first loop and looks into its body for a shadow that matches *call(* *.read(..))* but finds another loop. If ones requires a deep matching behaviour, then *looped *(pc)* should be used instead, where pc is the pointcut that should be matched inside a loop at some level down the top-level region.

```
open();
while(cond())
    read();
close();
```

```
open();
while(c1())
    while(c2())
      read();
close();
```

Figure 3.8: A match for *readloop()* (left) and a non-match (right).

Similarly, the *conditional()* pointcut can be used to express interest in join points that may (conditionally) occur. An example is shown in Figure 3.9, which matches the shadow on the left in Figure 3.10 but does not match the one on the right. To match a join point conditionally at any level, *conditional *(pc)* should be used, where pc is the pointcut that should match conditionally.

```
transcut readConditionally() {
  pointcut open: call(* *.open(..));
  pointcut read: conditional(call(* *.read(..)));
}
```

Figure 3.9: Using *conditional()* pointcut in a transcut

```
open();
if(success())
    read();
```

```
open();
if(success())
    if(c1())
      read();
```

Figure 3.10: A match for *readConditionally()* (left) and a non-match (right).

Looped() and *conditional()* pointcuts can be composed to create new transcuts that can express various composite join point relationships, as shown in Figure 3.11, which can potentially matches the code in Figure 3.12 (if the calls are all on the same object.)

Figure 3.13 shows an advice that targets the transcut in Figure 3.11 and simply executes the original join point. The purpose of this example was to show the composition of *looped()* and *conditional()* and does not have any special purpose.

3.2.3 Dependent Pointcut

Dependent pointcut, *dependent(),* is the generalized form of matching join points that execute conditionally, that is, either in a loop body or in a conditional, at any regional depth.

```
transcut loopedConditionalRead(Object obj) {
  pointcut open: call(* *.open(..)) && target(obj);
  pointcut loopedRead: looped(conditional(readData(obj)));
  pointcut condRead: conditionalReadLoop(obj);
  pointcut close: call(* *.close(..));
}
transcut readData(Object obj) {
  pointcut a: call(* *.readA(..)) && target(obj);
  pointcut b: call(* *.readB(..)) && target(obj);
}
transcut conditionalReadLoop(Object obj) {
  pointcut read: conditional(looped(call(* *.read(..))
                                    && target(obj)));
}
```

Figure 3.11: Composition of *conditional()* and *looped()* pointcuts.

```
open();
while(running()) {
    if(ready()) {
        readA();
        readB();
    }
}
if(success()) {
    while(ready())
        read();
}
close();
```

Figure 3.12: A match for *loopedConditionalRead*.

```
void around(Object obj): loopedConditionalRead(obj) {
  proceed(obj);
}
```

Figure 3.13: An advice that targets a transcut and simply executes the original join point.

This pointcut is a powerful and syntax-independent way of expressing conditional execution of a join point. For instance, consider the different ways a read operation can occur in a program in Figure 3.14, all of which can be designated with the transcut in Figure 3.15.

```
open();
while(running())
      if(ready())
            read();
}
close();
```

```
open();
if(ready())
      while(running())
         read();
close();
```

```
open();
if(ready())
      read();
close();
```

Figure 3.14: Three different matches for the same transcut.

```
transcut read() {
  pointcut open: call(* *.open(..));
  pointcut read: dependent(call(* *.read(..)));
  pointcut close: call(* *.close(..));
}
```

Figure 3.15: A transcut using *dependent()* pointcut.

3.2.4 Dataflow Pointcuts and Context Exposure

The traditional context-binding dynamic pointcuts (*i.e. this(), target(), args()*) bind their variables to the corresponding values exposed by join points. For instance, when an *args()* pointcut is matched against a method-call join point, it binds its variables to the respective arguments of the method call (which are already evaluated and available). Therefore, if the join point matches, the exposed context can be used (and/or altered) in advice.

The context variables exposed by a transcut, however, could be bound by a context-binding pointcut at any point within the transcut. The problem is that not all the join points designated in a transcut are activated at the time the transcut is being matched and advised, therefore, their context values might not be available yet to be exposed; in other words, an exposed context variable might not have any meaningful value if its value is being defined within the join point. Transcut parameters are dual purpose: they are used both for context exposure (as in traditional pointcuts) and for establishing data-flow relations between join points (there might be opportunity for improvement here by separating the two roles.) In the latter case, the transcut binds its variable to the compile-time representation of the context value (a local variable in the intermediate code) and makes it possible to track the data-flow using a must-alias analysis.

In general, it is meaningful to expose a context value only if the value is generated outside of and used inside the region join point and remains unchanged throughout the join point. This definition of context ensures that the exposed context value makes sense right at the time the region join point is activated (which is the beginning of the designated computation). An example is *logger* context variable in Figure 3.2, which is bound by the *target(logger)* pointcut at line 17.

The *return()* pointcut, which is similar to the one defined in [39], binds its parameter to the returned value of a join point, which does not exist before the join point execution.

3.2.5 Overlapping Transcuts

The designated join points of multiple transcuts can overlap without any restriction because pointcuts do not have computation effects [18] and transcuts are no exception. However, two transcuts can simultaneously receive *around()* advice only if their designated join point shadows are disjoint or one is contained in the other (the head or tails of the shadows can overlap). *Before()* and *after()* advice are applicable even when the transcuts overlap.

```
transcut abc() {
  pointcut a: call(* *.a(..));
  pointcut b: call(* *.b(..));
  pointcut c: call(* *.c(..));
}
transcut be() {
  pointcut a: call(* *.b(..));
  pointcut b: call(* *.e(..));
}
```

Figure 3.16: Two overlapping transcuts

The transcuts in Figure 3.16 match two overlapping regions of the following code:

```
a(); b(); c(); e();
```

Therefore, *around()* advice is not allowed, if both are advised. However, *before()* and *after()* advice can be applied.

It it worth mentioning that these kinds of situations cannot be handled through refactoring because two different views of the target computation are needed simultaneously.

3.3 Program Dependence Graph: the Join Point Representation

Program Dependence Graph (PDG) [21] is a program representation in which both control flow and data flow relationships among program operations are made explicit. PDG has successfully been used for program optimization, parallelization, slicing, automatic testing, *etc.* all of which require that dependencies among program statements be easily accessible. Nodes in PDG are the same as the nodes in Control Flow Graph (*i.e.* basic blocks, or at a lower level, instructions). Edges in PDG denote control dependency and data dependency between two nodes. Control flow information is implicitly available in PDG through ordering, but it can also be explicitly represented through control flow edges.

As explained in the previous section, transcuts capture the control flow and data flow relationships among join points; therefore, PDG, quite naturally, is the most suitable program representation for realizing transcuts because it makes both types of relationships explicit and readily available. In this section, we illustrate a PDG of a simple program and explain how it is constructed.

Figure 3.17 shows a program and its CFG, Figure 3.18 shows its dominator and post-dominator trees, and Figure 3.19 shows the corresponding PDG. Only control flow dependencies are shown in the PDG because we currently do not use explicit data flow edges in PDG.

```
void m()
{
   a();
   while(c1())
   {
      b();
      while(c2())
         d();
      if(c3())
         e();
      else
         e2();
      f();
   }
   g();
}
```

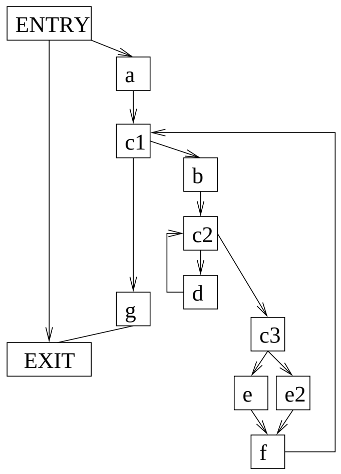


Figure 3.17: A program and its CFG

Informally, node B is control dependent on node A if the execution of A determines whether B executes or not. The formal definition from [21] is as follows. Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

1. there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and

2. X is not post-dominated by Y.

For instance, method call *b()* is dependent on *c1()* because there is a path from *c1()* to *b()* that only contains *b()* and *c1()* (therefore the first condition holds because there is no Z in the path), and *c1()* is not post-dominated by *b()* (see Figure 3.18). Intuitively, *b()* is control dependent on *c1()* because the execution of *b()* depends on the result of the execution of *c1()*. But *c1()* is not control dependent on *a()* because *a()* is post-dominated by *c1()*. Intuitively, *c1()*'s execution is not dependent on *a()*'s execution.

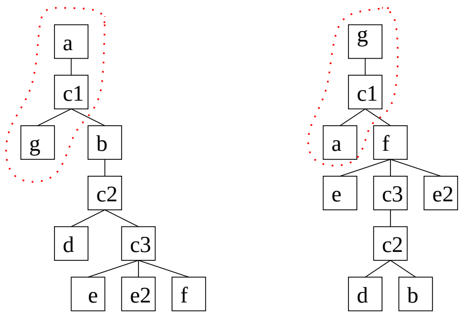


Figure 3.18: Dominator (left) and post-dominator (right) trees

Nodes in a PDG can be CFG nodes or region nodes and there is an edge from A to B if B is control-dependent on A. A region node summarizes and factors out the set of control dependences of a set of nodes in a PDG. For instance, all the nodes within the body of the top-level loop in Figure 3.17 are control dependent on *c1()*; so, a region node can represent this shared dependence set: region node *R2* in Figure 3.19 is created and made control dependent on *c1()* and all the nodes in the body of the loop are made control-dependent on *R2*. This dependence set summarization is performed for all control dependences and the created region nodes are added to the PDG. At each level of the PDG, nodes are ordered from left to right according to the flow of control. In other words, control flow in each region in PDG (*i.e.* the child nodes of the region) is from left to right.

Initially, we used weak regions (defined in the next section) as the basis of transcut matching, however, we did not have a PDG as the program representation, therefore, the relationships between regions were not available to the matching algorithm.[5] PDG makes the region hierarchy explicit and available for matching. For instance, inner loops can easily be identified (*e.g.* region *R3* is the inner loop in region *R1*).

Various algorithms have been proposed to efficiently compute the PDG[6] of a program (*e.g.* [21], [12], [27], [6]). We reuse the region analysis machinery that we initially implemented to build weak regions (based on the algorithm in [6]) and construct the PDG based on the algorithm given in [21].

[5]For example, the region corresponding to a loop body and the parent region that contained the whole loop were considered independently for matching.

[6]Some only compute the Control Dependence Graph and some only the set of regions.

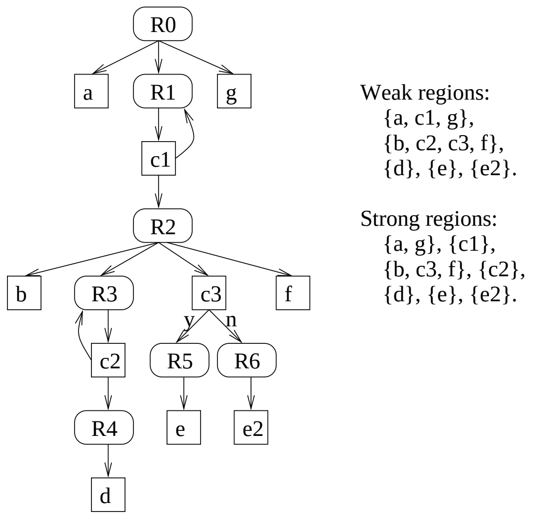


Figure 3.19: PDG of the program in Figure 3.17

3.3.1 Region Analysis

In any execution path from the beginning of the flow graph to the end, either all the nodes in a region execute or none of them do. Regions are, therefore, the natural extension of basic blocks (the control enters through the header of a block and exits through the end). Regions can be *weak* or *strong*. We use these definitions from [45], which result in slightly different regions than the ones defined in [1].

- **Weak Region**: vertices v and w are in the same weak region iff for any complete control-flow path, v and w are both in the path or are both absent from the path.

- **Strong Region**: vertices v and w are in the same strong region iff v and w occur the same number of times in any complete control-flow path.

We used an algorithm presented in [6] which finds weak regions based on the observation that

- v and w are in the same weak region iff (v dominates w and w post-dominates v) or (w dominates v and v post-dominates w).

Strong regions would be the same as weak regions if there were no loops in the CFG. Distinct vertices v and w are in the same strong region iff

- they are in the same weak region **and**

- (v is in every cycle containing w) and (w is in every cycle containing v.)

The weak and strong regions of the CFG in Figure 3.17 are shown in Figure 3.19. Note that the regions in the PDG correspond to the strong regions. The linear algorithm given in [6] is based on the key observation that for any CFG the vertices of each weak region form a chain in the post-dominator tree that is the reverse of a chain in the dominator tree (see Figure 3.18 for the chains corresponding to the weak region $\{a, c1, g\}$).

After computing the weak regions, we compute the strong regions and construct the PDG at the same time. The outline of the algorithm is as follows. Given the list of weak regions, PDG can be constructed by finding the inter-region dependencies. Starting from the top-level weak region, a top-level PDG node, R, is created and for each (CFG) node A in the region, first, a PDG node is created to represent A in the PDG, then, a dependency edge is added from R to the PDG node representing A.

Then the set of nodes that are dependent on A are found: for each edge (A,B) in the CFG such that B does not post-dominate A, let L be the least common ancestor of A and B in the post-dominator tree. Either L is A or L is the parent of A in the post-dominator tree (see [21] for proof). If L is the parent of A, then all nodes in the post-dominator tree on the path from L to B, including B but not L, are control dependent on A. If L is A, then all nodes in the post-dominator tree on the path from A to B, including A and B, are control dependent on A (this case captures loop dependence.) Both cases can be covered by traversing backwards from B in the post-dominator tree until we reach A's parent (if it exists, or A otherwise) and adding all visited nodes to a list as nodes that re control dependent on A.

The A's PDG node is changed to be a “Conditional” PDG node to represent the fact that there are nodes that depend on it; then, for each of the dependants of A, the containing (weak) region is looked up[7] and a PDG node is created to represent it; a dependency edge is then added from the A's PDG node to the region's PDG node. This step is repeated for all the nodes in the list of dependants that are in a different weak region than the previously processed dependants of A. Loops that contain abrupt exit or continuation statements cause some conditions that need to be checked in the above steps. When it turns out that a loop header is being processed, a new strong region is created along with its corresponding PDG node which is added to the graph and the appropriate dependency edges are added. It is worthwhile to mention that loops create circular dependencies in the PDG. For more details on how PDG is constructed and also how we handle non-normative control flow see Appendix A.

[7]This information is available from the region analysis phase.

Regions have an important property that makes them very useful in realizing transactional pointcuts: if the normal flow of control enters the region (which occurs only through the head node of the region), it will go through all the nodes in the region, and eventually exit through the tail node of the region; this property is similar to the property of basic blocks [1] with the difference that regions can consist of non-contiguous pieces of code. In other words, while the control is within a region, other regions might be activated but the control would eventually return to the original region.

3.3.2 Join Point Shadows

Join points in our model are either

- traditional call, constructor, or set/get join points, or
- a set of related join points that occur in the same region of control dependency.

The above definition is recursive, in the sense that, once join points are selected using a transcut in our model, they can be used in other transcuts to designate other composite join points.

We match a transcut against all the potential join point shadows. A join point shadow in our model is a Single-Entry-Single-Exit (SESE) segment of a control dependence region in the PDG. For example, the segment of code that begins with *a()* and ends in *g()* is a SESE sub-region of *R0* in the PDG in Figure 3.19. Similarly, the sub-region that begins with *c3()* and ends in *f()* is a SESE sub-region of *R2*, and so on. To instantiate all the potential region shadows in a method, we consider all the regions in the PDG and make a linear list of nodes, ordered based on control flow, in that region (*e.g. [a(), c1(), g()]* in *R0*). Each sub-region of such a linear list is a SESE sub-region and should be considered a join point shadow and be checked for potential matches. There could be different ways of instantiating these sub-regions and each approach might affect the semantics of matching. In our current approach, each node in the region is the start of a shadow and the end of the shadow is determined at matching time. Figure 3.20 shows a SESE region and its potential shadows.

The matching is aware of the PDG structure, that is, region hierarchy, loop structure, conditional regions. In fact, without PDG it would not be possible to support nested, looped, and conditional transcuts.

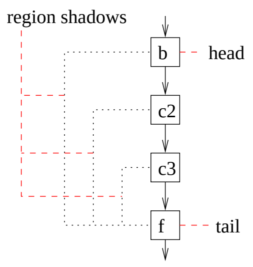


Figure 3.20: A region and its potential shadows

3.3.3 PDG-based Matching Algorithm

Figures 3.21, 3.22, and 3.23 show a simplified version of the PDG-based matching algorithm. For brevity and readability, many details (types, parameters, context variable tracking, *etc.*) are omitted. Also, only the conditional pointcut matching is presented as the loop and dependent pointcuts are similar. Some comments are added in the algorithm to make it easier to understand. A more narrative description follows.

The algorithm is divided into three procedures to make it easier to present. The top-level procedure *match(tc, rsm)*, matches the transcut *tc* against the region shadow *rsm* and returns a residue that can be either *AlwaysMatch*, *NeverMatch*, or primitive operation such as binding a variable to a value, or a combination thereof. A residue represents the code that need to be inserted before a join point to be executed at runtime. Needless to say, *AlwaysMatch*, and *NeverMatch* represent no code but are used to determine if there is definitely a match or no match.

After some initialization (explained inside the figure), a loop iterates through the ordered list of pointcuts that constitute the transcut; each pointcut is matched against potential positions in the region starting at location i which is the head of the current shadow, rsm (line 29, *matchPointcutInRegion()*). If a match cannot be found, then the transcut matching returns with no matching; otherwise, the returned residue is combined with the residues of the previous constituent pointcuts' residues (line 40). If the loop reaches to the end of the list of pointcuts, that means there is a match and the accumulated residue is returned.

The index of the current matching target within the region, i, which represents the index of the current matching target within the region, is boxed before being passed to other matching procedures so that it can retain applied changes. There is another boxed variable of type boolean that is used to track when the head of the shadow is matched. This variable is necessary because, in our implementation, each instruction is the head of a new shadow, as discussed previously, and it has to match the current pointcut if the current pointcut is

```
Residue match (Transcut tc, RegionShadowMatch rsm) {

  Residue res = AlwaysMatch;
  pointcuts = ordered list of constituent pointcuts in tc;

  if (pointcuts is empty)
    return NeverMatch;

  region = the region containing the shadow, rsm;
  units = the ordered list of instructions in the region;

  //boxed integer to keep track of the current position in the region
  i = index of the head of the shadow in the region;

  //boxed boolean, set to true when the head of the shadow is matched
  headMatched = false;
  /*
    Iterate through the constituent pointcuts and match one by one.
    There is a match if the last pointcut matches within this shadow.
  */
  while (not reached the end of the list of pointcuts) {

    currentPC = next pointcut in the list;

    If (reached the end of the shadow)
      return NeverMatch;

    //Match the current pointcut in the region from the current position
    r = matchPointcutInRegion (tc, currentPC, region, i, headMatched);

    if (r == NeverMatch)
      return r;

    /*
      At this point, the current pointcut has matched the current
      position in the region; combine the new residue and the
      previous one; continue to the next pointcut and position.
     */

    res = combine (res, r);
  }

  /* At this point, a match has been found.*/
  return res;
}
```

Figure 3.21: Transcut Matching Algorithm

the first pointcut in the transcut.

Figure 3.22 shows the algorithm for procedure *matchPointcutInRegion()* called from *match()*. At the beginning of the procedure, after some initialization, the current index, i (which is copied to j), is checked to see if it is the end of the current region, in which case the matching fails. This check is done at the beginning of almost all of the matching procedures because these procedures might be called mutually recursively form within other procedures.

From line 10 to 17, depending on the type of the current pointcut, the proper matching procedure is called. For example, if the pointcut is a *conditional()* pointcut, then *matchConditionalInRegion()* is called whose algorithm is depicted in Figure 3.23. If the pointcut is none of the dependent pseudo pointcuts (*i.e. looped(), conditional(), or dependent()*), then the matching continues in the same procedure by iterating through all instructions from the current index, j, to the end of the current region (line 17.)

Each instruction can be the beginning of different kinds of method positions, such as statement position and region position, for each of which a special position object is created (line 20, which, for simplicity, is showing only one position object;) then a procedure is called (*doShadows()*) that goes through all different kinds of shadows to see if the position can be a candidate for any of them. If so, a shadow object is created for that position and matched against the currentPC pointcut. If a match is found, the corresponding residue is returned. For brevity, details have been omitted.

After a match is is found, if currentPC is a transcut itself (*i.e.* a nested transcut, which has matched), then the newly-bound context variables should be checked to be compatible with the previously-bound ones (line 29.) The context variables are the variables that are declared by the transcut definition and bound within it. A must-alias analysis is used to find out whether the new bindings are compatible with the old ones. The bindings are saved in the transcut that is being matched (lines 37, 48.) A *NeverMatch* is returned if the bindings are not compatible.

Whether the current pointcut is a transcut or a traditional pointcut (*call()*), upon successful matching, the current matching index is adjusted accordingly (line 36, 47), the context is saved (lines 37, 48), the *headMatched* flag and the i box are set if needed (lines 39-42, 50-53), and finally, the residue is returned (line 43, 54.) Otherwise, if the current position does not match the current pointcut, then the accumulated context bindings in the transcut are cleared (line 60), and if this match was supposed to be a head match but the *headMatched* flag is false, then the matching fails; otherwise, the loop continues with the next instruction.

```
Residue matchPointcutInRegion (tc, currentPC, region, i, headMatched) {

  units = the ordered list of instructions in the region;
  //Note that i is a boxed integer and j is a normal integer
  j = i;

  If (reached the end of the shadow)
    return NeverMatch;

  if (currentPC is a Loop pointcut) // looped()
    return matchLoopInRegion(tc, currentPC, region, i, headMatched);
  else if(currentPC is a Conditional pointcut)  //conditional()
    return matchConditionalInRegion(tc, currentPC, region, i, ...);
  else if(currentPC a Dependent pointcut) //dependent()
    return matchDependentInRegion(tc, currentPC, region, i, ...);

  else for(all instructions from j to the end of region) {

    current = the j'th instruction in the region;
    pos = the position of the current instruction in the method;
    r = doShadows(currentPC, pos);

    if(r != NeverMatch) {

          /* If currentPC is a transcut, check the newly-bound context
          variables to be compatible with the previously-bound ones.*/

      if (currentPC is a Transcut) {
        r = checkBoundContextVars(tc, currentPC, r);

        if(r != NeverMatch) {

          /* Matched nested transcut... moving on to the
          next pointcut/statement */

          j = the index of the last matched instruction + 1;
          save bound context in tc;

          if(!headMatched)
            headMatched.setValue(true);

          i.setValue(j);
          return r;
        }
      }
      else {
        j = the index of the last matched instruction + 1;
        save bound context in tc;

        if(!headMatched)
          headMatched.setValue(true);

        i.setValue(j);
        1 r;
      }
    }
    //Not matched ...
    //clear the temporary bound variables in the last try
    clear bound context in tc;

    /*If this was supposed to be a head match but the flag still
    is false, then it is a no match. */

    if(!headMatched)
      return NeverMatch;
  }
  /* Reached the end of the region without matching all the pointcuts;
    so, return no match. */
  return NeverMatch;
}
```

Figure 3.22: Transcut Matching Algorithm (matchPointcutInRegion)

If the loop reaches the end of the region while trying to match the current pointcut, the matching returns *NeverMatch* (line 74.)

Figure 3.23 shows the procedure for matching a conditional pointcut in a region. Essentially, the PDG nodes in the region are iterated, starting from the node corresponding to the current instruction (i), until a conditional PDG node that is not a loop header is found. Then, for each PDG node dependent on the conditional node, the previously discussed *matchPointcutInRegion()* is called to match the inner pointcut of the conditional pointcut (currentPC) within the dependent region represented by the dependent PDG region; that is, for a conditional pointcut, which has the form *conditional(pc)*, *matchPointcutInRegion()* is called to match pc in the dependent region.

The matching index and the boolean flag indicating a head match need to be created, set to their default values, and passed in the matching procedure (lines 28-31) because this matching is in a new region. If the inner pointcut does not match within the dependent region, then if the conditional pointcut is in deep matching mode (*i.e. conditional *(pc)*), another matching effort is made: this time, *matchConditionalInRegion()* is recursively called to match for the current conditional pointcut (currentPC) within the dependent region, to give a chance to all the regions that are nested deep in the region (lines 34-38.)

In any case, when a match is found, then the next instruction from where the matching should continue is determined and the i index is adjusted accordingly. It is worth mentioning that when a match is found, the tail of the shadow match need to be adjusted due to different way dependent pointcuts match. Basically, the first and last point of the conditional shadow match should be adjusted in such a way that it mimics a reduction of the conditional node in the program. The intuition is that, if the shadow is not executed, it is as if the conditional was never there. The same goes with loop matches. When constructing the PDG, the flow relationships between the PDG nodes are determined as well; therefore, once the current conditional PDG node (cnode) is found out to be a match for the current conditional pointcut, then next PDG node in the flow is retrieved whose head instruction will be the next matching index (44-46.) The rest of the matching is similar to the previous procedure.

3.4 Applications

In this section we present a few examples to show the power of transcuts to address some familiar problems.

```
Residue matchConditionalInRegion(tc, currentPC, region, i, headMatched) {

  units = the ordered list of instructions in the region;
  j = i;

  If (reached the end of the shadow)
    return NeverMatch;

  pdgNodes = ordered list of PDG nodes in the region;

  curPDGNode = the PDGNode corresponding to the current instruction;

  /* If this is supposed to be a head match */
  if(!headMatched) {

    if(curPDGNode is not a Conditional node
      or the current instruction at index i is
      not a branching instruction)
      return NeverMatch;
  }

  //from the current PDG node to the end of the list
  while(there is a node (cnode)) {
    if(cnode is Conditional and not a loop header) {
      foreach (dependent node (depNode) of cnode) {

        dependentRegion = the region corresponding to the dependent node
        createNewBoxes: i2box = 0, headMatched2 = false;

        r = matchPointcutInRegion (tc, currentPC.getDependentPointcut(),
                    dependentRegion, i2box, headMatched2);

        //Go deep in the conditional regions?
        if((r == NeverMatch) and (currentPC is a deep pointcut) {
          create and initialize boxes (as above);
          r = matchConditionalInRegion(tc, currentPC, dependentRegion,
                        i2box, headMatched2);
        }
        if(r != NeverMatch) {

          /* find the next instruction from where matching should
          continue*/

          nextInFlow = the PDG node next in the control flow of cnode
          firstOfNex = the first instruction in nextInFlow

          j = index of firstOfNext in the region  + 1;
          tc.saveBoundContext();

          if(!headMatched)
            headMatched.setValue(true);

          i.setValue(j);
          return r;
        }
      }
      /* If this was supposed to be a head match but did not match
      the first node, then this cannot be a match. */

      if(!headMatched.getValue())
        return NeverMatch;
    }
      }
  return NeverMatch;
}
```

Figure 3.23: Transcut Matching Algorithm (matchConditionalInRegion)

3.4.1 Modularizing Exception Handling

Exception handling concerns cross-cut the implementation of the normal behaviour of a system. In Java and many other languages repetitive exception handling code is tangled with the normal code. But some studies [36] show that the number of reactions to different exceptions is considerably lower than the number of places exceptions are caught, therefore there is the opportunity for reuse of the handling patterns.

Writing flawless exception-handling code is hard. As reported in [57], many programs fail to properly release acquired resources along all execution paths in the presence of run-time errors. Many programmers that are aware of exceptions and use proper constructs to catch and handle them, still write faulty exception-handling code. Writing correct exception-handling code becomes even more difficult when the number of resources that need to be handled increases (*e.g.* a database connection, a query statement, and a query result set are three resources typically involved in a database operation). Correctly dealing with N resources typically requires N nested try-finally statements or a number of run-time checks to track if resources are still allocated.

Therefore, it makes sense to factor exception handling code into a separate module. However, traditional programming languages, such as C++ and Java, do not support such separation. AOP languages (*e.g.* AspectJ) provide facilities to achieve this separation to some extent. As an example, Figure 3.24 shows an AspectJ program that captures *IOException*'s that are raised inside the methods of *Loader*.[8] With the help of this aspect, the original code is shorter and more readable. Tangible results of applying aspects for handling exceptions in a real code base are reported in [36].

The limitations in AspectJ's join point model prevent us from fully separating exception handling code from the base code. If the target piece of code is a single join point, then it can be handled in AspectJ. But if it consists of more than one join point, its exception handling cannot be modularized in AspectJ without refactoring, because there is no join point corresponding to the execution of an arbitrary block of code. This limitation, that AspectJ does not provide the necessary means to capture and handle exceptions inside method boundaries, is mentioned in [36] but no solution has been proposed. [8] shows that refactoring these target blocks to expose them as join points results in low cohesion among other things.

[8]In Java, potential checked exceptions must be either handled or thrown; to get around this problem and handle exceptions in an aspect, AspectJ provides a work-around and that is to *declare* that a checked exception is *soft*. When an exception is softened, a *SoftException*, which is an unchecked exception, is thrown instead of the original exception.

```
aspect ExceptionHandlingAspect {

  pointcut loadMethods():
        execution(* Loader.load*(..));
  declare soft: IOException: loadMethods();

  void around(): loadMethods() {
    try {
      proceed();
    }
    catch(IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Figure 3.24: Handling an exception is an aspect

Figure 3.25 shows an example program that performs an input operation within a method (from [50].) During the execution of *someMethod()*, an *IOException* could be raised somewhere between the creation of the file input reader up to the point that the reader is closed (lines 12-16). Although in AspectJ one can wrap *try-catch* block around captured join points, there is no join point corresponding to the execution of an arbitrary block of code. Transactional pointcuts provide a solution; Figure 3.26 shows an aspect that implements the recommended practice for handling resources in the presence of exceptions.

```
 1 public class BufferedReaderExample {
 2
 3   public static void main(String[] args) {
 4     BufferedReaderExample example = new BufferedReaderExample();
 5     example.someMethod();
 6   }
 7
 8   public void someMethod() throws IOException {
 9     doSomeWork();
10
11     //now read some data ...
12     BufferedReader reader = new BufferedReader(new FileReader("out.txt"));
13     String str = null;
14     while ((str = reader.readLine()) != null)
15       process(str);
16     reader.close();
17
18     doSomeOtherWork();
19   }
20 }
```

Figure 3.25: A file operation within method boundaries that could throw exceptions

The transactional pointcut *fileInputOp* (lines 7-12) designates the target piece of file opening-processing-closing code, which could leak the reader resource if an exception occurs and is not properly handled. The aspect contains a field, *m_bufReader*, of type *BufferedReader* to keep track of the actual *BufferedReader* object from the creation point to the

```
 1 aspect ExceptionAspect percflow(execution(* BufferedReaderExample.*(..))) {
 2
 3   declare soft: IOException: call(* BufferedReaderExample.*());
 4
 5   private BufferedReader m_bufReader = null;
 6
 7   transcut fileInputOp(BufferedReader bufReader, FileReader fileReader) {
 8     pointcut create: call(FileReader.new(..)) && return(fileReader);
 9     pointcut bufCreate: call(BufferedReader.new(..)) && args(fileReader)
10                                 && return(bufReader);
11     pointcut close: call(* BufferedReader.close()) && target(bufReader);
12   }
13
14   pointcut bufferCreate(): call(BufferedReader.new(..))
15                            && cflow(execution(void BufferedReaderExample.*()));
16
17   after() returning(BufferedReader bf): bufferCreate() {
18     m_bufReader = bf;
19   }
20
21   void around(BufferedReader bf, FileReader fr): fileInputOp(bf, fr) {
22     try {
23       proceed(bf, fr);
24       m_bufReader = null;
25     }
26     catch(IOException e) {
27       //Handle IOException if possible
28     }
29     finally {
30       //resource created and not released?
31       if(m_bufReader != null)
32         try{ m_bufReader.close();}
33         catch(Exception e){ //Handle it}
34     }
35   }
36 }
```

Figure 3.26: An aspect that implements recommended practice for handling resources

release point. The *after()* advice at lines 17-19, sets *m_bufReader* to the returned object of the successful constructor call. The *around()* advice at lines 21-35, which intercepts the execution of the target file operation (designated by *fileInputOp*), wraps the operation inside a *try-catch-finally* to capture any exception; if the target file operation finishes successfully (*i.e.* the execution gets to line 23), *m_bufReader* is set to *null* because, at this point, one can be sure that the BufferedReader.close(), in the target file operation, has executed successfully. *IOException* is handled at lines 26-28, and then in the *finally* block (that is, regardless of whether an exception is thrown or not), *m_bufReader* is checked to see if the object has been created and not released, in which case, it will be released (lines 31-34). If other types of exceptions (*e.g. NullPointerException, etc.*) are thrown somewhere in the operation, *m_bufReader* will still be properly handled and the unhandled exception will be propagated.

One might wonder why in the *bufferCreate* pointcut (lines 14-15) we keep track of the target *BufferedReader* object using a member field in the aspect, while the transcut binds one of its parameters to the object (Line 10). This is necessary because it only makes sense to expose the context values that do not change within the join point; the BufferedReader object is created within the join point, therefore its value cannot be exposed before the join point has even executed. In such cases, he context variables only act as data flow variables which are used to establish data flow relations among join points. Hence, the aspect keeps track of the created object manually through another variable.

3.4.2 Transaction Management

The program in Figure 3.27 connects to a database and updates a list of old values in a table with new ones. There are at least three important concerns that a programmer should have and handle when writing this code. First, most of the code statements can potentially throw checked exceptions, which need to be handled, because Java enforces the handling of checked exceptions unless the containing method's signature is explicitly annotated to throw them. Secondly, there are two resources in this code that should be released properly. If the code executes normally, the *close()* statements release the resources. But if an exception is raised anywhere and not properly handled, then one or two of the resources can leak. Finally, it might be required by the specifications that the values should be updated altogether and if not all of them can be updated (*e.g.* because of an error) then none of them should be changed. That is, the updates should be transactional.

```
Connection conn = DriverManager.getConnection(...);
PreparedStatement update =
          conn.prepareStatement("UPDATE names SET name = ? WHERE name = ?");
Iterator<String> itr = old2newMap.keySet().iterator();
while(itr.hasNext())
{
  String oldvalue = itr.next();
  String newvalue = old2newMap.get(oldvalue);

  update.setString(1, newvalue);
  update.setString(2, oldvalue);
  update.executeUpdate();
}
update.close();
conn.close();
```

Figure 3.27: A typical database client code

These three concerns could be implemented by inserting the necessary code in the client code, resulting in tangled code that is hard to maintain. Furthermore, in applications that work with databases there are many instances of code that is similar in pattern to the code in Figure 3.27.

```
 1 aspect TransactionAspect percflow(dbcut(Connection)) {
 2   private Connection m_con = null;
 3   declare precedence: ResourceHandlerAspect, TransactionAspect;
 4
 5   transcut dbUpdate(Connection con, PreparedStatement st) {
 6     pointcut createStmt: call(* Connection.prepareStatement(..)) && target(con)
 7                            && return(st);
 8     pointcut rsClose: call(* Statement+.close()) && target(st);
 9   }
10
11   transcut dbcut(Connection con) {
12     pointcut createCon: call(* DriverManager.getConnection(..)) && return(con);
13     pointcut rsClose: call(* Connection.close()) && target(con);
14   }
15
16   after() returning(Connection con) throws SQLException:
17         call(* DriverManager.getConnection(..)) && cflow(dbcut(Connection)) {
18     m_con = con;
19     m_con.setAutoCommit(false);
20   }
21
22   void around(Connection con, PreparedStatement st): dbUpdate(con, st) {
23     boolean success = false;
24     try {
25       proceed(con, st);
26       success = true;
27     }
28     finally {
29       if(success) {
30         try {con.commit();}
31         catch(Exception e1) {}
32       }
33       else {
34         try {con.rollback();}
35         catch(Exception e2) {}
36       }
37     }
38   }
39 }
```

Figure 3.28: Transaction handling transcut

Figure 3.28 shows an aspect that implements the transaction management for instances of database code that match the expressed behaviour. This behaviour, in this specific example, is captured by two transcuts, *dbUpdate()* (lines 5-9) and *dbcut* (lines 11-14). The former captures pieces of computation beginning with the creation of a database *Statement* (query) object and ending with the statement releasing the object[9]; and the latter does the same for the *Connection* object. An aspect instance is associated with each flow of a computation designated by *dbcut()* (*i.e.* instances of code similar to Figure 3.27).

The aspect uses an instance variable (*m_con*) to keep track of the associated *Connection* object, and also disables automatic commit on the connection so that the aspect can commit when appropriate (lines 16-20). Then an *around()* advice wraps the designated database query execution (*e.g.* for code in Figure 3.27, this corresponds to the execution of lines 2-

[9]In AspectJ, “Statement+” means the “Statement” type and its subclasses.

13) in a *try-finally* block and uses a flag to keep track of the success of the query execution.

If the *proceed()* at line 25 returns successfully (that is, the target query statement is executed with no error) then the success flag is set to *true*. Otherwise, if an exception is raised, the flag will be *false*. In the *finally* block, the transaction will be committed if the flag is *true* (query execution successful) or rolled back otherwise. The raised exceptions will still be propagated up from *TransactionAspect* so that the *ResourceHandlerAspect* (Figure 3.29) can handle the exception and resources.

```
aspect ResourceHandlerAspect  percflow(dbcut(Connection)) {
  declare soft: SQLException: call(* DBConnectionExample.*(..));
  Connection m_con = null;
  Statement m_stmt = null;
/*
  dbQuery() and dbcut() are the same as before...
*/
  after() returning(Statement stmt): call(* Connection.prepareStatement(..))
                                      && cflow(dbcut(Connection)) {
    m_stmt = stmt;
  }
  void around(Connection con, PreparedStatement st): dbQuery(con, st) {
    try {
      proceed(con, st);
      m_stmt = null;
    }
    finally {
      if(m_stmt != null) {
        try {
          m_stmt.close();
          m_stmt = null;
        }
        catch(Exception e) {}
      }
    }
  }
  void around(Connection con): dbcut(con) {
    try {
      proceed(con);
      m_con = null;
      }
    finally {
      if(m_con != null) {
        try {
          m_con.close();
          m_con = null;
        }
        catch(Exception e1) {}
      }
    }
  }
}
```

Figure 3.29: Transcut to handle resources

The exceptions and resources are handled in one single aspect as shown in Figure 3.29.[10] This aspect captures any exceptions raised for the whole computation from *Connection* cre-

[10]Obviously, there is opportunity for reuse between these two aspects, but that was not the point in this example.

ation, query execution, to *Connection* release. In this specific example, the only special handling occurs in the *finally* blocks, by checking the instance variable that tracks the corresponding resource and releases the resource if not released already. Note that at line 15 and 30, the instance variables are set to *null* to indicate that the target code has successfully executed to the end and that the corresponding resource is released.

3.4.3 Synchronization

Concurrent systems designers often desire customizable synchronization mechanisms because different composition and deployment contexts demand different synchronization policies. An AOP synchronization library like FlexSync [58] achieves customizability through decoupling synchronization intentions and mechanisms (three mechanisms are supported: Java *synchronized*, *Atomic Blocks*, and *Software Transactional Memory*). FlexSync requires refactoring to convert blocks into methods so that they can be picked out by pointcuts. Transcuts can be used to designate the pieces of code that are targets of synchronization without refactoring.[11]

Even if one single synchronization mechanism is used, customized policies for different contexts may still be desirable. For instance, the granularity of synchronization can affect performance. If large blocks of code are synchronized (locked) other threads might be blocked for long periods of time waiting for the thread that owns the lock to release it. On the other hand, if the granularity is too low, the overhead of the lock/unlock mechanism might affect performance, and ensuring consistency becomes tricky. Transcuts can be used to separate demarcation of the boundaries of a critical section from the code so that different boundaries could be used for different synchronization granularities (This cannot be accomplished by FlexSync because the critical sections are refactored into methods and hence are fixed.) The synchronization mechanism itself can then be customized once the transcuts designate the critical section.

Consider a typical shared *Buffer* object that is used by a number of reader and writer threads. One way of synchronizing the buffer is to make all of its read/write methods synchronized. The problem is that, if the buffer must be checked by the writers to make sure it is not full before writing (and similarly not empty before reading) then the buffer needs to be locked before checking and released after the actual write (read) operation, in order to make the buffer update atomic.

[11]The designer of FlexSync confirmed in a private conversation at ICSE'09 that transcuts can be very useful in FlexSync.

Transcuts can be used to express the synchronization intention (that is, the boundaries of the critical sections) as well as applying different synchronization mechanisms to the designated critical sections. Figure 3.30 shows parts of the code of two types of threads: a writer and a reader which write to and read from a shared buffer, respectively. Multiple threads of both types could be running simultaneously. We would like to have lines 2-3 of the writers and lines 2-5 of the readers execute atomically with respect to other threads. Figure 3.31 shows a transcut that captures the critical section of the reader threads and the advice that synchronizes the section. A similar transcut/advice pair is used (not shown) to synchronize the writer threads.

```
1 while(running) {
2   if(!buf.isFull())
3     buf.write(data);
4 }
```

```
1 while(running) {
2   if(!buf.isEmpty()) {
3     Object d = buf.read();
4     process(d);
5   }
6 }
```

Figure 3.30: Critical sections in writer (left) and reader (right) threads.

```
transcut readbuf(Buffer buf) {
  pointcut test: call(* Buffer.isEmpty()) && target(buf);
  pointcut read: conditional(call(* Buffer.read()) && target(buf));
}
void around(Buffer buf): readbuf(buf) {
  synchronized(buf) {
    proceed(buf);
  }
}
```

Figure 3.31: Transcut to synchronizes reader's critical section.

3.4.4 Parallelization

Can aspects be used on existing single-threaded code to take advantage of increasingly common multi-core systems? AOP can help, to some extent, to identify the desired join points and run them in a thread. However, the dynamic pointcut-advice AOP is stuck at the limits that were discussed before. Consider the following scenario.

Imagine a game engine that uses typed messages for communication and coordination among its components, such as graphics subsystem, physics subsystem, sound subsystem, and AI. For instance, of a missile is fired, a *PlaySoundEffectMessage* message needs to be created, configured with the proper values, and sent to the sound subsystem to play the missile sound. The general form of a message creation and configuration is as below:

```
SpecificMessage msg = new SpecificMessage();
```

```
msg.setField1(...);
//...
msg.setFieldN(...);
someEntity.send(msg);
```

There are tens of different message types and, obviously, they have different field names, and consequently, setter method names. When the message is sent, it is processed by the destination entity synchronously and the processing time varies with the type of the message.

One might want to write an aspect to parallelize the execution of such message creations and dispatch so that the main thread can continue without having to wait for the message to be processed[12]; however, the overhead of advice and thread creation might not be worth the parallelization of a single message creation and dispatch. But if there are cases in the code where a list of messages are dispatched together in a loop, then the parallelization might well be worth the effort.

The pattern of message creation and dispatch can be expressed using transcuts and a loop that contains instances of such transcuts can be identified using a *looped()* pseudo pointcut. Once designated, an *around()* advice can be written to intercept the whole loop and executes it on a newly created and launched thread:

```
void around(): messageLoop() {

  Thread t = new Thread() {

    public void run() {
      proceed();
    }
  };
  t.start();
}
```

3.4.5 Data and Control Context Designation

Transcuts can be used to define a control flow and dataflow context that can be used to more selectively target join points. For example, consider a resource that is created and used within a method. Resource objects usually have a well-defined usage pattern (interface protocol) that captures the correct way of using the resource. Transcuts can capture such patterns and be used to filter illegal or useless uses of the resource.

The example of Figure 3.32 shows how to skip any redundant calls to *close()*, and skip any method call on a resource before a call to *open()* and after a call to *close()*). That is, in the following example, method calls in lines 1 and 3 should be skipped:

[12]Of course, some messages might have to be processed before the main thread can continue.

```
resource.read(); //skip this call
resource.open(); resource.read(); resource.close();
resource.read(); //skip this call
```

```
aspect FilterRedundantCalls {
  transcut closeAndAfter(Resource f) {
    pointcut c1: call(* Resource.close()) && target(f);
    pointcut c2: call(* Resource.*()) && target(f);
  }
  pointcut anyUseAfterClose(Resource f, Resource f2):
                          call(* Resource.*())
                         && !call(* Resource.close())
                         && target(f)
                         && cflow(closeAndAfter(f2))
                         && if(f == f2);

  around(Resource f, Resource f2): anyUseAfterClose(f, f2) {}

  transcut openAndBefore(Resource f) {
    pointcut c1: call(* Resource.*()) && target(f);
    pointcut c2: call(* Resource.open()) && target(f);
  }
  pointcut anyUseBeforeOpen(Resource f, Resource f2):
                          call(* Resource.*())
                         && !call(* Resource.open())
                         && target(f)
                         && cflow(openAndBefore(f2))
                         && if(f == f2);

  around(Resource f, Resource f2): anyUseBeforeOpen(f, f2){}
}
```

Figure 3.32: Transcut for open/close usage

The same behaviour could be achieved in AspectJ by introducing two flags that track whether *open()* and *close()* have been called, and skip the useless calls accordingly. Transcuts should be able to do this more declaratively; however, as we discuss it in Section 3.6, the pinpointing mechanism needed for doing the above example more declaratively has not been implemented yet, therefore, we rely on traditional pointcuts to express the intent.

3.4.6 Static Verification of API Usage

The programming interfaces of application frameworks and software libraries do not usually enforce (statically) their usage protocol. For instance, a file system interface might provide methods for initialization and finalization of the underlying system but it cannot specify, in a checkable manner, that a programmer must not call any services before initialization and after finalization method calls. These design rules are usually communicated to the application programmers through code comments and API documentation. The systematic process of making sure that the use of an Application Programming Interface (API) complies with the designer's intent is referred to as API usage conformance and design intent verification. Many techniques have been developed to statically check such design rules and interface protocols [15], [30].

In AspectJ, the *declare error/warning* advice can be used to statically check some kinds of design rules. Figure 3.33 shows a very simple example of an aspect that can stop compilation with an error whenever the *new* operator is used to explicitly create an object in some class.

```
aspect EnforceFactory {
  pointcut newInSomeClass() : within(SomeClass) && call(*.new(..));

  declare error : newInSomeClass():
        "Must only use factory methods to instantiate objects!";
}
```

Figure 3.33: *declare error* advice in AspectJ

Transcuts can also be used as the target of *declare error/warning* and therefore can capture and verify some object protocols and design rules. For example, consider the file system initialization and finalization example and assume that the file system object is a singleton. Figure 3.34 shows an aspect that enforces the aforementioned interface protocol.

```
aspect EnforceProtocol {
  transcut useBeforeInit() {
    pointcut someservice: call(* FileSystem.*(..));
    pointcut init: call(* FileSystem.initialize(..));
  }
  transcut useAfterFini() {
    pointcut someservice: call(* FileSystem.finalize(..));
    pointcut init: call(* FileSystem.*(..));
  }
  declare warning: useBeforeInit(): "FileSystem should be initialized before use!";
  declare warning: useAfterFini(): "FileSystem cannot be used after finalization!";
}
```

Figure 3.34: Static API usage verification using transcuts.

Currently, transcuts that use dataflow relations between join points cannot be used in *declare* advice because the AspectJ language requires a static pointcut as the target of *declare* and our dataflow pointcuts are used both for context exposure (dynamic) and dataflow relations (static). In the future, we would like to allow the use of transcuts with dataflow pointcuts as the target of *declare* advice to enable powerful static checking for objects and resources. Consider a resource that should be used in a specific manner. For instance, a file object that needs to be first opened, then read, then closed. It is possible to inform the programmer at compile time if a file object is used after closing or before opening.

3.5 Implementation

We have implemented[13] transactional pointcuts as an extension to the AspectBench compiler (*abc*) [5]. The AspectBench compiler is an implementation of AspectJ developed to make it easier to add language extensions and optimizations. The back-end of *abc* is based on the Soot framework [53], which we used to implement region analysis.

The *abc* compiler supports two front-ends one of which is based on JastAdd [20], a compiler framework, which we chose to use for the transcuts front-end implementation. Jimple [54] is the underlying intermediate 3-address code used for code generation and weaving. We used AspectJ to implement some parts of the extension. The binary for the transcut extension and the examples presented in this thesis are available online for download.[14]

Jimple intermediate representation has been designed to simplify the implementation of analysis and optimization algorithms for Java. While many existing optimization and transformation algorithms work with 3-address code (in which an instruction operates on named operands), Java bytecode is stack-based and its instructions have implicit effects on the evaluation stack. The following is a piece of bytecode from [54]:

```
iload 1  // load variable 1, and push it onto the stack
iload 2  // load variable 22, and push it onto the stack
iadd      // pop two values, and push the sum of the two onto the stack
istore 1 // pop a value from the stack, and push it onto the stack
```

whose Jimple translation looks like:

```
int x, y, z;
...
z = x + y;
```

In Jimple, all operands have to be explicitly-typed local variables. The Jimple framework can translate Java bytecode to Jimple and vice versa.

We extended the *abc* compiler through its provided extension points. Whenever an invasive code modification in the *abc*'s code base seemed necessary, we used aspects instead to implicitly extend and adapt the behaviour. The use of aspects essentially helped to create more extension interfaces and allowed us to avoid modifying base code.[15]

[13]The implementation is not perfect; for instance, there is a fixable bug which causes changes to local variable within an around-advised join point to be lost; or many small enhancement that could improve the usability of transcut to a good extent but were not implemented due to time constraints.

[14]http://bit.ly/cSLKJN

[15]The use of aspects in extending *abc* was originally inspired by [25].

3.6 Limitations

The expressive power of transcuts can be improved by adding auxiliary pointcuts, operators, and modifiers to help designers better capture their intended join points. For example, the data flow relations are currently limited to must-alias relations; it would potentially be useful to enable may-alias relations and expose them through a proper interface. In addition, the pattern language can be enriched to allow better control over the matching algorithm. For instance, a *not(x)* pseudo pointcut can be implemented to control the non-contiguous matching behaviour in a way that it fails if it comes across a join point that matches x. Additionally, the lexical *withincode()* pointcut should be enabled to work with transcuts. It will be very effective in selecting join points based on their presence within a specific context.

Perhaps the most useful future enhancement is a direct pinpointing mechanism that allows targeting a segment of a transcut directly. Currently, one has to work around this limitation, as seen in some of the examples, which does not always work as desired.

3.6.1 Fragility

Pointcuts, in non-invasive pointcut-advice models, rely on the properties of join points to select them; this implicit dependency on join points implies that the evolution of the base code may break the existing pointcuts; that is, a simple change in the code can prevent a previously selected join point from being selected, and also, pointcuts may now pick out the join points that were not intended to be selected. This property of pointcuts is referred to as pointcut *fragility* and is inherent in pointcut-advice mechanisms as they implicitly rely on properties of join points.

Transcuts appear to be making fragility worse because of stronger reliance on structural and behavioural information of the target code. Transcuts should be carefully designed along with the target system to minimize such negative effects. At the same time, the designer should have a rich set of constructs to be able to clearly express her intent.

From another perspective, transcuts can help in some situations to remove dangerous dependency on method names. Traditional pointcuts can be broken by a simple name change of a method if they rely on the method's name. If instead of identifying the target method using its name, a transcut can be used to directly identify the operation of interest, then the method's name can freely be changed without breaking the transcut.

Whether the dependency on method name is worse or the dependency on the structure of the actual target operation is something that needs to be investigated; however, it seems that in the context of framework design where carefully-designed well-written code is frequently reused, transcuts can be used with the fairly safe assumption that the designated pieces of code will not change frequently. This assumption would probably not be a safe assumption in the context of application code written by inexperienced programmers.

3.6.2 Interaprocedural Limits, Escaping Objects, and Buried Effects

One important criticism against transcuts can be explained as follows. Consider a read operation on a file in the following form:

```
file.open();
data = file.read();
file.close();
```

where *file* is an object representing a file. This pattern of I/O is very common and a transcut can be composed to identify the set of such file operations by relating three key join points, *i.e. open(), read(),* and *close(),* all on the same target object, *i.e. file*; however, the identified pattern is not the only way a file operation can be coded. For instance, the same behaviour can be coded in the following form:

```
file.open();
data = read_and_close(file);
```

where *read_and_close()* is a procedure that reads some data from a file and then closes the file. This example is a case from a more general scenario in which the objects within a target join point escape the current matching scope and are used within a called method (*e.g. file* passed in to *read_and_close()*). Consequently, current matching algorithm cannot determine what methods are called on the escaped object. For instance, in the above example, the matching algorithm does not consider the body of *read_and_close()* when looking for a *read()* method call; therefore, no match is found.

This issue is closely related to the fragility problem and many cases can be avoided if a system is designed with aspects in mind. In cases where aspects need to be added to an existing design, one can still write a few transcuts to capture different code patterns that perform the same operation with respect to the key join points and the involved objects. This approach addresses almost all of this issue because a method name is an abstraction of the code within its body and therefore can be used in identification of the abstracted code.

There is another add-in feature that can help partially address the above limitation. An *effect()* pointcut can be designed that takes a set of desirable computational effects (*e.g.* method calls) and matches a join point if the execution of that join point emits that effect;

that is, a call join point can match *effect(read())* if *read()* is potentially called at some point within the execution of the method corresponding to the join point[16].

3.6.3 Continuation vs. History Semantics

One of the conceptual criticisms to our work is that we base our model on the dynamic pointcut-advice join point model while the matching algorithm uses static program representation. This is not an issue because *call()* pointcut in AspectJ is also a static pointcut matched based on the static representation of join points, still, it is defined within the dynamic join point model because it matches runtime join points (dynamic join points.)

In a similar fashion, transcuts are static in the sense that the matching algorithm uses static program representation to identify whether a section of code can generate the desired dynamic join points. What complicates the semantics of transcuts is the fact that, the dynamic join points that are identified as matches for a transcut are identified conservatively; that is, if the matching algorithm cannot predict that the join points generated by a piece of code match with the key pointcuts in a transcut, then that piece of code is not chosen as a match even if it actually generates the desired join points.

For instance, consider the following transcut:

```
transcut ab() {
  pointcut a: call(* a());
  pointcut b: call(* b());
}
```

In trace-based approaches, the semantics of such a sequence of pointcuts is clear because they are based on the history of runtime events; but in our model, in which predicting future events is necessary to allow *before()* and *around()* advice, the semantics is not as intuitive anymore. The reason is that it is not possible to determine whether a sequence (or any other pattern for that matter) of dynamic join points occurs before all of its join points are executed, at which point only a history of the target computation is at hand which can only be affected using *after()* advice.

Therefore, transcuts make available to the programmer what is practically possible and obviously cannot do any magic. In fact, in the thesis statement, we had to qualify the target "pieces of computation" with "well-formed" to capture this subtlety that a target piece of computation is matched if it can be determined statically that it will be a match at runtime. In a way, when a join point that can potentially be the beginning of a match for a transcut is activated at runtime, its (partial) continuation should match the rest of the transcut in order

[16]A simple semantic pointcut that matches join points based on their effect on heap is introduced in [9].

to have a complete match for the transcut. In other words, the matching algorithm does not wait until a history of execution is available at the end of computation, but acts on the available continuation of the beginning of candidate computation (join point).

Consequently, there will be pieces of computation at runtime that generate the same event (join point) pattern but could not be identified as matches before runtime. For example, in the code { *a(); b();* }, there is a match for the *ab()* transcut, whereas in the code { *a(); c();* } where *c()* is defined as { *b(); ...* }, there is no match because the current matching algorithm does not look into the body of the called methods when matching within another method. Even {*a(); if(...) then b();* } is not a match because whether *b()* is executed or not depends on the guarding condition[17].

It might be easier to think of transcuts as static pattern matchers that act on program text, which is not correct. Transcuts act on control flow and control dependence graphs and that separates them from language-level and text-level pattern matching and transformation techniques.

It is fair to say that, according to the above argument, the semantics of transcuts is confusing and counter-intuitive. That said, the usefulness of transcuts pushes us to find better ways to give meaning to them and think about them.

[17]This case can be captured using *dependent* pointcuts, which is not the point of this argument.

Chapter 4

The Semantics of Transcuts

In this chapter, it is shown that transcuts and the associated join point model are compatible and can be explained in the existing semantic accounts of dynamic joint points. The semantics framework that is used here is the Dutchyn's continuation semantics for dynamic joint points [18] explained in Section 4.1. It is shown, in Section 4.2, that this semantics can be extended to allow transcut matching.

4.1 Continuation-based Semantics for Dynamic Join Points

Dutchyn [18] gives a continuation-based semantics to dynamic join points, pointcut, and advice for a simple procedural language with first-order mutually recursive procedures and a top-level expression. Following Reynolds definitional interpreters [47] style of semantic specification, he defines an interpreter for the language and then augments it to support aspect-oriented constructs. The interpreter is implemented in continuation passing style (CPS) [4] to make the continuations explicit in the interpreter. Continuations in such an interpreter are regarded as "the rest of the computation" [14], which basically sequence the remaining steps of a computation. Continuations are usually represented as closures and, therefore, demand higher-order interpreter implementation; however, there are techniques that can be used to defunctionalize a continuation and avoid higher-order procedures. Dutchyn, specifically, linearizes the continuations and represents the entire continuation as a list (stack) of continuation frames.

Each continuation frame represents a single step in the computation and is represented as a data structure that contains the relevant fields for that computation. The top frame in the list is the immediate action that is taken if the continuation is activated (*i.e.*, provided a value to continue with.) For instance, a *CALL* continuation frame has a field that contains the procedure name to be called, and it expects (consumes) a list of values (evaluated arguments). An *EXEC* continuation frame keeps a list of values (evaluated arguments) and expects a procedure. *SET* and *GET* continuation frames represent setting and getting global variables, respectively. There are a few auxiliary continuation frames that are not essential but help in sequencing multiple argument evaluation and passing, as well as conditional constructs.

In the unmodified interpreter (non-AOP), the evaluation is an interplay between an *eval(expr, env, cont)* function and an *apply(cont, val)* function; the *eval()* function evaluates the trivial expressions and sequences the ones that are non-trivial (*e.g.* function application) by creating continuation frames and pushing them on the continuation frame stack. When a value is available (*i.e.* computed), the *apply()* function is called to send the value to the continuation: if the continuation stack is empty, the computation is halted; otherwise, the top-most frame is popped and processed. For instance, when expression "a()" is evaluated, the *eval()* function determines that this expression is a function application, therefore, it first evaluates its arguments with a continuation extended with a new CALL continuation frame (callF [id] ; CALL id :: !val). That is, after the arguments (if any) are evaluated, the values are sent to the awaiting continuation, which now has a CALL frame at the top. The CALL continuation frame contains the name of the function that is waiting for the values of its arguments. (The arguments are also evaluated using two auxiliary continuation frames: one is used to hold an expression and environment and to consume a list of evaluated arguments; the other is used to hold a list of evaluated values and consumes newly evaluated arguments to be added to its list. In the end of argument evaluation, a list of evaluated arguments are sent to the top-most continuation frame which is going to be a CALL.)

When the empty list (of arguments) is applied to the CALL continuation frame (which contains "a" as the name of the function), the procedure corresponding to the name is looked up and a new EXEC continuation frame is created (execF [args] ; EXEC val... :: !proc) and pushed on the continuation stack and at the same time *apply()* is called to send the looked-up procedure to the continuation (which has now an EXEC frame at the top.)

When a continuation that has an EXEC frame at the top is applied to a procedure, the procedure is examined and if it is a primitive procedure then it is invoked with the values held within the EXEC frame (the evaluated arguments) as well as the continuation (which does not contain the top EXEC anymore), as arguments. If the procedure is a user-defined procedure then the *eval()* function is called to evaluate the body of the procedure in an environment that binds its formal arguments to the values within the EXEC frame, and with the continuation from top of which EXEC is popped.

Dynamic **join points** are originally defined as the "principled points in the execution" [32]. The key idea behind the continuation-passing semantics is that by using continuations and specialized continuation frames, the meaningful states in the interpreter are exposed. Join points in this interpreter are the activations of non-auxiliary continuation frames in the CPS interpreter. In other words, join points are points in the execution (in the interpreter) where values are to be consumed by continuation frames. Therefore, there is four kinds of join points in the interpreter: CALL, EXEC, SET, and GET join points, corresponding to the state in the interpreter where a procedure call continuation frame is activated (and provided a list of values), the state where an execution frame is activated and provided with a procedure to evaluate, the state where a global variable is being set, and the state where the value of a global variable is being retrieved, respectively. For brevity, from now on, we focus on the most important continuation frames (join points), *i.e.* CALL and EXEC. The following notation is used in [18] to represent these two kinds of join points[1]:

- CALL (id $\vdash \neg$ val ...), which means that a call join point expects/consumes a list of values and carries an id (the name of the procedure.)
- EXEC (val... $\vdash \neg$ proc), which means that an execution join point expects/consumes a procedure and carries a list of values (the value of the parameters.)

Pointcuts are "means of identifying join points" [32]. In [18], they are predicates over the value provided to a continuation frame and the fields in the frame. A pointcut is a syntactic construct that examines the interpreter's state, that is, the activated continuation frame and the available value to the continuation, to determine whether it matches a given continuation frame with desired attributes. Therefore, a pointcut returns true if the current join point matches the given criteria.

For the above interpreter, there are two pointcut constructs, *CALLPC (pname, ids)*, and *EXECPC (pname, ids)* that can match two major kinds of join points: CALL and EXEC, respectively. The CALLPC pointcut returns true if the current continuation frame is a procedure call (CALL) that holds a procedure name equivalent to *pname*. Similarly, the EXECPC pointcut returns true if the top continuation frame is a procedure execution (EXEC) and that the supplied value is a procedure whose name is equivalent to *pname*.

A combinational pointcut is also defined, ORPC, that given two sub-pointcuts, matches the first pointcut and if that fails the second one is matched. If there is a match in any case

[1]Join points are the activation of continuation frames not the frames themselves; however, because a continuation frame represents a join point, they are used interchangeably in the text depending on context.

of the above pointcuts, the list of id's (*ids*) in the pointcut will be returned. If there is no match, false is returned. It is emphasized in this semantics that 0 matching does not alter the current continuation or values. Pointcuts identify dynamic join points and do not have computational effects; that is, they do not change the behaviour at those join points.

Advice is "a means of affecting semantics at those join points" [32]. In the continuation-passing interpreter, this is implemented as procedures that operate on continuation frames (in order to affect the rest of the computation.) Syntactically, advice has two parts: a pointcut and an advice body. The advice is semantically similar to a procedure that is invoked at the join points matched by the pointcut, and runs in an environment augmented with the matched pointcut's identifiers bound to the current join point's relevant values, and a special *proceed* identifier, denoting the original join point.

Consider the definition of a procedure, *pick* that takes a boolean argument and returns a integer (from [18]:)

```
(define (pick x) (if x 1 2))
```

Now, consider the expression

```
 (+ (pick #t) 3)
```

where the procedure *pick* is applied to value #t to have the result 1, and the overall result of the computation becomes 4. In fact, *pick* transforms the continuation of the procedure application from

```
(lambda (n)       ;await number
        (+ n 3))       ; add three; halt.
```

to

```
(lambda (b) ; await boolean
       (let ([n (if b 1 2)]) ;select number
       ((lambda (n) (+ n 3)) ; original continuation
        n))) ; given the selected number.
```

pick() extends the original continuation from $\neg$ *integer* (consumes integer) to $\neg$ *boolean* (consumes boolean). That is, *pick* can be viewed in two different modes: as a value transformer, it has the type *boolean* $\rightarrow$ *integer* and as a continuation transformer it has the type $\neg$ *integer* $\rightarrow$ $\neg$ *boolean*. Advice, in the continuation-passing interpreter, behaves like a procedure application to continuations; that is, it is applied to continuations and can extend or specialize them.

Here is a summary of the weaving steps in the interpreter. When a continuation frame is activated (*i.e.* a value is ready to be sent to it), a matching process collects all the advice whose pointcuts match the current continuation frame. If there is not such a match, the interpreter continues to run the operation corresponding to the current frame (*i.e.* the original join point.) If there is any match, the weaving procedure runs as follows. The advice body of the first match is evaluated in an environment extended with the list of id's from the matched pointcut, the *proceed* symbol, and the remaining matched advice.

The *proceed* is a closure that is created when a match is found. It contains everything needed to continue the original computation (*i.e.* the matched continuation frame and the value applied to it.) The remaining matched advice are also kept in a special environment variable. In the body of the advice, when a *proceed()* expression (if any) is evaluated, the list of remaining advice is retrieved from the environment and the next advice is evaluated. If there is no more advice to evaluate, a *proceed()* activates the original continuation frame potentially with values changed by advice.

In summary, in a defunctionalized continuation-passing interpreter, the activation of non-auxiliary continuation frames are considered as meaningful points in execution, or join points, that can be identified through predicates based on their type and run-time values. Special procedures (advice) can then be executed to specialize the behaviour of these continuation frames. The importance of this continuation-based semantics is that it does not rely on a vague or intuitive definition of join points; instead, join points arise naturally in the semantics of the programming language as the activation of continuation frames [19].

4.2 Extending the Continuation-based Semantics to Support Transcuts

In this section, the function of a CPS interpreter is explained through an example; then it is shown how weaving is added to the interpreter to enable join point matching and advice. Finally, transcut matching and advice is explained within the continuation-based semantic framework.

4.2.1 Defunctionalized CPS Interpreter: a Running Example

Consider the expression *e: b(a())*, in a simple procedural language. To compute the value of *b(a())*, first method *a()* is executed, then method *b()*, sending the returned value of *a()* to *b()* as an argument. To better achieve our goal in this section, we first step through the evaluation of this expression in the continuation-passing interpreter without weaving support. Then, it is shown how weaving is integrated for traditional join point matching and advice[2]. Finally, it is shown how transcuts fit in.

Assume that *e* is evaluated with a continuation k. As described in the previous section, in the defunctionalized CPS interpreter, evaluation is an interplay between an *eval (expr, env, cont)* function and an *apply (cont, val)* function. The *eval()* function evaluates an expression (*expr*) in an environment (*env*) and with a continuation (*cont*), which means when the evaluation of *expr* is finished and its result is available as a value, it should be sent to the awaiting continuation *cont*. The function that sends an available value (*val*) to a continuation (*cont*) is *apply()*. A continuation in this interpreter is represented by a list of continuation frames each of which represents a kind of operation that needs to be done on the available value. For instance, when an expression is a procedure application (method call), first, the arguments need to be evaluated, then the procedure is evaluated provided with the value of the arguments.

To demonstrate how the list of continuation frames changes in the course of evaluation, we use a list notation which extends from left to right, with k (the initial continuation awaiting the evaluation of *e*) represented as *(...)*. The evaluation starts with calling *eval ("b(a())", r, k)*, in which r is the current environment in which *e* is evaluated. When recognizing that the expression is a procedure application, the *eval()* function recursively and indirectly calls itself to evaluate the arguments of the application first in the same environment, r, but with the following extended continuation:

```
(..., callF [b])
```

callF [id] is a continuation frame that contains the name of a procedure; when activated, it looks up the procedure and consumes a list of evaluated arguments. The *eval()* function extends the current continuation (k) with a *callF* continuation frame before moving on to evaluate the arguments. In other words, when the evaluation of the arguments is finished,

[2]Note that we use Dutchyn [18]'s implementation to step through this evaluation. Details are omitted when appropriate for simplicity.

the list of evaluated values is sent to a continuation whose top frame is a *callF*. This is how the CPS interpreter sequences the steps in a computation.

To support multiple procedure arguments, two auxiliary continuation frames are used to evaluate the arguments one by one and add each evaluated value to a list. This is a recursive operation and *eval()* calls a recursive procedure to create the necessary continuation frames and add them to the continuation list. For each argument expression, a *randF [exp env]* is created that contains the argument expression (exp) and the current environment (env); then the procedure is repeated for the rest of the arguments with the continuation extended with the newly created *randF*. For the above example, the continuation list would look like the following after the procedure reaches the end of the list of argument:

```
(..., callF [b], randF ["a()", r])
```

Note that "a()" inside the *randF* is an expression and r is the environment in which *e* is evaluated. When there is no more arguments, the interpreter applies the empty list *'()* (which is a value) to the continuation by calling *apply (cont, '())*. This means that the top frame in the current continuation ($cont$) needs to be activated to consume the given value. In the example, the top frame is a *randF* containing the expression representing the first argument. The *apply()* procedure pops the top frame and recognizes its type and, similar to the *eval()* procedure, sequences the actions that need to be taken to process it, which in this case is to evaluate an argument expression and append it to the final list of arguments. It first extends the continuation with a frame *konsF [vals]*, which is an auxiliary frame that holds the evaluated list of arguments and consumes the next evaluated argument to be appended to its list of values, initialized with the value provided to randF (which is '() in the first invocation); then, the argument is recursively evaluated in the extended continuation, by calling[3]

```
eval (randF.expr, randF.env, extend(cont, konsF ['()]))
```

after which, the continuation looks like the following:

```
(..., callF [b], konsF ['()])
```

The argument is a procedure call itself, therefore a similar evaluation takes place that will once again sequence the operations needed to evaluate *a()*; that is, a new *callF* contin-

[3]Note that the pseudo code is intended to be simple to understand and it might not follow the syntax of any specific language.

uation frame is created and pushed into the current continuation. At this point, the continuation looks like the following:

```
(..., callF [b], konsF [’()], callF [a])
```

Similar to the evaluation of *b()*, the list of arguments need to be evaluated in the new extended continuation. When the list is (or becomes) empty, the *apply()* procedure is called to process the top-most continuation frame provided with an empty list (’()) as the value. In other words, the interpreter calls *apply(cont, ’()),* which in turn, activates (removes and processes) the *callF [a]* continuation frame; it creates and adds a new continuation frame of type *execF [args]* that contains the list of argument values and loads the procedure whose name is in the *callF* frame, in this case, *a*; then, once again, the *apply(cont, v)* is called where *cont* is the extended continuation (below) and v is the procedure corresponding to *a*.

```
(..., callF [b], konsF [’()], execF [’()])
```

The top frame in the continuation is an *execF*, therefore the call to *apply(cont, v)* will recursively evaluate the body of the procedure passed in v by calling

```
eval (v.body, r1, k1)
```

where *v.body* is the body of the procedure corresponding to *a*, $r1$ is a new environment in which the names of the parameters of the procedure are bound to the values embedded in the *execF [args]* (*args* is the list of argument values, empty list in this case), and $k1$ is the following:

```
(..., callF [b], konsF [’()])
```

The CPS interpreter relies on the continuation to decide what the next action should be after an expression is evaluated. The evaluation of the body of the procedure *a()*, in turn, creates the relevant continuation frames and recursive calls to *eval()* the result of which is a value available to the rest of the computation (*i.e.* the current continuation). For the sake of our example, let us assume that the result of evaluating *a()* is the integer 0. When the continuation is applied to the returned value, the top frame in the continuation is a *konsF* that contains a list of argument values. When the frame is activated, the list of arguments is extracted and the provided value (0 in this case) is appended to it; then, *apply (cont, args)* is called, where *args* is the list of arguments with one element (0) and *cont* is the following continuation:

```
(..., callF [b])
```

This call activates the *callF* continuation frame which, in turn, results in the creation of an *execF* frame and the evaluation of *b()*'s body. In the end, the result of the expression *b(a())* will be sent to the awaiting continuation k, *i.e. (...)*.

4.2.2 AOP-Enabled CPS Interpreter

Dynamic join points correspond to the activation of non-auxiliary continuation frames in the CPS interpreter; that is, when the interpreter calls the *apply(cont, v)* procedure, a join point is reached. Therefore, to enable join point matching and advice in the CPS interpreter, the *apply()* procedure is replaced by another procedure that, at each invocation, matches the pointcut of each advice against the current join point by calling a *match(pc, v, f)* procedure, in which pc is the pointcut of the current advice, v is the value sent to the continuation, and f is the continuation frame at the top of the continuation (*i.e.*, active frame.) f and v together form the current join point.

If the current pointcut-advice does not match, then the normal *apply()* procedure is called, as in the non-AOP interpreter; otherwise, the body of the matched advice is evaluated, similar to a procedure, with a difference: the environment is extended with these special identifiers: *proceed*, which points to a representation of the matched join point, and *advs*, which keeps the list of remaining pointcut-advice to be matched against the current join point. During the evaluation of the advice's body, if the *eval()* encounters any *proceed()* expression, it uses the environment to retrieve the list of the remaining advice as well as the original join point. If the list of the remaining advice becomes empty, then the evaluation of *proceed()* results in the original join point (continuation frame and value) being evaluated; otherwise, the next pointcut-advice is considered for matching.

The *match(pc, v, f)* procedure decides, based on the type of the pointcut (pc), whether the current join point matches or not. For instance, if pc is a *callC [pid ids]* pointcut, then the procedure first checks the type of the current join point and if it is a CALL join point; then, its embedded procedure name is matched against pid and if it matches then the procedure creates a match structure as the successful result. A match structure contains a list of the identifiers of the pointcut, a list of values bound to the identifiers, and a function abstraction that represents *proceed*. This function takes a parameter to allow passing changed values down to the original join point. For example, if a method call join point is designated and, in the corresponding advice, its proceed is called with different arguments than the original arguments, then the original join point should be activated and receive the new arguments; therefore, *proceed* is represented as a function that receives a parameter to allow this.

Consider the following pointcut-advice:

```
around(arg): call(a()) { proceed();}
```

This simply identifies join points of type CALL that contain “a” as the name of the called procedure. It binds the argument of any matched join point to v. The advice directly calls *proceed()* which results in the original CALL join point to be activated. Revisiting the steps of the evaluation of expression *b(a())*, there is a state in the interpreter when the empty list of arguments (’()) is applied to the following continuation:

```
(..., callF [b], konsF ['()], callF [a])
```

The AOP-enabled interpreter calls the *match(pc, v, f)* procedure where pc is *callC [”a” arg]*, v is the empty list ’(), and f is the active continuation frame, *callF [a]*. The type of the pointcut matches the type of the active frame (both are procedure calls) and the name of the called procedure in the active frame matches the desired name given in the pointcut (*i.e.*, “a”); therefore the match is successful and a match structure is created. The structure will look like

```
('(), '(), (lambda (nv) (values nv callF [a])))
```

As described before, the first field is the list of identifiers bound by the pointcut, the second field is the list of bound values, and the third field is a lambda abstraction that represents the original join point by packaging the active value and continuation frame. Note that the reason the nv is used instead of v is that, as explained earlier, when *proceed()* is called, whatever parameter value is passed to it is sent to the original join point; hence, nv is used in the proceed package[4].

After the interpreter finds a match and makes the match structure, it recursively evaluates the body of the corresponding advice ({ *proceed()*}, in this case) with a new environment that binds the special *proceed* and *advs* symbols to the packaged proceed in the match structure, and the list of remaining advice, respectively. Also, in the new environment, the identifiers in the match structure (first field) are bound to the values in the match (second field.) This evaluation is done with the continuation *(..., callF [b], konsF [’()])* (because the callF at the top is now processed.) If there were no calls to *proceed()* in the body of the

[4]Different kinds of join points need different packaging. For brevity, we avoid discussing all kinds of join points here. The CALL join point alone serves the purpose of the discussion.

advice, then the advice would be evaluated just like a procedure and the original join point would never be executed. In the above example, however, the *proceed()* is called with a new value for the argument. The interpreter retrieves the value of *proceed* in the environment which should, in this case, be a lambda abstraction that receives a parameter. The lambda abstraction is applied to the current value (*proceed()*'s argument) which results in the tuple *('() callF [a])*. The interpreter uses the value and the frame in this tuple to call the basic (non-AOP) *apply()* procedure to process the original call with the new argument.

4.2.3 Enabling Transcuts in CPS Interpreter

As described in the previous section, dynamic join points are the states in the defunctionalized continuation-passing style interpreter where a non-auxiliary continuation frame is applied to a value. The main observation that helps understand the meaning of transcuts (in the continuation-based semantic framework) is that the activation of a continuation frame is not usually isolated but related to other frames in the continuation waiting to be activated. In other words, the activation of a continuation frame can be considered in the context of (related to) the activation of other continuation frames.

Intuitively, various operations (continuation frames) within a computation (part of a continuation) relate to each other in the context of that computation; hence, the activation of such interrelated operations can be regarded as the activation of the computation itself. For instance, consider two different sequences of method calls *a(); b();* and *a(); c()*. In the traditional model, the join points corresponding to the activation of *a()* in these two sequences cannot be differentiated because join point matching and advice occurs at the level of individual continuation frames. In our model, not only do individual activations define join points (*i.e.*, can be identified and advised), but also the activation of a sequence of related continuation frames can, together, define a join point as well. That is, in the above example, the sequence of activations of *a(); b()* can be designated and advised as a join point, individuated from the sequence of *a();c()*.

Continuing with the example in the previous section, the following continuation is active when the arguments of *a()* are evaluated (empty in this case) and the procedure is to be called:

```
(..., callF [b], konsF ['()], callF [a])
```

At this state (join point) in the interpreter, when the *apply()* procedure is called to activate the top continuation frame, a matching procedure can select the join point (as previously explained) if there is any pointcut that designates a call to *a()*. Transcuts identify join points that are composed of a sequence of join points; the designated join points are activated when the first constituent join point in the sequence is activated. Intuitively, the remaining join points in such a sequence identify the elements of the remaining computation, or continuation. Consequently, to identify join points that match a transcut, the matching procedure should be extended to look at the whole (part of) continuation not just the top continuation frame.

Consider the following transcut that identifies states in the interpreter where a procedure *a()* is about to be called, with a continuation that contains a call to *b()*.

```
transcut ab() {

    call( a() ),
    call( b() );
}
```

The enhanced matching procedure would first match the head pointcut in the transcut against the top frame in the continuation. Upon successful match, the procedure moves to the next pointcut in the transcut and looks further down the list of continuation frames, and in this case, it matches a frame that represents a call to *b()*. There is no more pointcuts in the transcut to match, therefore the matching procedure returns successfully with a match, similar to the old matching procedure, with one major difference. In the old matching procedure only the top continuation frame would be considered for matching and, upon successful matching, would be removed and packaged in a *proceed* structure for later potential use; however, in the enhanced matching, the top sequence of frames in the list of continuation frames, starting with the current active frame down to the frame matched against the last pointcut in the target transcut would be removed and packaged for later retrieval (*i.e.*, if *proceed()* is ever called.)

Back to the example, after *callF [a]* and *callF [b]* are successfully matched against *call(a())* and *call(b())* pointcuts, respectively, the top section of the continuation starting from callF [a] and ending with callF [b] is chopped off and saved in a *proceed* structure. Then, the interpreter moves on to execute the body of the matched advice with the continuation *(...)* and in an environment that binds *proceed* to the packaged section of the continuation. If a *proceed()* is encountered in the body of the advice, the saved chunk of continuation is retrieved and activated, which, in turn, pushes the sequence of frames on top of the continuation and start activating them.

The above semantics leads to more powerful transcuts than what was presented in Chapter 3. Consider the above transcut for the following example.

```
1 c() {
2   d(a());
3 }
4
5 main() {
6   b(c());
7 }
```

When a callF [a] is activated the continuation looks as follows:

```
(..., callF[b], konsF['()], callF[d], konsF['()], callF[a])
```

The first pointcut matches *callF [a]* (corresponding to the call expression *a()* at line 2), and the last pointcut in the transcut matches *callF [b]* (corresponding to the call expression at line 6). Consequently, the top chunk of the continuation from *callF [a]* to *callF [b]* is removed from the continuation and put in a *proceed* structure. The intriguing result, in this example, is that the matched interval of continuation frames transcends procedure boundaries: *callF [a]* is in procedure *c()* and *callF [b]* is in procedure *main()*. This would not happen if the matching scope was limited to regions of control dependency in the control flow graph (which is the case in transcut realization for *AspectJ*). The main concern, then, is whether this leads to meaningless matching/advice. It appears that this concern originates form a tendency to think of weaving as a code transformation: how can one weave around advice for a join point whose one end is in one procedure and the other in another? This is a valid concern for static weaving (compile-time weaving), however, in a CPS interpreter, the rest of computation is represented and controlled explicitly and, hence, can be manipulated. In other words, it does not matter if the rest of the computation is the result of code in the same procedure or from multiple ones; what is known is that it is the "remaining computation" and it can be manipulated. Nevertheless, the matching/advice can always be restricted, or controlled, to prevent manipulations that are regarded as *radical* for a language.

One might think that transcuts are language-dependent concepts; however, this is not the case. The idea of transcuts is based on the principles of dynamic join points, and therefore, is independent of any particular programming language. Nevertheless, when it comes to implementation, the semantics of the target language as well as practicality dictates the extent to which dynamic join points, and transcuts for that matter, can be realized.

Chapter 5

Evaluation

We have chosen the cross-cutting concern of handling exceptions as a suitable modularization target for the purpose of evaluation of our new construct. The reason behind this choice is that separation of exception handling was the initial motivation behind this work; besides, it is easy to spot the implementation of this concern in a system and observe how it is scattered all over the system and tangled up with the code of other concerns.

Several researchers have investigated the possibility and potential impact of modularizing exception handing using aspect-oriented techniques, and have called for more powerful designation mechanisms to better achieve it (*e.g.* [36], [37], [8], [29], [10]).

Exception handling concern cross-cuts the implementation of the normal behaviour of a system. In Java, and many other languages, repetitive exception handling code is tangled with the code of other concerns. Consequently, its modularization can result in higher quality software. Additionally, there are other reasons why one might be interested in separating exception handling code into a module. Some studies show that the number of reactions to different exceptions is considerably lower than the number of places exceptions are caught, therefore there is opportunity for reuse of the handling patterns [36], [37].

Also, writing flawless exception-handling code is hard. As reported in [57], many programs fail to properly release acquired resources along all execution paths in the presence of run-time errors. Many programmers that are aware of exceptions and use proper constructs to catch and handle them, still write faulty exception-handling code. Writing correct exception-handling code becomes even more difficult when the number of resources that need to be handled increases (*e.g.* a database connection, a query statement, and a query result set are three resources typically involved in a database operation). Correctly dealing with N resources typically requires N nested try-finally statements or a number of run-time checks to track if resources are still allocated. These handling can result in badly-tangled code.

Still, whether the modularization of exception handling results in better code is not definite and is dependent on the side effects of the modularization as well as other design-specific factors. Castor *et al.* [8] have done an extensive study on the modularization of exception handling using aspects. They refactor some existing systems to modularize exception handling and compare them with the original version based on four software quality attributes, namely, separation of concerns, coupling, cohesion, and conciseness. Their general conclusion is that *"AOP does not fix poor designs"* [8]. In other words, in systems with poorly-structured or complex exception handling, there is little AOP can do to help the quality. They present scenarios where aspectization can be beneficial or harmful.

They also show that AOP improved the separation of concerns between exception handling code and normal application code. They also observe that even though aspects can help reuse exception handler code, it is sometimes difficult to achieve this kind of reuse without careful planning. The other relatively counterintuitive result of their study is that in systems with application-specific exception handling strategies, using aspects does not result in smaller number of lines of code. In terms of the measured quality attributes, they show that separation of concerns improved in the refactored systems; coupling was not affected much; cohesion was affected negatively, and size was not affected significantly.

As far as software quality is concerned, a careful design of the original system which is not oblivious to the potential use of aspects could significantly improve the results. Aspectization as above resulted in reduced cohesion in components, expressed in in terms of the number of method and advice pairs that do not access the same field. It turns out that the main reason of the poor results in cohesion is the large number of methods that were created to expose join points that AspectJ can target. In other words, their study confirms one of our observations that doing refactoring [22] to expose join points can be harmful and can compromise the original design. The authors reiterate this result a few times in their paper and point out the need for improvement in join point designation:

"... the increase caused by refactored operations, albeit small, is negative in most situations ... These new operations are not part of the system and possibly do not clearly state the intent of the developer. In some cases, the refactored operations comprises just a few lines that do not make sense when separated from their original context. This suggests that there is still room for improving AspectJ in order to more precisely select join points of interest" [8].

Other researchers have come to the same conclusion in their efforts in modularizing other cross-cutting concerns, not just exception handling. For instance, Zhang [58] has designed an AOP synchronization library called FlexSync to achieve customizability through decoupling synchronization intentions and mechanism; however, FlexSync requires refactoring to convert blocks into methods so that they can be picked out by pointcuts, which as explained, would negatively affect software quality.

5.0.4 Evaluation Strategy

To evaluate transcuts, it is necessary to realize what it is that transcuts enable developers to do. Transcuts remove the need for refactoring to expose join points (which is the major source of quality degradation when using aspects to modularize cross-cutting concerns); therefore, the right question to ask in the evaluation is "do transcuts significantly reduce the need for refactoring to expose join points in real existing software?"

To answer the above question, we considered two real-world software systems from two different domains. We modularized exception handling in these systems using aspects in the following fashion: for each exception handling block (identified by a *try* {}), we determined whether the piece of code within the *try* {} can be identified using a conventional AspectJ pointcut; if so, a pointcut was composed to select the target join point and an advice added to handle the exceptions of the target code. Basically, the handler blocks (*i.e.* *catch*{}, *finally*{} were moved to the corresponding advice. If the target piece of code could not be designated by a conventional pointcut, then a transcut would be composed for it (without having to do refactoring the target code into a method.)

```
...
try {
  x
}
catch(Exception e) {
  y
}
...
```

```
...
x
...
```

```
pointcut X(): select the desired x;
//or transcut ...

void around(): X() {
  try {
    proceed();
  }
  catch(Exception e) {
    //handle exception
  }
}
```

Figure 5.1: Original code (left), after removing exception handling code (middle), possible handler aspect (right).

Figure 5.1 shows abstractly what would happen to each piece of exception handling code. Note that in some cases, a combination of pointcuts and transcuts was needed to implement the desired handling behaviour.

We divided the exception handling cases in each system into four categories, as shown

Treatment (Technique)	Description
Conventional	*i.e.* pointcuts
Unreliably Conventional	With minor refactoring; or no refactoring, but minor change would require transcuts
Transcuts	Occasionally in concert with pointcuts
Complex cases	Not treated; usually need complicated workaround, careful refactoring; examples include unsupported join point boundary and return from handler

Table 5.1: Four different treatments of exception handling cases in the experiment.

in Table 5.1. The first category includes the cases that can be handled conventionally, that is, using AspectJ pointcut/advice. For example, if the code within a *try*{} block is a single method call, then it is selectable by a *call()* pointcut; so is the whole body of a method.

The second category includes the cases that can be handled using traditional pointcuts with the help of very minor refactorings (excluding extraction into method) or redesigns. Also, in this category are the cases that can be handled using pointcuts but could potentially break with minor future changes. Consider the following example:

```
1 try {
2   //may throw MyException, which is not an IOException (nor its subclass)
3   a();
4   b(); //may throw IOException
5 }
6 catch(IOException e) {
7 }
```

Only *b()* needs to be guarded for IOException because *a()* (line 3) cannot throw IOException. In case any other exception is thrown by *a()*, the next containing handler would catch it and that is not the concern of the shown handler. Therefore, one could write a pointcut to select *b()* (line 4) and handle the IOException in an advice, without changing the run-time behaviour of the above code. However, imagine a future change that requires MyException be handled at the same location as IOException: the pointcut has to be replaced by a transcut that can designate the whole composite join point beginning at *a()* and ending at *b()*. These cases form the grey area between pointcut-designatable and transcut-designatable[1] cases, to which we refer as "unreliably conventional".

The third category includes the cases that could only be handled using transcuts, possibly in concert with other AspectJ constructs. Without using transcuts, these cases would have required refactoring some target piece of code into a method to be selectable by conventional pointcuts.

[1]It should be noted that a transcut that only has a single constituent pointcut selects the same join points as the constituent pointcut.

Exception Handling Modularization: TSafe (classes: 123, LOC: 10556)	
Technique	# of cases (*try* blocks)
Conventional	20
Unreliably Conventional	6
Transcuts	41
Complex cases	9
Total	76

Table 5.2: Results of the exception handling modularization in TSAFE system.

Lastly, there were cases that we could not handle even using transcuts. For example, if the boundary of a region of code is formed by statements that are not join point in AspectJ, then neither pointcuts nor transcuts could select them. Examples of these statements are arithmetic operators and assignment into local variables. Also, as realized and reported in [8], there are complex cases of exception handling that are not worth the effort of separation. Examples include handlers that change local context variables, or return from the containing method, or affect the control flow of a containing loop. While transcuts could handle some of such cases, there were others that needed careful redesign/refactoring to allow reliable application of transcuts.

We should mention that in this experiment, the original code was oblivious to the whole aspectization, which explains some of the complex cases that had to be handled. The obliviousness factor in our experiment only shows how powerful transcuts can be; however, we do not believe that aspects can be used very reliably in a system without prior careful architectural and design consideration and provisions (Note that one can be “not oblivious” and “not intrusive” at the same time.)

In the rest of this section, we present the results of our experiment with the two systems and an example case of separation of exception handling form each.

5.0.5 Case Study 1: Tactical Separation Assisted Flight Environment

TSAFE (Tactical Separation Assisted Flight Environment) system is “a new tool to aid air-traffic controllers in detecting and resolving short-term conflicts between aircraft”[2]. Conceived at NASA and developed at MIT, it is usually used as a testbed for software verification experiments. It is a relatively high-quality piece of software with around 10,500 lines of code and 123 classes.

Table 5.2 shows the summary of cases we identified in this system. 41 out of total 76 exception handling cases were dealt with using transcuts. Without using transcuts, all of

[2]http://sdg.csail.mit.edu/tsafe/

these cases had to be refactored into new methods to be selectable by conventional pointcuts. That means 41 new methods would have had to be created most of which would not have had any meaning outside of their original context. As explained before, these would have been the source of a significant decrease in cohesion in the system.

There were cases where once single transcut was used to handle multiple cases of exception handling. Figure 5.2 shows the original code of a method responsible for loading data inside TSAFE. There are 6 *try-catch* blocks that are *abstractly* doing the same thing; that is, create a *FileReader* object for a file and reading data from the object (*e.g.* lines 8-9.) The first operation (creating a *FileReader* object) may throw a *FileNotFoundException*. The second operation may throw an IOException.

We composed a transcut to capture all 6 instances of these operations and an advice to handle the exceptions. Figure 5.3 shows the new method after removing exception handling code.

Figure 5.4 shows the corresponding exception handling aspect. Note that the handlers in the original code is accessing a local context variable (*errorMessages*); therefore, the aspect should capture a reference to the referenced object for use in the advice. Line 3 in the aspect code declares a reference which is set in the *after()* advice in lines 18-21. This advice is executed after successful execution of the *Vector* creation join point corresponding to line 3 in Figure 5.3, and saves a reference to the created list.

Transcut *readData()* is defined in lines 23-28 to select all instances of the described operations. Note that data flow pointcuts are used to establish a relation between two join points, which requires that the returned value of the first join point be the first argument of the second join point.

An *around()* advice is written to handle the exceptions thrown by the designated transcut. The original exception handlers add a different error message to the list of error in case of an exception. In order to write one single handler for all the join points selected by *readData()*, we stored the error messages in a static string array (lines 4-11) to be indexed using a variable (line 12) that is incremented after the execution of each join point (line 36.)

Note that this indexing is not reliable and could probably have been done better. These kinds of workarounds are reduced when the client code is not totally oblivious to the aspects. For instance, the static array of error messages could have been defined somewhere else to be accessible by anyone interested using a relevant key (*e.g.* the name of the source file that is being read.)

The next complication is that in the original code, if an exception occurs, it is handled

```java
List readStaticData(String[] dataFiles) {
  java.util.List errorMessages = new Vector();

  // Read fixes into database.
  try {
    //throws FileNotFoundException
    Reader reader = new FileReader(dataFiles[0]);
    //throws IOException
    this.readFixes(reader, tsafeDB);
  }
  catch (java.io.IOException e) {
    errorMessages.add("Unable to read fix file.");
    return errorMessages;
  }

  // Read airports into database.
  try {
    Reader reader1 = new FileReader(dataFiles[1]);
    this.readFixes(reader1, tsafeDB);
  }
  catch (java.io.IOException e) {
    errorMessages.add("Unable to read airport file.");
    return errorMessages;
  }

  // Read navaids into database.
  try {
    Reader reader2 = new FileReader(dataFiles[2]);
    this.readFixes(reader2, tsafeDB);
  }
  catch (java.io.IOException e) {
    errorMessages.add("Unable to read navaid file.");
    return errorMessages;
  }

  // Read airways into database.
  try {
    Reader reader3 = new FileReader(dataFiles[3]);
    this.readAirways(reader3, tsafeDB);
  }
  catch (java.io.IOException e) {
    errorMessages.add("Unable to read airway file.");
    return errorMessages;
  }

  // Read airways into database.
  try {
    Reader reader4 = new FileReader(dataFiles[4]);
    this.readSids(reader4, tsafeDB);
  }
  catch (java.io.IOException e) {
    errorMessages.add("Unable to read sid file.");
    return errorMessages;
  }

  // Read stars into database.
  try {
    Reader reader5 = new FileReader(dataFiles[5]);
    this.readStars(reader5, tsafeDB);
  }
  catch (java.io.IOException e) {
    errorMessages.add("Unable to read star file.");
    return errorMessages;
  }
  return errorMessages;
}
```

Figure 5.2: The original code of a method from TSAFE.

```
List readStaticData(String[] dataFiles) {

  java.util.List errorMessages = new Vector();

  // Read fixes into database.

  Reader reader = new FileReader(dataFiles[0]);
  this.readFixes(reader, tsafeDB);

  // Read airports into database.

  reader = new FileReader(dataFiles[1]);
  this.readFixes(reader1, tsafeDB);

  // Read navaids into database.

  reader = new FileReader(dataFiles[2]);
  this.readFixes(reader2, tsafeDB);

  // Read airways into database.

  reader = new FileReader(dataFiles[3]);
  this.readAirways(reader3, tsafeDB);

  // Read airways into database.

  reader = new FileReader(dataFiles[4]);
  this.readSids(reader4, tsafeDB);

  // Read stars into database.

  Reader reader5 = new FileReader(dataFiles[5]);
  this.readStars(reader5, tsafeDB);

  return errorMessages;
}
```

Figure 5.3: After removing exception handling code.

```
aspect ParserReadStaticDataAspect percflow(read()) {

  private java.util.List errorMessages = null;
  private final static String[] msgs =
                         {"Unable to read fix file.",
                          "Unable to read airport file.",
                          "Unable to read navaid file.",
                          "Unable to read airway file.",
                          "Unable to read sid file.",
                          "Unable to read star file."
                         };
  private int index = -1;
  private boolean error = false;

  declare soft: IOException:
          execution(List ParserInterface.readStaticData(..));

  after() returning(java.util.List errors): call(Vector.new())
            && withincode(List ParserInterface.readStaticData(..)) {
    errorMessages = errors;
  }

  transcut readData(Reader reader) {
    pointcut createReader: call(FileReader.new(..))
                      && return(reader);
    pointcut readit: call(* ParserInterface+.read*(..))
                      && args(reader, ..);
  }

  void around(Reader reader): readData(reader)
            && withincode(List ParserInterface.readStaticData(..)) {
    //skip all other reads in case of error
    //(this simulates "return" in catch.)
    if(error) return;

    index++;
    try {
      proceed(reader);
    }
    catch(IOException e) {
      error = true;
      errorMessages.add(msgs[index]);
}}}
```

Figure 5.4: Exception handling aspect for method in Figure 5.3.

Exception Handling Modularization: EIMP (classes: 120, LOC: 9000)	
Technique	# of cases (*try* blocks)
Conventional (Pointcuts)	34
Unreliably Conventional	20
Transcuts	22
Complex cases	0
Total	76

Table 5.3: Results of the exception handling modularization in EIMP system.

and then the handler returns from the containing method. When the exceptions are handled in an aspect, there is no way to return from the method containing the corresponding join point without using extra variables and logic; however, in this specific case, we worked around this limitation by using a boolean variable (line 13) that indicates whether an error has occurred. This error flag is set in the handler in the around advice (line 41) and is checked at the beginning of the advice so that the execution of the rest of the join points is skipped if the previous one ended with an error. This behaviour simulates the behaviour of the original code.

5.0.6 Case Study 2: Eclipse Instant Messenger Plugin

EIMP (Eclipse Instant Messenger Plugin) is an open-source Eclipse plugin that allows remote collaboration in a project by integrating popular instant messaging protocols[3]. This plugin has about 120 classes and 9000 lines of code.

Table 5.3 shows the summary of cases we identified in this plugin. 22 out of 76 cases of exception handling had to be modularized using transcuts which is still significant. Figure 5.5 shows the original code of a case that was handled using a transcut. In this case, the piece of code from line 7 to line 10 needs to be designated.

Figure 5.6 shows the code after removing the exception handling code and the corresponding aspect is depicted in Figure 5.7. Note that the transcut *execCmd()* (lines 5-10) does not establish any data flow relations because it was not necessary; in fact, even the two pointcuts at lines 7 and 8 are redundant because the desired join point can be designated by the first and last pointcuts (*i.e. getEnv* and *append*.)

As it is evident from this case, there is not always a decrease in the number of lines of code (LOC) when it comes to aspectizing exception handling. This observation has been reported in [8] as well. However, the changes in LOC could vary from system to system and can be improved by an aspect-aware design (as opposed to oblivious design.)

[3]http://eimp.sourceforge.net

```
public void run() {
  if (ss == null)
    return;

  StringBuffer buf=new StringBuffer();
  try {
    Process p = Runtime.getRuntime().exec(cmd);
    buf.append(getOut(p.getInputStream()));
    buf.append("\r\n");
    buf.append(getOut(p.getErrorStream()));
  }
  catch (Exception e) {
    buf.append("\r\n");
    buf.append(e.getMessage());
    StringWriter s = new StringWriter();
    PrintWriter p = new PrintWriter(s);
    e.printStackTrace(p);
    buf.append(s.getBuffer());
  }

  ss.sendMessage(new MimeMessage(buf.toString()));
}
```

Figure 5.5: A sample case in EIMP system before separating exception handling.

```
public void run() {
  if (ss == null)
    return;

  StringBuffer buf=new StringBuffer();

  Process p = Runtime.getRuntime().exec(cmd);
  buf.append(getOut(p.getInputStream()));
  buf.append("\r\n");
  buf.append(getOut(p.getErrorStream()));

  ss.sendMessage(new MimeMessage(buf.toString()));
}
```

Figure 5.6: Sample code after removing exception handling concern code.

```
privileged aspect ScriptEHAspect {

  declare soft: Exception: execution(* ImCommandServer.RespCmd.run());

  transcut execCmd(StringBuffer buf) {
    pointcut getEnv: call(* Runtime.getRuntime());
    pointcut start:  call(* Runtime.exec(..));
    pointcut getErr: call(* Process.getErrorStream());
    pointcut append: call(* StringBuffer.append(..)) && target(buf);
  }

  Object around(StringBuffer buf): execCmd(buf) {
    Object res = null;
    try {
      res = proceed(buf);
    }
    catch (Exception e) {
      buf.append("\r\n");
      buf.append(e.getMessage());
      StringWriter s = new StringWriter();
      PrintWriter p = new PrintWriter(s);
      e.printStackTrace(p);
      buf.append(s.getBuffer());
    }
    return res;
  }
}
```

Figure 5.7: Exception handling aspect corresponding to code in Figure 5.6.

5.0.7 Weaknesses of Transcuts

Our effort to use transcuts in modularizing exception handling in the EIMP and TSAFE systems better clarified some of the weaknesses of transcuts. Some of these weakness areas pertain to incomplete design and implementation while others are more intrinsic to the idea of transactional pointcuts. In this section, we show some of the cases from the TSAFE system where transcuts could not be applied, required auxiliary refactoring, or design-time a priori knowledge.

Consider the code depicted in Figure 5.8. This is a case from TSAFE that we categorized as “complex” because the desired target piece of code ends with an unsupported join point, in this case an arithmetic operator. The primitive join points supported by transcuts are inherited from AspectJ, therefore, this probably is not a transcut limitation per se; nevertheless, these cases make it harder for transcuts to do what they were originally designed to do.

Even though, in this case, *NumberFormatException* can only be thrown by the *parseInt()* method call, it is necessary to guard the whole region for the exception to achieve the correct behaviour; that is, line 6 should not execute if an exception is raised in the guarded block. There can be cases where this behaviour is not needed, and therefore, the

```java
try {
  String[] constraints = TSAFEProperties.getLatitudeConstraints();
  int minLatDegrees = Integer.parseInt(constraints[0]);
  ...

  maxLon = maxLonSign * (maxLonDegrees + (maxLonMinutes / 60.0));
}
catch (NumberFormatException e) {
}
```

Figure 5.8: Unsupported boundary join point

tight boundaries can be relaxed.

Another complex case that is a result of limitations inherited from AspectJ occurs as follows: inside an advice procedure one cannot "return" from the containing method. For instance, if an *around()* advice is executed when join point jp is activated within method *m()*, then one cannot return the control to the caller of method *m()* to simulate a "return" statement inside *m()*. This limitation implies that in Figure 5.9, we cannot straightforwardly extract exception handling behaviour because then the handling advice would have to return from the containing method (*getFixLatLon()*) if an exception is raised.

```java
Fix getFixLatLon(String fixDescription) {
  // Extract the latitude and longitude strings
  String latString = ... ;
  String lonString = ... ;
  ...

  // Get the latitude in decimal form
  try {
    lat = sign * getDecimalCoordinate(latString);
  }
  catch (NumberFormatException e) {
    return null;
  }
  ...

  // Get longitude in decimal form
  try {
    lon = sign * getDecimalCoordinate(lonString);
  }
  catch (NumberFormatException e) {
    return null;
  }
  //Return the parsed fix
  return new Fix(fixDescription, lat, lon);
}
```

Figure 5.9: Exception handling advice would need to "return" from the method containing the target join point.

Figure 5.10 shows another complex case that only looks similar to the previous case but is of a completely different nature. This is one of the situations where the lexical (text-based language-level) characteristic of exception handling constructs conflicts with the control-flow-based semantics of transcuts. As explained in Section 3.3, transcuts operate within regions of control dependency; that is, the matching begins at the beginning of regions and scans to the end while trying to match constituent pointcuts (some of which can be dependent pointcuts that take the matching into the nested regions.) This single entry single exit (SESE) property is essential in our join point model.

```
 1 try {
 2   // Extract fix from description
 3   ...
 4   relativeFix = getFixNamed(fixName, tsafeDB);
 5   if (relativeFix == null) {
 6     return null;
 7   }
 8   ...
 9   meters = METERS_PER_MILE * Integer.parseInt(distanceDesc);
10 }
11 catch (IndexOutOfBoundsException e) {
12   //handle the exception
13 }
14 catch (NumberFormatException e) {
15   //handle the exception
16 }
```

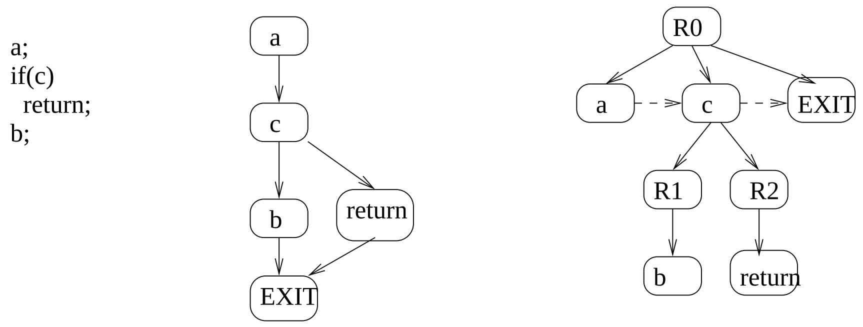
Figure 5.10: The complex case of “return” from region

If the *return* statement in line 6 did not exist, the control flow would merge back into the SESE region that is guarded by the *try* block. In other words, the desired piece of code could be identifiable by a transcut; however, the *return* statement is a (second) exit path from the otherwise SESE region, which in turn, makes all the following statements dependent on the containing conditional (line 5.)

Consequently, what is perceived as a (textual) region at the language level is not a candidate SESE region for our matching algorithm, but is instead composed of multiple regions that are separately considered for matching. Figure 5.11 shows a simplified version of this case and the corresponding CFG and PDG.

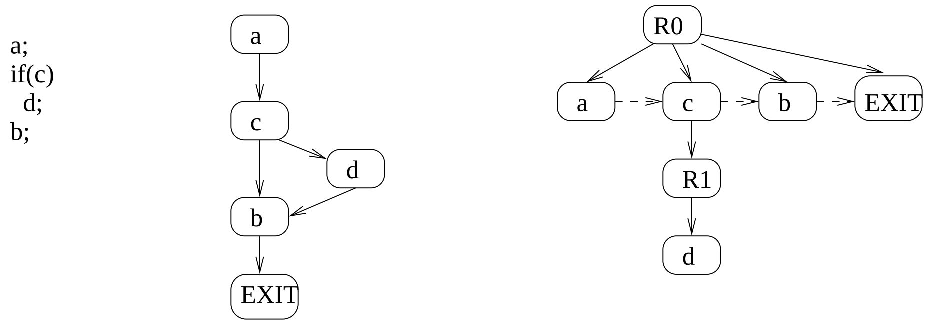
The *return* node causes the control flow to merge into the exit node of the method. This can be compared with the case in Figure 5.12 where *return* is replaced with a simple statement that flows the control into node b. This in turn makes b part of the same region as a and c, and therefore, would not have the same issue as the above case.

Although this behaviour is a limitation when refactoring existing exception handling code into aspects, it is the correct behaviour based on the definition of transcuts. The matching algorithm relies on the dependency information to choose the next potential match for its constituent pointcuts. If a node is not in the current dependence region where matching



```
a;
if(c)
  return;
b;
```

Figure 5.11: The complex case of “return” from region (left), its corresponding CFG (middle), and PDG (right)



```
a;
if(c)
  d;
b;
```

Figure 5.12: A simple case (left), its corresponding CFG (middle), and PDG (right)

takes place, then it should not be considered for matching because it may not execute at all[4]. This can also be considered a strong point of transcuts because it makes them independent of the language-level (source code level) constructs.

There are also other issues such as bugs and boundary cases that are typical in software systems and can be fixed given more time. For example, local variables within a designated region (join point) do not retain their changes if the join point is advised by an *around()* that calls *proceed()*. This is a bug and a result of the way around advice is implemented, and can be fixed by boxing the local variables before calling the advice in the compiler.

Also, the compiler should detect and report potentially erroneous transcut usage and help the programmer to understand and resolve the issue (Currently, most of such cases are detected through runtime exceptions and extensive debugging.)

[4]A *dependent()* pointcut can be used to capture such cases, however, when a *return* statement exists in the inner regions of the designated area, some special treatment is needed to produce correct behaviour for the *around()* advice.

5.0.8 Summary of Results

The above experiments showed that transcuts can in fact reduce the need for potentially harmful refactorings in cross-cutting concern modularization. The degree to which they can positively influence software quality can vary from system to system; if aspect-oriented techniques are chosen to modularize cross-cutting concerns, then transcuts can be a powerful tool in the toolbox. It is the designer of the system who should decide when the use of transcuts is worth the effort.

The aspect-oriented community recognizes that obliviousness of the base code to the (potential) presence of aspects can result in difficulty in designation and advice of desired join points (*e.g.* complex cases in our experiment), which would in turn lead to low quality and unreliable aspect code. Transcuts are no exception; however, transcuts can sometimes help reduce the negative effects of oblivious code by providing more designation power.

Chapter 6

Related Work

Region pointcut [2] is an independent work that addresses almost the same limitations as our work. A region pointcut can be used to specify a region in code that can be then advised. However, their technique and the concepts used are at the language-level whereas ours is at the level of control flow and the intermediate language. This choice of abstraction level has implications both in the semantics of region pointcuts and applications. Semantically, it is not clear what a region is and concepts like block and statements make it very dependent on the format of the written code. Also, existing applications in binary format cannot be targeted because the source code might not be available.

Region pointcuts do not have the data-flow relation semantics and constructs that transcuts provide. For example, you might designate the sequence *a(); b();* with a region pointcut but you cannot express the requirement that both *a()* and *b()* must be called on the same object. Also, transcuts can be embedded which allows modularization and reuse that is not supported in region pointcuts. Loops and conditional pointcuts within transcuts provides more control on what can be in a region.

Perhaps the most relevant previous work to this research is trace-based aspect mechanisms in which join points are run-time execution events and pointcuts consist of patterns of events (Tracematch [3], Declarative Event Patterns [55], and Trace-based Aspects [16]). Among these mechanisms, Tracematch seems to be the most developed and has been added as an extension to the *abc* compiler.

A tracematch is a module that includes three parts: an event definition part that defines the run-time events of interest using pointcuts; a regular pattern on the defined events; and advice to be executed when the pattern is matched at runtime. An automaton is used at runtime to recognize the specified event pattern. Tracematches use temporal relationships in join point selection and advice. Run-time events are monitored and once a specified pattern of events is matched, an advice can be executed.

There are some differences between trace-based mechanisms and our work. Trace-based mechanisms use an execution-time event-based join point model which naturally falls back on run-time event monitoring to determine when advice should apply. These models are inherently unable to take potential future events into account and can only express past and current events. Consequently, only *after*-typed advice is feasible. Transcuts, on the other hand, provide *around, before,* and *after* advice . Run-time performance will not be affected much in our model because transcuts do not rely on run-time event monitoring. Trace-based approaches, specifically tracematches, are on the path toward more context-aware join point matching; however, they are inherently limited in effectiveness and performance because of their event-based model.

Ptolemy [46] is a language designed to address some of the existing concerns (in aspect-oriented languages), such as dependency on syntactic structure and limited predefined types of *events* (join points). Ptolemy allows explicit declaration of event types that have a name and a set of variables used to expose context. Arbitrary sequences of expressions can be annotated as having a specific event type. Objects can register to be able to advise the event types they are interested in. When an event of a specific type is fired, all the relevant advice is executed. The designers of Ptolemy also recognize the need for arbitrary pieces of code to be treated as typed events (join points). In Ptolemy, the events have to be explicitly announced in the target code (*a.k.a.* base) while implicitness (non-invasiveness) is one of our design goals. That is, transcuts allow arbitrary event type declarations in an implicit manner. Also, the transcut designation and advice model complies with the existing dynamic pointcut-advice model which made it possible for integration and interaction with a language such as AspectJ. Similarly, in [29] Explicit Join Points (EJP) are used to announce join points in the base code explicitly, hence, their work differs from our approach in the same way Ptolemy does. Typed join point [52] is a similar work that reduces some of the explicitness in the explicit join point mechanisms but not in the case of arbitrary blocks of code.

A dataflow pointcut is introduced in [39] which allows selecting join points only if there is a dataflow relation between the current join point and a previous one. They introduce a *return* pointcut, similar to ours, that binds its variable to the return value of the target join point. Although the join point model they use is different than the join point model we introduce (*arbitrary computation cannot be designated as join points, which results in less effectiveness*), by harnessing the dataflow relation between join points, their work follows part of the same goals as our work. Similarly, extra control-flow pointcut operators are provided in [17] which make it possible to more selectively designate join points, however, because they rely on the traditional join point model, they inherit the aforementioned shortcomings.

A *loop()* pointcut is defined in [25] to designate and advise loops, which is in essence, a special case of transcuts since loops are composed of other join points. Transcuts generalize the notion of arbitrary computations as join points that are referenced to through their key constituent join points. Although loops are an instance of such computations, the *loop()* pointcut cannot select loops based on what they do, instead, all found loops are exposed.

LogicAJ [48], provides three basic pointcuts to match against three main elements in the target language structure, *i.e.* statements, expressions, and declarations. The use of logic meta-variables that bind to elements in code makes it possible to designate and interrelate various elements in the code, which makes it similar to transcuts that interrelate join points. The major differences between our work and LogicAJ is that transcuts match at the level of the control flow graph whereas LogicAJ works at the level of the language structure (code). Also, we define a join point model that emphasizes the notion of “arbitrary pieces of computation” whose shadows appear as regions in the CFG; the transcut is designed to realize this model. On the other hand, LogicAJ does not define a join point model or what the designated structures represent.

Program Query Language, PQL [38], is a query language that allows programmers to express design rules and find application errors. PQL queries are patterns that match on the execution trace; when a pattern matches, an action can be taken. The query pattern connects a set of primitive events (method call, field load/store, etc.) using sequencing, alternation, negation, and sub-query constructs. The query variables are objects in the target program and the query expresses a relationship between the objects. PQL is fundamentally different than our work in the same way as tracematches: queries are matched on the program execution events. PQL’s main goal is to detect program errors and design rules violations. The types of reactions to a match are limited (which is an inherent property of trace-based approaches). Also, PQL is a stand-alone language and not a language extension, therefore, its designers had more freedom in the design of its structure.

Java Tools Language, JTL [11], is another logic-based query language for Java. It has a very powerful pattern language which hides the underlying logic-based concepts from the programmer. Similar to other program query languages that work with the program static structure (code), JTL is fundamentally different from our work, which is based on the dynamic join point model.

Chapter 7

Conclusions

In this chapter, we review the contributions of this thesis and present some of the possible directions for future work.

7.1 Contributions

We presented a more liberal definition of join points in dynamic pointcut-advice model that allows join points that are composed of other join points in an arbitrary but controlled way, instead of limiting the kinds of identifiable join points to a primitive “well-defined” set. This new view of join points paves the way for new constructs that make designation and advice more flexible and powerful. Join points in this model can now be identified as part of a bigger context and in relation to other join points. Also, it is no longer necessary to create artificial method boundaries (refactoring) to expose join points.

We presented the transactional pointcuts, or transcuts, as a realization of this new model based on the AspectJ language. Program dependence graph (PDG) was used as the main program representation so that the control dependencies and region hierarchies are available to the new matching algorithm. Also, this representation makes binary code (byte code), in addition to source code, accessible for matching and weaving.

Example applications were presented to demonstrate how transcuts can work along with traditional aspect-oriented constructs to solve real problems. While some of theses application areas, such as exception handling, transaction management, and parallelization, are easily recognizable as potential targets for transcuts, there could be many other situations that transcuts can help. Similar to any other programming construct, the power of transcuts comes from abstraction and composition, which allows transcuts to be applied in new ways to solve new problems.

We showed what transcuts mean in the continuation-based semantics of dynamic join points. The continuation-based semantics of dynamic join points is a step forward in understanding the meaning of dynamic join points.

The effectiveness of transcuts was evaluated in modularizing the concern of exception handling in two real-world systems. The results show that transcuts can significantly improve the quality in aspect-oriented software systems, especially its cohesion.

One of the important design decisions in this work has been easy integration of transcuts with AspectJ and eventually adoption by programmers, which has affected the construct both syntactically and semantically. The transcut is a realization of the ideas presented in this thesis but not the only possible one. There is room for improvement in this realization in different directions, from structure to semantics, to matching algorithms, and so on.

7.2 Future Work

One possible future direction is to address the limitations in the design of transcuts, most of which are related to the expressiveness of transcuts (as discussed in Chapter 3.) It would be interesting to see how transcuts would perform when extended with more auxiliary pointcuts, operators, and modifiers to help designers better capture their intended join points. For example, the data flow relations are currently limited to must-alias relations; it would potentially be useful to enable may-alias relations and expose them through a proper interface. In addition, the pattern language can be enriched to allow better control over the matching algorithm. For instance, a *not(x)* pseudo pointcut can be implemented to control the non-contiguous matching behaviour in a way that it fails if it comes across a join point that matches x. Additionally, the lexical *withincode()* pointcut should be enabled to work with transcuts because it would be very effective in selecting join points based on their presence within a specific lexical context. Also, making it possible to advise constituent pointcuts within a transcut would significantly increase the power of transcuts and allow writing more succinct aspects.

Transcut construct is implemented in *AspectBench compiler (abc)* [5], which is an AspectJ compiler developed mainly to simplify language extensions and experiments. Although *abc* is a full AspectJ compiler, *ajc*[1] is the one mostly used in industry. Therefore, the next step toward possible adoption of the construct is to make it available experimentally within *ajc* so that people can easily try it and eventually use it to address their existing problems. With the heavy and frequent trial, new application areas would emerge as well

[1]http://www.eclipse.org/aspectj/

as new ways to improve the transcut construct.

Pointcut fragility is a well-known issue in AOP and transcuts appear to exacerbate it. Pointcuts in AspectJ rely on signature patterns to select join points, and therefore, even minor changes to a method/field signature (including name) can break the existing pointcuts. Transcuts are also composed of traditional pointcuts; consequently, they suffer from the same issue. Additionally, transcuts deal with not just signatures but code patterns, which in turn, adds to their fragility. It seems that interactive tool support for transcuts may help deal with the fragility to a good extent. Therefore, design, implementation, and integration of such tools is a fruitful future direction.

We evaluated transcuts only with respect to a single cross-cutting concern in real-world systems. It is necessary to evaluate transcuts with respect to other well-known cross-cutting concerns in real systems, which will help to understand their utility and limitations to a greater extent.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, second edition, 2006.

[2] Shumpei Akai and Shigeru Chiba. Extending AspectJ for separating regions. In *Generative Programming and Component Engineering (GPCE'09)*, pages 45–54, Denver, Colorado, October 2009. ACM.

[3] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. *SIGPLAN Not.*, 40(10):345–364, 2005.

[4] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[6] Thomas Ball. What's in a region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Lett. Program. Lang. Syst.*, 2(1-4):1–16, 1993.

[7] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[8] Fernando Castor, Nélio Cacho, Eduardo Figueiredo, Alessandro Garcia, Cecília M. F. Rubira, Jefferson Silva de Amorim, and Hítalo Oliveira da Silva. On the modularization and reuse of exception handling with aspects. *Softw. Pract. Exper.*, 39(17):1377–1417, 2009.

[9] Curtis Clifton, Gary Leavens, and James Noble. MAO: Ownership and effects for more effective reasoning about aspects. *ECOOP 2007 –Object-Oriented Programming*, pages 451–475, 2007.

[10] Roberta Coelho, Awais Rashid, Arndt von Staa, James Noble, Uirá Kulesza, and Carlos Lucena. A catalogue of bug patterns for exception handling in aspect-oriented programs. In *PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs*, pages 1–13, New York, NY, USA, 2008. ACM.

[11] Tal Cohen, Joseph Gil, and Itay Maman. JTL - the java tools language. In *OOPSLA'06 International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 89–108, 2006.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[13] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[14] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *The Fourth ACM SIGPLAN Continuations Workshop (CW'04)*, Birmingham B15 2TT, United Kingdom, 2004. School of Computer Science, University of Birmingham.

[15] Robert Deline and Manuel Fahndrich. Typestates for objects. In *18th ECOOP*, pages 465–490. Springer, 2004.

[16] Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. *Aspect-Oriented Software Development*, pages 201–217, 2005.

[17] Remi Douence and Luc Teboul. A pointcut language for control-flow. In *3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 95–114. Springer, 2004.

[18] Christopher Dutchyn. *Dynamic Join Points: Model and Interactions*. PhD thesis, University of British Columbia, November 2006.

[19] Christopher J. Dutchyn. Specializing continuations a model for dynamic join points. In *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, pages 45–57, New York, NY, USA, 2007. ACM.

[20] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

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

[22] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995.

[24] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, 2002.

[25] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM.

[26] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Not.*, 28(10):411–428, 1993.

[27] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. *SIGSOFT Softw. Eng. Notes*, 18(3):160–170, 1993.

[28] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM.

[29] Kevin Hoffman and Patrick Eugster. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 91–100, New York, NY, USA, 2008. ACM.

[30] Daqing Hou and H. James Hoover. Using scl to specify and check design intent in source code. *IEEE Trans. Softw. Eng.*, 32(6):404–423, 2006.

[31] G. Kiczales. What is a concern? Speaking at Working sessions on Comprehending Aspect-Oriented Programs: Challenges and Open Issues. co-located with International Conference of Software Comprehension ICPC'07, 2007.

[32] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, June 1997.

[33] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[34] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[35] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[36] Martin Lippert and Cristina V. Lopes. A study on exception detection and handling using aspect-oriented programming. Technical Report CSL-99-1, 1999.

[37] Cristina Lopes, Jim Hugunin, Mik Kersten, Martin Lippert, Erik Hilsdale, and Gregor Kiczales. Using aspectj for programming the detection and handling of exceptions.

[38] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. *SIGPLAN Not.*, 40(10):365–383, 2005.

[39] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *1st Asian Symposium on Programming Languages and Systems*, pages 105–121. Springer, 2003.

[40] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. *ECOOP 2003 – Object-Oriented Programming*, pages 219–233, 2003.

[41] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In G. T. Leavens and R. Cytron, editors, *FOAL'02 Foundations of Aspect-Oriented Languages Workshop at AOSD'02*, 2002.

[42] Doug Orleans and Karl Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag.

[43] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Architectures and Component Technology: The State-of-the-Art in Software Development*, January 2000.

[44] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[45] Andy Podgurski. Forward control dependence, chain equivalence, and their preservation by reordering transformations. Technical Report CES-91-18, Case Western Reserve University, August 1991.

[46] Hridesh Rajan and Gary Leavens. Ptolemy: A language with quantified, typed events. *ECOOP 2008 –Object-Oriented Programming*, pages 155–179, 2008.

[47] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363 – 397, December 1998.

[48] Tobias Rho, Günter Kniesel, and Malte Appeltauer. Fine-grained generic aspects. In *(FOAL'06) Foundations of Aspect-Oriented Languages*, 2006.

[49] Hossein Sadat-Mohtasham. Arbitrary non-contiguous pieces of computation: a new join point model for aspect-oriented programming. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 757–758, New York, NY, USA, 2008. ACM.

[50] Hossein Sadat-Mohtasham and H. James Hoover. Transactional pointcuts: Designation, reification, and advice of interrelated join points. In *Generative Programming and Component Engineering (GPCE'09)*, pages 35–44, Denver, Colorado, October 2009. ACM.

[51] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, 2006.

[52] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, 20(1):1–43, 2010.

[53] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[54] Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, 1998.

[55] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 159–169, New York, NY, USA, 2004. ACM.

[56] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[57] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM.

[58] Charles Zhang. Flexsync: An aspect-oriented approach to java synchronization. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 375–385, Washington, DC, USA, 2009. IEEE Computer Society.

Appendix A

Construction of the Program Dependence Graph in the Presence of non-Normative Control Flow

A Program Dependence Graph (PDG) is a digraph whose nodes are program units (e.g. instructions, or basic blocks) and whose edges denote dependencies between the nodes. The PDG representation has been shown to have a variety of applications in compiler optimization, slicing, and parallelization, as well as other more specialized areas (e.g. in Transcut matching in AOP.) The original paper that introduces PDG presents an algorithm to compute it, which not only is hard to comprehend but also ignores the exceptional control flow constructs. Other papers present faster algorithms but are either harder to implement or do not cover exception-handling constructs. Here, we report how we combined two algorithms to compute PDG and how we deal with exceptional control flow using our newly added control flow graph representation. Also, some parts of the implementation and some examples are explained. Java is our target language although most of the arguments and algorithms are applicable to other imperative languages. Also, we use Soot framework for its support for intermediate 3-address code (Jimple) and flow analysis utilities.

Program Dependence Graph (PDG) [21] is a program representation that makes explicit the control dependencies between the program elements. Here, we report the way we build PDGs for Java programs using the Soot [53] analysis and optimization framework.

PDG can represent both control and data dependencies; however, here we mainly focus on control dependencies because specific data dependencies are usually application-specific and can always be added on top the PDG. In the original paper, also, the control dependencies are the focus.

There are some algorithms and implementations (*e.g.* [12, 6]) that compute control dependencies and generate a control dependence graph; however, few of them generate the complete PDG, that is, one that contains regions of control dependencies. Others might need the source code to do the 0 at parse time (*e.g.* [27]).

Our current implementation is the outcome of an incremental development in the following way. Initially, we needed to compute the set of control dependence regions in a method[1]. We used the algorithm given in [6] to compute the regions because it was fast and intuitive. Subsequently, we realized that we needed the hierarchy of the regions (*i.e.* the dependencies among them as well as other nodes.) Therefore, we used part of the original algorithm given in [21] to construct the complete PDG given the set of regions and the control flow graph. We do not claim that our algorithm is better in any way (we have not investigated it), which is understandable because this algorithm is a by-product of our main research work.

We use Soot framework and, especially, the Jimple [54] intermediate language which allows our approach to work with Java bytecode as well as source code. Soot provides a few kinds of control flow graphs, which support both unit graphs (graphs whose nodes are Jimple instructions) and block graphs (graphs whose nodes are basic blocks); however, in order to deal with exceptional control flow in a way compatible with the rest of our application, we had to add a new kind of control flow graph on top of what Soot provides. This graph and the rationale behind it will be explained in this report.

The rest of this report is as follows. Section A.1 presents some of the background material on control flow graphs (CFG) and dominator trees. In Section A.2, after presenting the definition of dependence in a CFG, region analysis and PDG construction are explained; also some parts of the implementation and an example usage are presented. We explain some of the situations that make a CFG (and in turn the corresponding PDG) complicated and our approach to addressing them in Section A.3. Finally, we conclude in Section A.4.

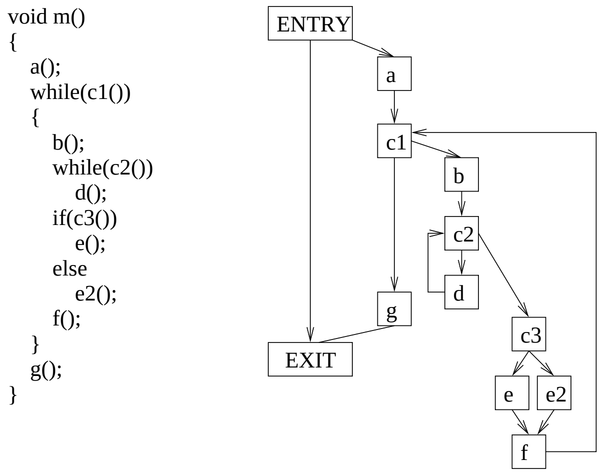
A.1 Background

A.1.1 Control Flow Graph

A control flow graph (CFG) is a directional graph in which nodes represent basic blocks (or instructions) and edges indicate flow of execution from a source node to a target node (see [1] for the precise definition and algorithms for computing CFG.) Figure A.1 shows a program and its CFG. The *START/EXIT* nodes are traditionally added to a CFG to simplify

[1]This was needed in the matching phase of Transcut [50] which is an AOP construct.

the implementation of graph algorithms (for instance, by making the graph single-headed). *START* represents whatever condition that leads to the execution of the method body, and is, in our implementation, a dummy *nop* instruction (so is an EXIT node). Nevertheless, we do not add them unless necessary.



```
void m()
{
   a();
   while(c1())
   {
      b();
      while(c2())
         d();
      if(c3())
         e();
      else
         e2();
      f();
   }
   g();
}
```

Figure A.1: A program and its CFG

Soot provides interfaces and implementation for a few kinds of CFGs. *UnitGraph* and *BlockGraph* represent CFGs whose nodes are *Unit*s and Blocks[2], respectively. *BriefUnitGraph* (*BriefBlockGraph*) and *ExceptionalUnitGraph* (*ExceptionalBlockGraph*) extend *UnitGraph* (*BlockGraph*). An "exceptional" graph includes potential exceptional flow in program whereas a "brief" graph only includes normal flow edges between the nodes. Therefore, depending on the application at hand, the appropriate implementation should be used.

In our case, neither brief nor exceptional version was adequate. In a brief graph, each catch or finally block starts a head in the graph. A head in a CFG is a node that does not have a predecessor. So, such CFG can potentially be multi-headed, which makes the dominator tree a forest, which in turn leads to problems in computing the program dependence graph. The multi-headedness can be solved by adding a *START* node to the CFG (as done traditionally), and an edge from the *START* to all heads in the graph; however, the graph would still be incomplete because the hierarchy information of the exceptional blocks would be lost and the dependencies would be inaccurate.

Exceptional graph, on the other hand, includes exceptional flow, and consequently, the multi-headedness would not be an issue because there would be an edge from potentially

[2]In Soot, *Unit* is an interface representing a program instruction, and *Block* is a basic block.

exceptional nodes to the catch/finally blocks; however, every node can potentially throw an exception which results in graph that has an extra exceptional edge from almost every node to the visible catch blocks. Again, for some cases this CFG could be appropriate but in our application this representation was not precise enough; therefore, we implemented our own CFG.

We implemented the *EnhancedUnitGraph* (*EnhancedBlockGraph*) which is a *UnitGraph* (*BlockGraph*) very similar to a *BriefUnitGraph* (*BriefBlockGraph*) with a proper representation of exception handling behaviour[3]. the details of how the exceptional flow is represented in the graph and the observation behind it are explained in Section A.3.2.

A.1.2 Dominator and Post-Dominator Trees

Node x dominates y (x is a dominator of y) in a CFG if every path from the START node to y goes through x. Similarly, x post-dominates y (x is a post-dominator of y) in a CFG if every path from y to the EXIT node passes through x. A dominator (post-dominator) tree is used to represent the dominance (post-dominance) relation in a CFG. Each node in a dominator (post-dominator) tree dominates (post-dominates) only its descendants in the tree. Figure A.2 shows the dominator and post-dominator trees for the program in Figure A.1 (ignore the dotted selection for now.)

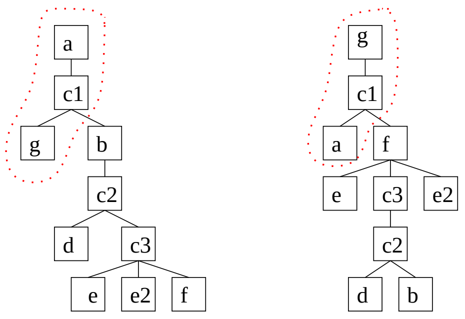


Figure A.2: Dominator (left) and post-dominator (right) trees

Dominator and post-dominator trees are used in various flow analysis algorithms; here, we used them for computing the regions of control dependence as well as the program dependence graph itself.

[3]In the absence of exception handling constructs (try-catch-finally), enhanced graph, brief graph, and exceptional graph are all the same.

A.2 Program Dependence Graph

Program Dependence Graph (PDG) [21] is a program representation in which the control dependencies between nodes in a CFG are made explicit and also region nodes are added to represent the common set of dependencies for a set of nodes. PDG has successfully been used for program optimization, parallelization, slicing, automatic testing, *etc.* all of which require that dependencies among program statements be easily accessible. Nodes in PDG are the same as the nodes in Control Flow Graph, though in our implementation, we use a wrapper class to carry some information about nodes. Edges in PDG denote control dependency between two nodes. Control flow information is implicitly available in PDG through ordering, but it can also be explicitly represented through control flow edges. In this section, we illustrate a PDG of a simple program and explain how it is constructed.

Figure A.3 shows the PDG corresponding to the program in previous section. Only control flow dependencies are shown in the PDG because we currently do not use explicit data flow edges in PDG.

Informally, node B is control dependent on node A if the execution of A determines whether B executes or not. The formal definition from [21] is as follows. Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

1. there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and

2. X is not post-dominated by Y.

For instance, *b()* method call is dependent on *c1()* because there is a path from *c1()* to *b()* that only contains *b()* and *c1()* (so the first condition holds because there is no Z in the path), and *c1()* is not post-dominated by *b()* (see Figure A.2). Intuitively, *b()* is control dependent on *c1()* because the execution of *b()* depends on the result of the execution of *c1()*. But *c1()* is not control dependent on *a()* because *a()* is post-dominated by *c1()*. Intuitively, *c1()*'s execution is not dependent on *a()*'s execution.

There are generally two kinds of nodes in a PDG: CFG nodes (*i.e.* basic blocks or instructions) and region nodes; and there is an edge from A to B if B is control-dependent on A. A region node summarizes and factors out the set of control dependences of a set of nodes in a PDG. For instance, all the nodes within the body of the top-level loop in Figure A.1 are control dependent on *c1()*; so, a region node can represent this shared dependence set: region node *R2* in Figure A.3 is created and made control dependent on *c1()*

and all the nodes in the body of the loop are made control-dependent on *R2*. This dependence set summarization is performed for all control dependences and the created region nodes are added to the PDG. At each level of the PDG, nodes are (either implicitly or explicitly) connected by control-flow edges. Throughout this thesis, the reader can assume that the control flow in each region in PDG (*i.e.* the child nodes of the region) is from left to right, unless the flow is explicitly shown in the graph. PDG makes the region hierarchy explicit and available for matching. For instance, inner loops can easily be identified (*e.g.* region *R3* is the inner loop in region *R1*).

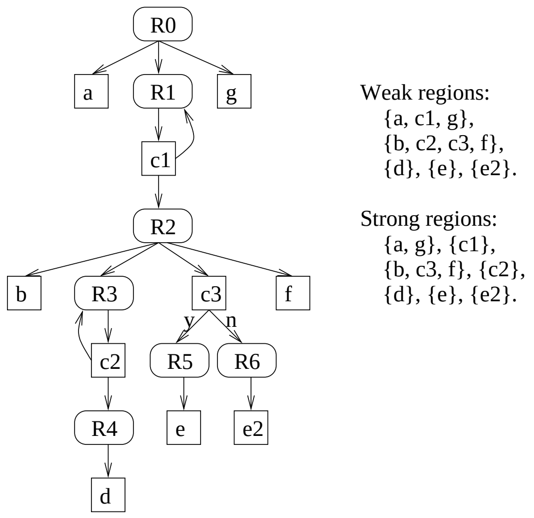


Figure A.3: PDG of the program in Figure A.1

Various algorithms have been proposed to efficiently compute the PDG[4] of a program (*e.g.* [21], [12], [27], [6]). We reuse the region analysis machinery that we initially implemented to build weak regions (based on the algorithm in [6]) and construct the PDG based on the algorithm given in [21].

A.2.1 Region Analysis

In any execution path from the beginning of the flow graph to the end, either all the nodes in a region execute or none of them do. Regions are, therefore, the natural extension of basic blocks (the control enters through the header of a block and STOPs through the end). Regions can be *weak* or *strong*. We use these definitions from [45], which result in slightly different regions than the ones defined in [1].

- **Weak Region**: vertices v and w are in the same weak region iff for any complete

[4]Some only compute the Control Dependence Graph or the set of regions.

control-flow path, v and w are both in the path or are both absent from the path.

- **Strong Region**: vertices v and w are in the same strong region iff v and w occur the same number of times in any complete control-flow path.

We used an algorithm presented in [6] which finds weak regions based on the observation that

- v and w are in the same weak region iff (v dominates w and w post-dominates v) or (w dominates v and v post-dominates w).

Strong regions would be the same as weak regions if there were no loops in the CFG. Distinct vertices v and w are in the same strong region iff

- they are in the same weak region **and**

- (v is in every cycle containing w) and (w is in every cycle containing v.)

The weak and strong regions of the CFG in Figure A.1 are shown in Figure A.3. The regions in the PDG correspond to the strong regions. The linear algorithm given in [6] is based on the key observation that, for any CFG, the vertices of each weak region form a chain in the post-dominator tree that is the reverse of a chain in the dominator tree (see Figure A.2 for the chains corresponding to the weak region {*a, c1, g*}).

A.2.2 Constructing the Region Hierarchy

As mentioned before, nodes in a PDG are either region nodes or CFG nodes (*i.e.* basic blocks.) A region node in the PDG represents a strong region that has dependency edges to its immediate dependent nodes. In the implementation, there are slight differences between weak regions, strong regions, and regions in the PDG, but conceptually, they are all the same: they represent a region of control dependence in a control flow graph. Consequently, in the code, *IRegion* is an interface and *Region* and *PDGRegion* are weak/strong regions and PDG region nodes, respectively. Design-wise, the latter two could probably have been consolidated but we had to keep the back-compatibility with our past implementation; hence, the current design.

After computing the weak regions, we compute the strong regions and construct the PDG at the same time. The outline of the algorithm is as follows. Given the list of weak regions, PDG can be constructed by finding the inter-region dependencies. Starting from the top-level weak region, a top-level PDG node, R, is created and for each (CFG) node A

in the region, first, a PDG node is created to represent A in the PDG, then, a dependency edge is added from R to the PDG node representing A.

Then the set of nodes that are dependent on A are found: for each edge (A,B) in the CFG such that B does not post-dominate A, let L be the least common ancestor of A and B in the post-dominator tree. Either L is A or L is the parent of A in the post-dominator tree (see [21] for proof). If L is the parent of A, then all nodes in the post-dominator tree on the path from L to B, including B but not L, are control dependent on A. If L is A, then all nodes in the post-dominator tree on the path from A to B, including A and B, are control dependent on A (this case captures loop dependence.) Both cases can be covered by traversing backwards from B in the post-dominator tree until we reach A's parent (if it exists, or A otherwise) and adding all visited nodes to a list as nodes that re control dependent on A.

The A's PDG node is changed to be a "Conditional" PDG node to represent the fact that there are nodes that depend on it; then, for each of the dependants of A, the containing (weak) region is looked up[5] and a PDG node is created to represent it; a dependency edge is then added from the A's PDG node to the region's PDG node. This is repeated for all the nodes in the list of dependants that are in a different weak region than the previously processed dependants of A. Loops that contain abrupt exit or continuation statements cause some conditions that need to be checked in the above steps. When it turns out that a loop header is being processed, a new strong region is created along with its corresponding PDG node which is added to the graph and the appropriate dependency edges are added. It is worthwhile to mention that loops create circular dependencies in the PDG and we label the back dependency edges in the PDG as "dependency-back" so that clients be aware of them.

A.2.3 Implementation and Usage

Our PDG is a graph that extends the *HashMutableEdgeLabelledDirectedGraph* class in Soot, which is a directed graph whose edges are 0, and implements the expected interface of a program dependence graph, *i.e., ProgramDependenceGraph*. The PDG classes can be used in any project that is properly set up to use Soot, and that imports the PDG package. Figure A.4 shows how this interface can be used:

The list of the PDG regions is created by doing a post-order traversal of the PDG and adding the PDGNodes with the same dependency (that is, the same parent), to the region. The reason we call them "PDG regions" is to avoid confusing them with strong and weak

[5]This information is available from the region analysis phase.

```

/* Body body = ...

    body represents the body of any method
    whose PDG is desired. It may be acquired through
    different ways (e.g. internalTransform method, etc.).
*/

//Create the CFG of the method
EnhancedUnitGraph cfg = new EnhancedUnitGraph(body);

//Create the PDG for the given CFG
ProgramDependenceGraph pdg = new HashMutablePDG(cfg);

//Print a textual representation of the graph.
System.out.println(pdg);

//Get a list of the regions in the PDG
List<PDGRegion> pdgRegions = ((HashMutablePDG2)pdg).getPDGRegions();

//The all the 1 of the top-level region
PDGNode head = pdg.GetStartNode();
List<PDGNode> deps = pdg.getDependents(head);
```

Figure A.4: An example of construction and usage of a PDG

regions, even though they are almost the same (The type of nodes are different because they belong to different program representations).

Figure A.5 shows part of the class diagram of the PDG package. A *PDGNode* represents a node in the PDG, which can be either a region node or a CFG node. A *LoopedPDGNode* is a *PDGNode* that represents a loop and has a header node and a body node, both of which are of type *PDGNode*. Similarly, *ConditionalPDGNode* represents a node in the PDG that has two 1, first of which runs when the corresponding condition is true, and second of which runs when the condition is false[6]. Many classes and relationships are not shown for brevity.

Edges in our PDG implementation are either “dependency”, “dependency-back” (representing loop dependency), or “controlflow”.

A.3 Non-Normative Control Flow

Non-normative flow occurs when an instruction alters the flow of control in the code in such a way that it changes the normal, expected, frequent flow of the code. It should be noted that this is not a precise definition in the sense that an alteration of flow might be completely normal and still (in our definition) a non-normative. Fortunately, there are only a few known instructions that cause such behaviour: *break* and *continue* statements in loops, *throw*, and

[6]In our implementation, we do not explicitly specify which dependent is associated with which the condition being true but this information can be added based on the predefined Jimple code generation rules in Soot.

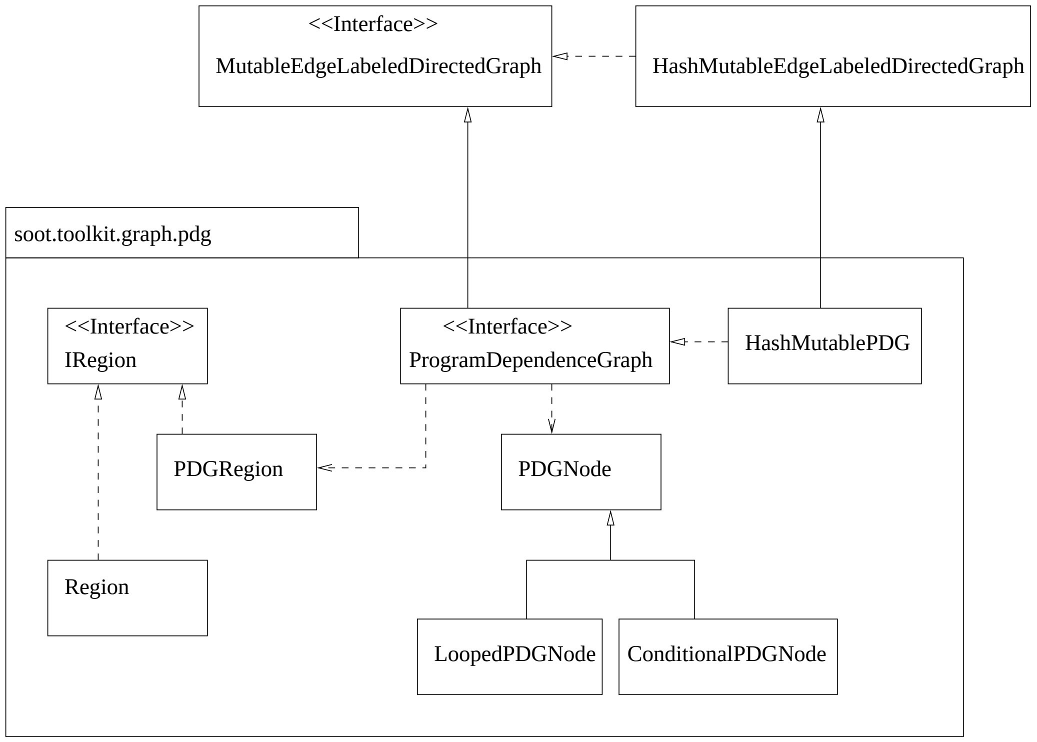


Figure A.5: UML class diagram for part of the PDG package

return (in the middle of methods.) In the following sections, we explain how these affect the CFG and how we deal with them in the enhanced CFG.

A.3.1 Abrupt Loop Exit and Continuation

The presence of *break* and *continue* statements in loops affects the PDG in tricky ways. In this section, we present an example to show how the PDG of a method with a loop that contains another loop looks like. The matters get worse when the target of a *break/continue* statement is an outer loop. Consider the method shown in Figure A.6, along its CFG (it does not do anything meaningful, so, do not try to understand the function but its structure).

Figures A.7 and A.8 show the dominator/post-dominator trees and the corresponding PDG for the method, respectively.[7]

A.3.2 Exceptional Flow

Figure A.9 shows two different kinds of flow out of a node (unit or block) in a CFG (if the node is a return or throw statement, then there is no flow out of it.) In fact, X represents

[7]In our implementation, we do not add ENTRY/EXIT (START/STOP) nodes to the CFG if not necessary. In fact, we know what nodes in the CFG are heads and tails; If the head and tail nodes are unique, they can play the role of ENTRY/EXIT nodes. Otherwise, we add the ENTRY/EXIT nodes.

```
public int f() {
    int i = 0;
    int j = 0;

outer:
    while(j < 10) {
        i = j + 2;

        while(condition2()) {
            i+=2;

            if(i < 3)
                continue;

            if(i > 4)
                break outer;

            i −=1;
        }

        if(i == j)
            i −= 1;

        if(i == 3)
            continue;

        if(i == 4)
            break;

        j++;
    }
   return i;
}
```

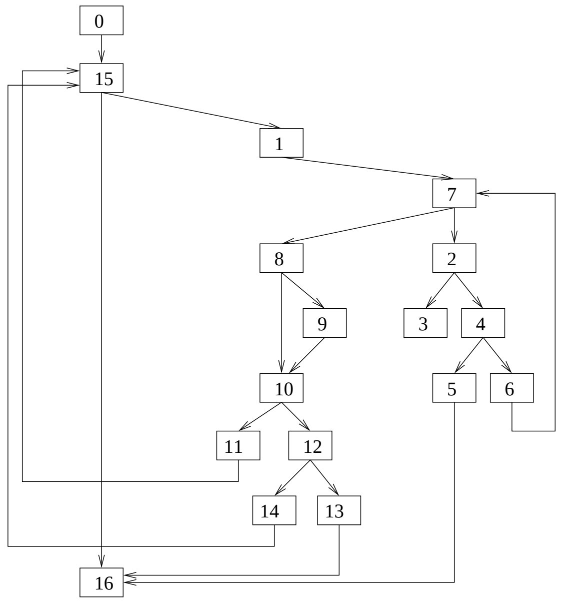


Figure A.6: A program and its CFG

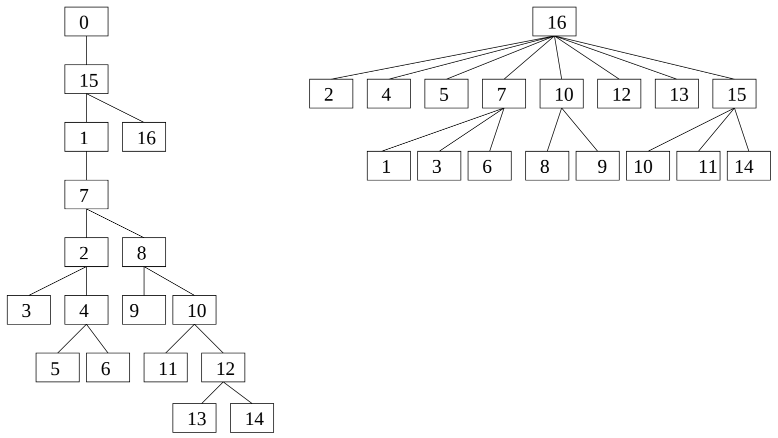


Figure A.7: Dominator (left) and Post-dominator (right) trees

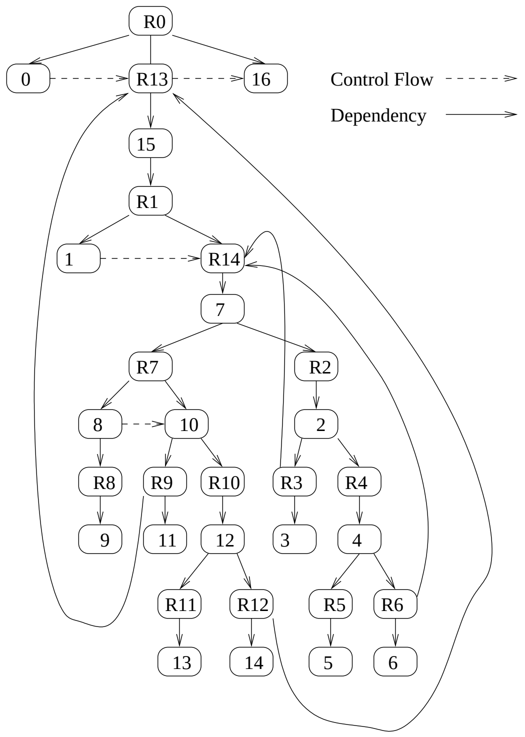


Figure A.8: The PDG of program in Figure A.6

any well-structured portion of the program: a single unit, a basic block, a whole loop, or a try-catch-block; the important property that all of them have is that the (normal) flow of execution enters at the beginning and exits at the end, no matter how complex the flow is within them.

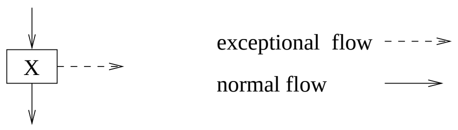
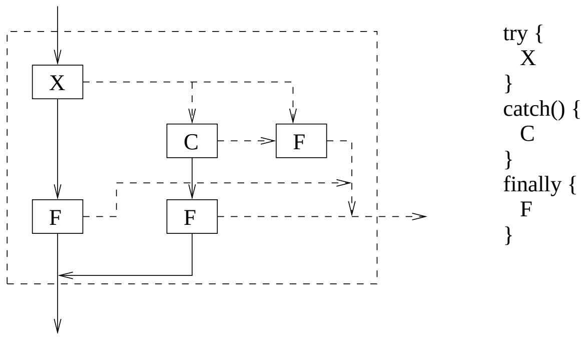


Figure A.9: Control flow out of a node

If X is not guarded by a try block, the exceptional flow causes the method to exit (exceptionally.) It is the presence of exception-handling constructs, however, that necessitates our introduction of the *EnhancedUnitGraph*. Figure A.10 shows the flow of control when X is guarded by a try-catch-finally (X is defined recursively.) X represents any well-structured segment of code and C and F represent catch and finally blocks, respectively. Also, one thing that should be mentioned is that, in Soot, the finally block is triplicated: for exceptional flow, for normal flow when no exception is raised, and for normal flow when an exception is raised but handled. It appears that there is no need to duplicate the finally code for the normal flow cases, however, we wanted to be consistent with the way Soot generates Jimple code.



```
try {
  X
}
catch() {
  C
}
finally {
  F
}
```

Figure A.10: Control flow inside a try-catch-finally

The important observation is that, well-structured segments of code, no matter how complex they are, can be abstracted and treated similar to X in the above figures; that is, either the execution reaches the end of the segment normally, or an exception occurs at some point in the execution. In our use cases, it does not matter at what specific point in X did the exception occur; after all, at analysis time, every point in X would be a potential source of exceptional flow; therefore, we only keep track of the exceptional flow at the level of X, and only if X is guarded in a try block.

We would like to have a CFG that is not as crowded as an ExceptionalUnitGraph but, at the same time, represents the exceptional flow behaviour at a higher level, *i.e.* the level of the exception-handling constructs. In order to do this, we enhance the brief CFG so that one exceptional control flow edge is added from X to the corresponding catch and finally blocks, representing the potential exceptional flow from within X to the exception handler blocks. This all is a little confusing, but it will perhaps become clear why it is needed when the brief graph corresponding to the program in Figure A.10 is presented in Figure A.11.

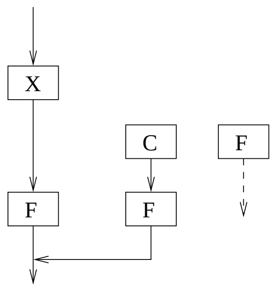


Figure A.11: Brief CFG

The catch and finally blocks in a brief CFG are heads in the graph because the graph does not include exceptional edges; even the nesting information of try-catch-finally structures is lost. The goal is to enhance such a brief CFG so that the exceptional flow information is added to the graph not at the level of individual nodes but at the level of X as shown in Figure A.10, where X is a well-structured segment of code guarded by a try block. This representation preserves the hierarchy of exception-handling blocks without making the graph too crowded to be useful. The enhancement takes into account the control dependence relation among the elements of the target structure; that is, the execution of the catch and finally blocks are all dependent on whether the execution of X raises an exception or not. We choose to insert a dummy node right before X and add an edge from that node to all corresponding handlers (catch/finally) to encode the dependence of the handler blocks on X because X itself might not necessarily be a single node in the CFG.

Figure A.12 shows the CFG after adding the dummy node and the corresponding exceptional flow to the brief CFG in Figure A.11.

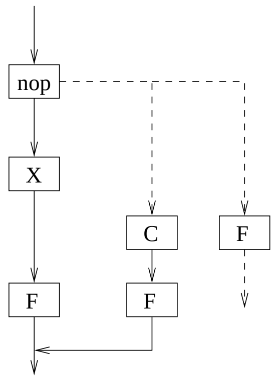


Figure A.12: Adding a dummy node to the brief CFG

There is one additional complexity with the exceptional finally block[8]: at the level of Jimple code, the exceptional finally block ends with a throw instruction and such an instruction, similar to a return instruction, is an exit point in the CFG of a method. If the control reaches this point, it means that X did not finish normally and the flow is exiting out of it exceptionally. Two cases arise: either the finally block (along with the corresponding try block) is guarded by a try-catch-finally or not. In the latter case, the throw at the end of the finally causes the method to exit, hence, it is treated similar to any other instruction in a method that can potentially and implicitly throw an exception (*i.e.* no explicit flow edge is needed.) However, in the former case, the exception could potentially be handled by the outer exception handler, and therefore, the flow would stay within the method and should

[8]By “exceptional finally block” we mean the execution of the finally block when an exception occurs but not handled, or when an exception occurs within the corresponding catch blocks and not handled within them.

be represented in the CFG. Interestingly, we do not need to worry about where exactly such a flow would go in the outer context because, as we explained, we represent all exceptional flow within an x with a single edge from the beginning of x to the corresponding handler block. x in here is another segment of program that contains a try-catch-finally, which contained X.

The only problem that remains is to somehow suppress the tail in the CFG generated by this throw in the *finally* block. If not treated, these tails create a forest, instead of a tree, for the post-dominator relation. The logical solution is to add an edge from the end of the *finally* to the merge point of the corresponding try-catch-finally block. This edge is harmless because the operational behaviour of the program is achieved by the execution of the code (which is not altered by the addition of this edge.) This enhanced CFG is a useful representation because it can make the graph represent the exceptional control dependencies in the presence of exception handling constructs, which is necessary to create a correct program dependence graph. It is not easy to find the merge point to which the auxiliary edge from $throw$ should be added. To get around this problem, we transform the code so that each $throw$ instruction is replaced by a method call that throws the same value as the original expression; however, because a method call merges back to its call site, an edge is automatically present from it to the next node in the flow.

Figure A.13 shows the enhanced CFG corresponding to the brief CFG in Figure A.11. For presentation purposes, in this figure, the exceptional flow out of the exceptional finally block is changed, midway, to a normal flow to indicate that, regardless of the operational behaviour of this finally block (which is throwing an exception), the enhanced CFG considers it as a piece of code, dependent on the execution of X, that merges into the end of the corresponding try-catch-block as explained before. Also, we ignore the exceptional flow from a catch block to the exceptional finally block because incorporating it makes the graph too complex and is not essential for our use case. That said, in the $EnhancedUnitGraph$, which is used to create program dependence graph, we do not differentiate between exceptional flow and normal flow.

Figure A.14 shows an enhanced CFG corresponding to a typical block of code with exceptional control flow constructs. A, B, C, F, and E represent chunks of code that map to the basic blocks in the CFG. As it can be seen, this generated CFG follows the given model for exceptional code. The corresponding PDG is shown in Figure A.15.

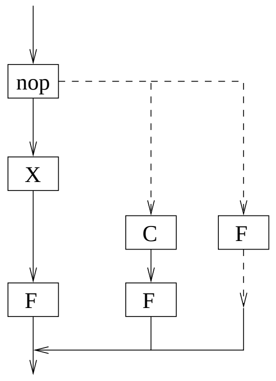
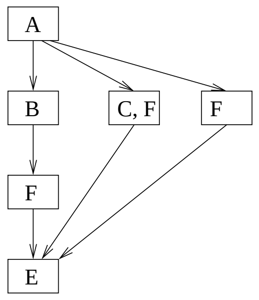


Figure A.13: The enhanced CFG corresponding to the brief CFG



```
A
try {
   B
}
catch(){
   C
}
finally {
   F
}
E
```

Figure A.14: The enhanced CFG of typical exception-handling code

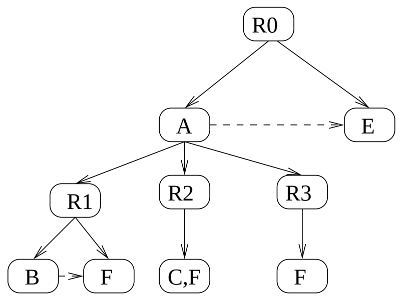


Figure A.15: The PDG for program in Figure A.14

A.3.3 Multiple Exits

Multiple return (or unhandled throw) statements in a CFG result in a post-dominator forest instead of tree. The reason is that the nodes that are considered to be exit nodes do not have any successor, and therefore, do not have any post-dominator. A forest causes some of our algorithms to break or not work correctly. In the construction of the $EnhancedUnitGraph$ (and after taking care of the throw statements that can potentially be handled within the same method) a STOP node is added, if it does not already exist, and if there are more than one exit (tail) in the CFG; then, an edge is added from each tail node to the STOP node so that the post-dominator relation can be represented by a tree (instead of a forest.)

A.4 Conclusion

We presented the construction mechanism, observations, and usage of the program dependence graph representation, implemented in Soot. Also, we explained how we deal with exceptional control flow with a the introduction of a new control flow graph representation. Some parts of the implementation as well as an example usage were presented.