An Extension of AspectJ to Weave Aspect into an Arbitrary Code Region

任意のコード領域にアスペクトを織り込むための AspectJ の拡張

by

Shumpei Akai

赤井駿平

08M37025

2010-01-29

A Master's Thesis Submitted to
Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

In Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Supervisor: Shigeru Chiba

Copyright © 2010 by Shumpei Akai. All Rights Reserverd.

Abstract

Synchronization is a good candidate for modularizing as an aspect in aspect-oriented programming since programmers have to choose the best granularity of synchronization for the underlying hardware to obtain the best execution performance. If synchronization is modularized as an aspect, it allows programmers to change the synchronization code independently of the rest of the program when the program runs on different hardware. However, existing aspect-oriented programming languages such as AspectJ have problems. They do not provide means to select an arbitrary code region as a join point. Moreover, they cannot enforce weaving of a synchronization aspect. Since it is an alternative feature in feature modeling, at least one of available synchronization aspects must be woven. Otherwise, the program would be thread-unsafe. Such a broken program should be detected and fixed before running. Since an aspect in AspectJ is inherently optional, programmers must be responsible for weaving it.

To solve these problems, this paper proposes two new constructs for AspectJ, *regioncut* and *assertions for advice*. Regioncut selects arbitrary code region as a join point. It enables to separate synchronization concern as an aspect in Aspect. To use regioncut you specify a sequence of pointcuts. The region including the join points that match the specified pointcuts in their order is selected as a join point. Assertion for advice provides a way to enforce weaving a mandatory advice. We can detect the lack of the essential advice or aspects that must be woven. To detect it, assertion for advice provides two annotations. One annotation is for the method. This annotation asserts that some advice should be applied to this method. Another annotation is for advice. This asserts that this advice should be woven into the target method.

We implemented these constructs by extending the AspectBench compiler, a extensible AspectJ compiler. We evaluated the design of our constructs by applying them to two open-source software products, Javassist and Hadoop.

Acknowledgments

I am deeply grateful to my supervisor Prof. Shigeru Chiba at Tokyo Institute of Technology. I also want to appreciate Dr. Muga Nishizawa at Rakuten, inc., Natsuko Kumahara at NTT DATA CORPORATION and her research. I would also like to express appreciation to my family for their support.

Contents

1 Introduction **1**

2 AOP and Expressiveness of Pointcuts **7**

2.1 Aspect-Oriented Programming 7

2.2 Motivating Example 8

2.2.1 Synchronization 8

2.2.2 Limitations in AspectJ 9

2.3 Variants of Pointcuts 11

2.3.1 Declarative Event Patterns and Tracematch 11

2.3.2 LoopsAJ 12

2.3.3 Synchronized Block Join Points 13

2.3.4 EJFlow 14

3 Regioncut **16**

3.1 Overview 16

3.2 Regioncut Matching 18

3.2.1 Identifying the first and the last bytecode of a region . 18

3.2.2 Region Expansion 19

3.3 Context Exposure 21

3.4 Implementation 21

3.4.1 Analysis of blocks and statements 22

3.4.2 Around advice support 23

4 An Assertion for Advice **32**

4.1 Overview 32

4.2 Implementation 35

4.2.1 Static Tester 35

4.2.2 Dynamic Tester 35

4.3 Limitation 36

5 Preliminary Evaluation **39**
5.1 Javassist 39
5.2 Hadoop 40
5.2.1 Separating Synchronization concerns by Regioncut . . 40
5.2.2 Assertion for Advices 41
5.3 Performance of Assertion for Advice 42

6 Conclusion **49**

List of Figures

1.1 Synchronized Statement 2
1.2 Lock by ReentrantLock Class 2
1.3 Synchronization Advice with Execution Pointcut 3
1.4 Synchronization by Before and After Advice 4
1.5 Synchronization by Before and After Advice with Exception . 5
1.6 Code Before Refactoring 5
1.7 Code After Refactoring 6

2.1 Example of Around Advice 9
2.2 Pseudo code of pointcut to synchronize a code 10
2.3 Example of Tracematch 12
2.4 Example of LoopsAJ 13
2.5 Example of loop with several successor nodes 13
2.6 Example of Synchronized Block Join Point 14
2.7 Example of EJFlow 15

3.1 Simple Example of Regioncut 17
3.2 Simple Example of Regioncut with three pointcuts 17
3.3 A method including two similar regions 18
3.4 Expression in long statement 19
3.5 The selected region in the first phase does not fit control structure 20
3.6 Context exposure by a regioncut 22
3.7 Example of *markers* 26
3.8 Example of Extraction of Join Point Shadow 27
3.9 A local variable is updated within a code region 28
3.10 A transformed version of the method in Figure 3.9 28
3.11 A program including jumps to out of a selected code region . 29
3.12 A transformed version of the method in Figure 3.11 30

3.13 Example of conflict of regioncuts 31

4.1 Advice with @SolveProblem is invoked by method with @AssertAdvised 33
4.2 Method with @AssertAdvised is invoked by advice with @SolveProblem 34
4.3 An annotation for a method 34
4.4 An annotation for an advice 35
4.5 The implementation of the annotations for advices 37
4.6 An example of false alert by @AssertAdvised 38

5.1 Fine-grained Synchronization by hand 43
5.2 Coarse-grained Synchronization by hand 44
5.3 The advices for fine-grained synchronization 44
5.4 The advice for coarse-grained synchronization 45
5.5 Example Code from Hadoop and Its Synchronization Aspect 1 46
5.6 Example Code from Hadoop and Its Synchronization Aspect 2 47
5.7 The benchmark program for assertion for advice 48

List of Tables

2.1 Typical Pointcuts in AspectJ 8

5.1 The execution time of the Javassist benchmark 40
5.2 The number of synchronization concerns in the TaskTracker class 41
5.3 The number of synchronization advices with regioncuts after the update to Hadoop 0.18.3 42
5.4 The performance comparison of assertion for advice 42

Chapter 1

Introduction

A synchronization concern is a candidate for being implemented as a separate module [26]. Since there are multiple synchronization policies for performance reasons, the software should be distributed with a set of the implementations of different policies and the users should be able to select an appropriate implementation and install it with the rest of the software. The users should not have to modify the program of the rest of the software when they change synchronization policies.

This style of software development leads us to feature-oriented programming [5]. Feature-oriented programming is a programming paradigm to deal with software product-lines [11]. A synchronization concern is one of features that will be incrementally included to compose large software. A synchronization policy (or implementation) is a sub-feature of that feature, or more precisely, an alternative feature in feature modeling [14, 15] since at least one of those sub-features must be included. Otherwise, the resulting software would not be thread-safe.

Aspect-oriented programming (AOP) languages such as AspectJ [16] are useful tools for implementing such a concern as a separate module. For example, AspectJ allows programmers to implement a synchronization policy in a separate module called *aspect* and combine it to the rest of the program without modifying the program. This process is called *weaving*; the aspect is attached at some execution points, called *join points*, to the rest of the program. Since weaving does not require modifying the rest of the program, AOP makes it easier to change synchronization policies to fit the underlying system.

In Java or other Java-base languages including AspectJ, we use synchronized statement to synchronize a code. Figure 1.1 shows the example of

```
1 synchronized(targetObject){
2         //critical section associated with "targetObject"
3 }
```

Figure 1.1: Synchronized Statement

```
1  Lock lock=new ReentrantLock();
2
3  void foo(){
4          lock.lock();
5          try{
6                  // critical section
7          }finally{
8                  lock.unlock();
9          }
10 }
```

Figure 1.2: Lock by ReentrantLock Class

synchronized statement. This language construct is strongly connected with the base code. It is not modular and pluggable. Java5 or later provides an alternative way to synchronized code: java.util.concurrent.locks package. It provide Lock interface and its implementation class ReentrantLock. Figure 1.2 shows how to use ReentrantLock. A critical section should be executed in a try clause, and invocation of unlock() should be occurred in finally clause in order to ensure the lock is released.

However, the implementation of a synchronization concern in AspectJ has problems. First, AspectJ does not allow an arbitrary code region within a method body to be join points. It is not possible to write an aspect that executes an arbitrary code region within a synchronized statement. Synchronization advice for only execution or call pointcuts are available (in Figure 1.3). Programmers must perform refactoring on their programs so that a synchronization aspect can be written within the confines of AspectJ. If we use ReentrantLock object and before and after advices, it seems able to synchronize a region (e.g. Figure 1.4 works). However, it dose not work correctly when exception is raised in the critical section (in Figure 1.5). It cannot release the lock. after advice can catch an exception raised in the its join points, but cannot catch one from other sites. Thus AspectJ dose not

```
1
2  void foo(){
3          //do something
4  }
5
6  void around(Object obj):execution(* *.foo()) && this(obj){
7          synchronize(obj){
8                  proceed();
9          }
10 }
```

Figure 1.3: Synchronization Advice with Execution Pointcut

provide a way to separate fine-grained synchronization as an aspect.

Another problem is that AspectJ does not provide a mechanism for enforcing synchronization implemented as an aspect. In AspectJ, weaving an aspect is optional at compile time or load time. If the programmer refactored the programs that contains join points or regions, an aspect may not be woven (in Figure 1.6 and Figure1.7). It is valid to run the program without a synchronization aspect. However, a synchronization concern is mandatory and a synchronization aspect is a alternative feature. At least one of aspects must be woven. Thus Aspect-Oriented programming language should tell programmers existence of such an unwoven aspect.

Solution by This Thesis

To address these two problems, this paper proposes new language constructs for AspectJ: *regioncut* and *assertions for advice*. Regioncut is a new kind of pointcut designator for selecting an arbitrary code region. To select a region, programmers specify the first and last join point in the region by giving pointcuts to the regioncut. The region is statically determined and, if it does not fit the control/block structure of the program, the region is implicitly expanded to fit. The region selected by regioncut can be applied around advice. It means that synchronization concern can be implemented as an aspect.

An assertion for advice tests at runtime or before runtime whether or not a specified advice is woven and it modifies the program behavior of a specified method. It is useful to enforce that at least one aspect is woven among several aspects that implement the same concern but are woven at

```
1
2  void critical(){
3          //do something
4          begin();
5          //critical section
6          end();
7          //do something
8  }
9
10 before(): call(* *.begin()){
11         lock.lock();
12 }
13 after(): call(* *.end()){
14         lock.unlock();
15 }
```

Figure 1.4: Synchronization by Before and After Advice

different join points or now woven. We provide an assertion for advice as annotations. To use assertion for advice, the method which needs advice should be annotated with a name of concern or name of problem. Advice also should be annotated with the name of concern which it solves and the class name advice should be woven in. Our checker analyzes and warns using these annotations.

The Structure of This Thesis

From the next chapter, we present background, specification and implementation issues of regioncut and assertion for advice The structure of the rest of this thesis is as follows:

Chapter 2: Aspect-Oriented Programming and Expressiveness of Pointcuts

We first mention Aspect-Oriented Programming and its features. Then we show our motivation: separating or modularizing synchronization using Aspect-Oriented Programming, and introduce existing language constructs for addressing the problem.

```
1
2  void critical(){
3          //do something
4          begin();
5        throw new RuntimeException();
6          end();
7          //do something
8  }
9
10 before(): call(* *.begin()){
11         lock.lock();
12 }
13 after(): call(* *.end()){
14         lock.unlock();
15 }
```

Figure 1.5: Synchronization by Before and After Advice with Exception

```
1
2  void foo(){
3          bar();
4  }
5
6  void around(): call(* *.bar()){
7          proceed();
8  }
```

Figure 1.6: Code Before Refactoring

Chapter 3: Regioncut

To separate regions, especially synchronization concerns, we propose a new language construct named *regioncut* for AspectJ. We explain the specification of regioncut and its implementation issues.

Chapter 4: An Assertion for Advice

In this chapter we propose a new feature for AspectJ : *assertion for advice*. An assertion for advice is used to detect the lack of advice. We explain its specification and the implementation issues.

```
1
2 void foo(){
3         bar(arg);
4 }
5
6 void around(): call(* *.bar()){
7         proceed();
8 }
```

Figure 1.7: Code After Refactoring

Chapter 5: Preliminary Evaluation

To show effectiveness of regioncut and assertion for advice, we applied them to the existing softwares. We find that our solution work effectively.

Chapter 6: Conclusion

We conclude this thesis.

Chapter 2

AOP and Expressiveness of Pointcuts

2.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a new programming paradigm to complement Object-Oriented Programming. It allows programmers to modularize concerns — logging, error handling, synchronization etc. — that are scattering in Object-Oriented Programming. Such concerns are called *crosscutting concerns*, and AOP helps to separate them.

In AOP, crosscutting concerns can be separated and packed into the new kind of module, which is called an *aspect*. To modularize crosscutting concerns, AOP provides *pointcuts* and *advice*. A pointcut is a construct for specifying an execution point in the programs that a programmer extend or change the behavior. Advice is the code which is invoked when the execution point specified by the pointcut is executed.

The representative Aspect-Oriented Programming Language is AspectJ [16]. It is based on Java and provides Aspect-Oriented features. AspectJ provides pointcuts show in Table 2.1. Pointcuts can be combined by logical operator: conjunction and disjunction.

AspectJ provides three types of advice: *before*, *after* and *around*. Before advice is invoked before the join point, and after advice is invoked after the join point. Around advice is is invoked instead of the specified join point, *proceed* special function can calls the join point in the advice.

Figure 2.1 is an usage example of around advice. If there is no aspect in the example, the `println` method in the `foo` method prints *"method_foo"*. If the advice `Example` is available, the method prints *"method_foo_advised"*.

pointcuts	*join points the pointcut matches*
call(*method*)	when the *method* is called
execution(*method*)	when the *method* is executed
get(*field*)	when the *field* is read
set(*field*)	when the *field* is set
within(*class*)	join point which appear in the *class*
this(*var*)	when the *this* at join point is instance of *var*'s type
target(*var*)	when the receiver of join point is instance of *var*'s type
args($arg_0, arg_1, \ldots$)	when arg_n is instance of *var* 's type

Table 2.1: Typical Pointcuts in AspectJ

The advice is invoked instead of the println method, the argument is bound to the *str* variable, and proceed calls original println.

2.2 Motivating Example

2.2.1 Synchronization

In 2006, we received a bug report for Javassist [1]. Javassist [7] is a Java class library for modifying Java bytecode and it is widely used in a number of Java products, mainly web application frameworks such as Redhat JBoss application server and Hibernate. The bug was that a method generating a proxy object was not thread-safe: there is no synchronized statement. To fix this bug, we had to modify the method to contain synchronized statements.

An interesting issue of this bug fix was lock granularity; which code block should be put into a synchronized statement. Since lock granularity affects concurrency, minimizing the granularity generally improves execution performance when multiple processor cores are available. However, as we discussed in our previous paper [18], excessive concurrency often have negative impact on performance. In year 2006, low-end servers were still single-processor machines and 4-way multi-processors machines were expensive (Intel Core-MA Xeon "Woodcrest" was shipped in 2006). On a single- or 2-way machine, small granularity may not improve execution performance under a heavy workload. In programs with small granularity synchronization invoke lock and unlock operations many times, and these operations spend more time than other operations. Small granularity synchronization may reduce performance. Thus, for the users who run their software on such a relatively slow machine, we should have modified Javassist to make the

```
1  public void foo(){
2      //do something
3      System.out.println("method_foo");
4      //do something
5  }
6
7  aspect Example{
8          void around(String str) :
9            call(void *.println(String) && args(str)
10         {
11           System.out.println("before");
12           proceed(str + "_advised");
13           System.out.println("after");
14         }
15 }
```

Figure 2.1: Example of Around Advice

lock granularity larger.

This experience shows that a synchronization concern is a good candidate for an aspect. If multiple implementations of the synchronization are supplied as a set of aspects, users can choose the best implementation and weave the aspect for that implementation when they install the software. They do not have to modify the rest of the program when they change the implementation. On the other hand, when we fixed the synchronization bug, we had to choose one implementation and hard-wire it since the software was written in pure Java, not AspectJ. The resulting software ran fast on some kind of hardware but not on other kinds.

2.2.2 Limitations in AspectJ

AspectJ can modularize various concerns such as logging and Observer pattern [16] by aspects but it has two problems for modularizing synchronization concerns. The first one is the granularity of the join points in AspectJ. In AspectJ, call pointcut selects method invocation or new. When call matches method invocations, it corresponds to *invokevirtual*, *invokespecial*, *invokestatic* or invokeinterfacein JVM's bytecode. If call matches new, it corresponds to a sequence of *new* and *invokespecial*. Get and set pointcuts select field accesses. They correspond to *getfield*, *putfield*, *getstatic*, and/or *putstatic*. A join point selected by call, get, or set pointcut corresponds to a

```
1  public void foo(){
2          //do something
3
4          beginCriticalWork();
5          //do critical something.
6          endCriticalWork();
7
8          //do something
9  }
10
11 aspect Synchronize{
12   void around() :
13   <from beginCriticalWork() to endCriticalWork() in method foo>
14   {
15     synchronize(someLockObject){
16       proceed();
17     }
18   }
19 }
```

Figure 2.2: Pseudo code of pointcut to synchronize a code

single bytecode instruction (method invocation or field accesses). It is too fine-grained for synchronization. A join point selected by execution pointcut is a whole method body and hence it is too coarse-grained. Single get and set operation do not need to be synchronized because they are not critical section. If we synchronize a method call or a method body, it is too coarse to improve concurrency. To implement a synchronization concern, it should be possible to select an arbitrary code region as a join point. For example, programmers should be able to select a code region from some statement to another statement in a method body as they insert a synchronized statement there (like Figure 2.2).

Some programmers might think AspectJ has sufficient expressiveness for implementing a synchronization concern. Programmers can modify a method body and extract a new method for such a code region. Then they can write an around advice with execution pointcut, which runs when that new method is attempted to be invoked. This around advice can execute the new method by proceed within a synchronized statement. However, our goal is to provide several aspects each of which implements a different synchronization policy: fine-grained granularity, coarse-grained, synchronized statement, ReentrantLock object in java.util.concurrent.locks package and so

on. Each aspect needs to put a different code region of the method body into a synchronization statement. It is not practical to modify an original method body to extract several new methods for those code regions. Strongly associated codes/methods may scatter. The readability of the resulting method body might be decreased. If two code regions intersect with each other, extracting a sub method for each code region is not possible.

The second problem is how to guarantee that at least one synchronization policy is applied. If no synchronization policy is applied, the program is thread-unsafe; this is a bug. However, if a synchronization policy is implemented as an aspect in AspectJ, we cannot confirm that the synchronization aspect is actually woven and synchronization is performed at runtime. In particular, when the base program is modified later, the existing synchronization aspect might be accidentally made not to work any more due to the fragile-pointcut problem [20]. This is a general problem of using an AspectJ's aspect for implementing an alternative feature in feature modeling [14, 15]. In our scenario, a synchronization concern is a feature in the contexts of Feature Oriented Programming [5]. The synchronization policies are alternative sub-features of that feature, that is, a set of sub-features one of which must be included. Implementing a feature by an aspect is good practice and it is not a new idea. Since an aspect can be attached and detached at a flexible *join point* to a base program without modifying the base program, *i.e.* due to the obliviousness property of AOP [10], it is a useful tool for implementing a feature [3]. This is definitely true for optional features but not for alternative (or mandatory) features.

2.3 Variants of Pointcuts

Several variants of pointcuts are proposed for improving expressiveness of AOP language. In this section, we give examples of new pointcuts.

2.3.1 Declarative Event Patterns and Tracematch

Declarative event patterns [23] and *Tracematches* [2] provide history-based pointcuts. With these pointcuts, an advice can be invoked when the given pattern matches on a dynamic execution history. In tracematch, its dynamic execution history is called *trace*. *Trace* consists of entrance and exit of method calls. Figure 2.3 is an example of tracematch. To use tracematch, you define symbols that represents pointcuts, and order them. This tracematch matches the exit of b() if the a() is entered before the exit of b().

```
1
2  tracematch ( ) {
3    sym a before: call(*.a(..));
4    sym b after: call(*.b(..));
5
6    a b
7
8    {
9      System.out.println("a;b;" );
10   }
11 }
```

Figure 2.3: Example of Tracematch

History-based pointcuts cannot be used to invoke an advice before a code sequence matching the given pattern starts running, since these pointcuts use execution history. It can be used only to invoke after the code sequence finishes running. They cannot weave around advice into code sequences. Thus, those pointcuts are not appropriate for separating a synchronization concern. Pointcuts should know future (like *GAMMA* [17]) to weave around advice into regions.

2.3.2 LoopsAJ

LoopsAJ [12] provides a pointcut for selecting a loop join point, which corresponds to a loop body. It allows parallelizing the execution of the specified loop body. Figure 2.4 is a example of LoopsAJ. The former pointcut matches any loop bodies. The latter matches *for(int i=MIN; i¡MAX; i+=STRIDE)* style loop bodies; *MIN*, *MAX* and *STRIDE* are passed to arguments of the advice. Since proceed() can be called in another thread, you can parallelize each loop body.

Loop join point is not so powerful to choose loop join points. If a method body contains multiple loops, LoopsAJ cannot distinguish these loops. The loop() pointcuts can distinguish loop join points by args.

A loop join point has another problem. A loop join point may not be applied around advice. If a loop has several successor nodes, it has labeled break for example (Figure 2.5), you cannot weave around advice.

```
1
2  void around(): loop(){
3          synchronize(lockObj){
4                  proceed();
5          }
6  }
7
8  void around(int min , int max , int stride):
9      loop() && args(min, max, stride)
10 {
11         Runnable r = new Runnable () {
12                 public void run () {
13                         proceed(min, max, stride);
14                 }
15         } ;
16         new Thread(r).start();
17 }
```

Figure 2.4: Example of LoopsAJ

2.3.3 Synchronized Block Join Points

Xi et al. proposed *synchronized block join points* [24, 25]. The contribution of their work is to enable selecting synchronized statements as join points. With their work, programmers can select existing synchronized statements. Figure 2.6 shows an example of synchronized block join points. Synchronize() pointcut can select synchronized statement. A new construct rm_proceed() calls the block of synchronized statement without synchronization. So, you can cancel the synchronized() statement and re-implement another strategy

```
1 outer:
2 while(...){
3   while(...){
4     if(cond()){
5       break outer;//exit 1
6     }
7   }//exit 2
8 }
```

Figure 2.5: Example of loop with several successor nodes

```
1
2  void around(Map m): synchronize()
3     &&withincode(*.foo()) && args(m)
4  {
5     try{
6           lock.lock();
7           rm_proceed();
8     }finally{
9           lock.unlock();
10    }
11
12 }
```

Figure 2.6: Example of Synchronized Block Join Point

of synchronization. However they cannot change the granularity of synchronization or add new synchronization code within a method body without a synchronized statement.

2.3.4 EJFlow

Cacho et al. proposed *EJFlow* [6]. EJFlow provides AOP constructs for exception handling. It allows programmers to select the flow of an exception. In EJFlow, *explicit exception channel* is used to abstract exception handling. Explicit exception channel is a 5-tuple: exception types, raising sites, handling sites, intermediate sites and exception interface. Figure 2.7 shows an example of EJFlow. The exception handling advice is invoked when an exception of Exception1 is raised in method A(), A() is not called from S() and the exception reached call site F().

```
1
2  pointcut raisingSite: withincode(public void A());
3  pointcut intermedeateSite : !withincode(public void S());
4  pointcut EEC() : echannel(Exception1, raisingSite, intermedeateSite);
5
6  void ehandler() boundto(EEC1()) catching(Exception1 e):
7  withincode(public void F()){
8  {
9          //exception handling
10 }
```

Figure 2.7: Example of EJFlow

Chapter 3

Regioncut

To solve the problems mentioned in the Section 2.2, we first propose a new construct named *regioncut* for AspectJ. A regioncut behaves like pointcuts but it selects code regions, not execution points. It statically determines the selected regions at compile-time to use around advice. A regioncut helps to separate various concerns such as synchronization, exception handling, and transaction.

3.1 Overview

The syntax of a regioncut is simple. A regioncut takes an ordered list of pointcut designators separated by a comma. Then it selects the code region including the join points selected by every pointcut designator in regioncut in that order. Only the call, get and set pointcut designators are available for a regioncut, which can statically select a single expression. A regioncut selects a code region within a method body; it does not select a code region stretching over multiple method bodies.

Figure 3.1 shows an example of regioncut. The parameter to the regioncut is a list of two pointcut designators. It selects a code region that starts with a method call to List.get(int) and ends with a field access to Foo.bar.

A parameter to a regioncut can be a list of more than two pointcut designators (Figure 3.2). Now the regioncut selects a code region in which the List.get(int) method is first called, then the Foo.foo field is set, and finally the Foo.bar field is read. The code region may contain other statements and expressions between the call to the List.get method and the access to the Foo.bar field. The access to the Foo.foo field is not the only expression between them.

```
1  void around():
2    region[
3      call(Object List.get(int)),
4      get(int Foo.bar)
5    ]
6  {
7          synchronized(lockObject){
8                  proceed();
9          }
10 }
```

Figure 3.1: Simple Example of Regioncut

```
1  void around():
2    region[
3      call(Object List.get(int)),
4      set(* Foo.foo),
5      get(int Foo.bar)
6    ]
7  {
8          synchronized(lockObject){
9                  proceed();
10         }
11 }
```

Figure 3.2: Simple Example of Regioncut with three pointcuts

Specifying an intermediate join point like set(* Foo.foo) is useful to distinguish similar code regions in the same method. For example, the pointcut in Figure 3.1 matches two regions in the method in Figure 3.3. If we want to select only the region from line 4 to 6, we must use the pointcut the pointcut in Figure 3.2 instead. The region from line 10 to 12 is not selected.

A code region selected by a regioncut is a collection of consecutive statements. The boundary of the code region is never in the middle of a statement. Suppose that a join point selected by an argument to the regioncut, for example, a method call is a term of some long expression (Figure 3.4). In this case, the selected code region would be expanded to include the whole statement for that long expression. If the selected join point is in the else block of an if statement, as we below describe, the whole if statement may

```
1  void methodWithSimilarRegions(List l,int i,Foo f){
2    int n;
3
4    Object o=l.get(i);
5    f.foo=o;
6    n= f.bar;
7
8    //do something
9
10   Object o=l.get(i+1);
11   System.out.println(o);
12   n= f.bar;
13 }
```

Figure 3.3: A method including two similar regions

be included in the selected code region.

3.2 Regioncut Matching

In this section, we describe how a regioncut matches on a code region. The matching algorithm consists of two phases.

3.2.1 Identifying the first and the last bytecode of a region

In the first phase, the compiler finds the first and the last join point (shadow) of a code region. It first constructs a sequence of all join points from a method body. It is a sequence of consecutive join points, in the lexical order, from the first to the last statement of the method body. The compiler may constructs multiple sequences. If the method body includes an if statement, it constructs two possible sequences; one goes to the *then* block and the other goes to the *else* block. A while statement (and other loop statements) also constructs two sequences. One goes through the loop body while the other goes to the conditional expression but skips the whole loop body. Loop iteration is ignored. A try-finally statement is treated as normal sequence of statements, it goes a try block first and a finally block next. However, if the body includes catch clauses, they dose not appear in normal sequences, which start from the head of a method. A catch clause constructs sequences which start from head of the clause and end with the tail. A return statement

```
1  void foo(){
2
3    //rcLong selects from here
4
5    f(g(h(i(begin())));
6
7    //do something;
8
9    j(k(l(m(end())));
10
11   //to here
12 }
13
14 pointcut rcLong():
15   region[
16         call(∗.begin()),
17         call(∗.end())
18   ];
```

Figure 3.4: Expression in long statement

is regarded as the end of a sequence. The compiler constructs all possible combinations of the multiple sequences made by these control statements.

Then irrelevant join points, which are never selected by an argument to the regioncut, are filtered out in all the constructed sequences of join points. For example, the pointcut shown in Figure 3.2 takes three arguments. Thus the compiler eliminates all join points except a call to the List.get method, an update of the Foo.foo field, and a read from the Foo.bar field.

Finally, the compiler searches all the constructed sequences for a sub sequence that the regioncut exactly matches on. Each join point in the subsequence must be selected in the same order by a pointcut given as an argument to the regioncut. In the case of the pointcut rc2, the compiler finds a subsequence consisting of three join points, which are a call to List.get, an update of Foo.foo, and a read from Foo.bar in this order. The join point selected by the first argument to the regioncut is the beginning of the code region while one selected by the last argument is the end of the code region.

3.2.2 Region Expansion

In the second phase, the compiler determines a set of statements included in the selected region. The initial candidate of the set is the statements

```
 1  pointcut rc1():
 2    region[
 3      call(* *.a()),
 4      call(* *.b())
 5    ];
 6
 7  void foo(){
 8    //do something
 9    if(cond){
10      a();
11    }
12    b();
13    //do something
14  }
```

Figure 3.5: The selected region in the first phase does not fit control structure

including the join points (shadow) of the subsequence found in the first phase.

However, this candidate might not fit control/block structures of the method body. In Figure 3.5, a regioncut rc1 selects the region from a() (line 9) to b() (line 12) after the first phase. Since the method a is not called when cond is false, the selected region does not fit the static block structure of the method body. We cannot apply around advice, which internally separates the region into a method at compile-time, to such a region. But we need to apply around advice specified by such a join points.

To weave such around advice into such a region, the region should be expanded. Our compiler expands the initial region to avoid inconsistency between the selected region and the static block structure of the method. The region after the expansion is the smallest region that contains the initial region and fits the static block structure. If we surround the region by curly braces, the resulting method body is still syntactically valid. In Figure 3.5, the initial region is expanded to contain the whole if statement. The resulting region contains the if statement and a method-call statement to b (line 8 to 12).

To implement this expansion, the compiler constructs a tree representing control structures within a method body. Each node of the tree is either a block, if, while, do, for, switch, try or a synchronized statement. The leaves of the tree are the other kinds of statements: method call statement, field

access statement, assignment statement and others. The children of a node are statements included in the block(s) of the statement represented by that node.

The compiler first finds the smallest sub-tree t that includes the initial set of the statements selected by pointcuts in the regioncut. Then, for each direct child node of the root of that sub-tree t — in other words each child at second level from the root of t — the compiler tests whether or not the child includes (part of) the initial set. If the child does not include, it is removed. The remaining consecutive children are the code region finally selected by the regioncut.

3.3 Context Exposure

In Java, a synchronized statement takes an object that will be locked. To implement a synchronization concern by an aspect, the object must be available within an advice body. However such a object is not always accessible int advice.

A regioncut can be used with other pointcut designators including this, args, and target. If a locked object is stored in a field of this object, the this pointcut can be used to obtain this object. The object in the this's field can be obtained via this object. If a locked object is in a parameter object or a target object, the args or target pointcut can be used to obtain it. If the value bound to the parameter to args or target is from a local variable or a field and the variable (or a field) is available at the beginning of the code region, the value of the variable (or a field) at the beginning of that region is passed to an advice body as an argument. Otherwise, if the variable is not available at that point, or if the parameter to args or target is a compound expression, then a compile error is reported.

Figure 3.6 is an example of context exposure. The regioncut for the around advice selects a code region from line 5 to 7. The argument to b at line 6 is taken from a local variable i, which is initialized before the call to a at line 5, and the target object of the field access at line 7 is directly taken from the obj field. Hence the advice parameters o and n are bound to the values of i and obj at line 5.

3.4 Implementation

We implemented regioncut for AspectJ as an extension to the AspectBench Compiler (abc) [4]. The intermediate language of abc is Jimple [22]. We use

```
1  class Foo{
2    SomeObject obj;
3    void bar(){
4      int i=10;
5      a();
6      b(i);
7      obj.c;
8    }
9  }
10
11 void around(SomeObject o, int n):
12   region[
13     call(* *.a()),
14     call(* *.b(int)) && args(n),
15     get(* SomeObject.c) && target(o)
16   ]
17 {
18   proceed(o,n);
19 }
```

Figure 3.6: Context exposure by a regioncut

Jimple for pattern matching for regioncut.

3.4.1 Analysis of blocks and statements

Jimple has no information about where blocks, statements, and control structures start and end. If we do control flow analysis or decompilation, we can know the information of control structures at some level. However, simple blocks to add scope cannot be restored. So, we decided to extend Jimple directly to make this information available, in order to perform the region expansion described in Section 3.2.2,

We introduced a new Jimple instruction *marker*. A marker has two properties. One is a kind of the structure, which is statement, block, `if`, `while`, or others. The other is whether the marker is beginning or ending. We modified the Java-to-Jimple compiler so that a pair of marker will surround all the instructions of each statement. Other structures such as a block are also surrounded by a pair of markers. Java code is internally compiled to Jimple like Figure 3.7.

After the region expansion, all the inserted markers are removed and then Jimple instructions are converted into Java bytecode.

3.4.2 Around advice support

To implement proceed in an around advice, the abc compiler extracts a new static method from the code corresponding to a join point shadow [19], for example, one selected by execution and initialization pointcuts. Figure 3.8 shows a simplified example of extraction of a join point shadow. The call of the method b() in the method bar() is moved to the static method aShadow(). The statement which previously called b() calls anAdvice() now. The values of method parameters and so on at the join point shadow are passed as arguments to the static method. We adopt and extend this implementation technique for supporting an around advice with a regioncut.

Assignments to local variables

Suppose that a static method is extracted for a code region selected by a regioncut. If a new value is assigned to a local variable within that code region and that variable is declared out of the region, then the new value must be first stored in another local variable in the extracted static method and then reflected on the original local variable. Join points selected by ordinary pointcuts in AspectJ do not cause this problem; such a join point never contains assignment to local variables. In regioncut, however, we should deal with this problem.

To implement this behavior, we make an object whose fields are copies of the local variables accessed in the code region. It is passed to the extracted static method and, if some fields of the object are updated in the static method, then the updated values are copied back to the original local variables.

For example, in Figure 3.9, the code region between the two method calls to a and b is selected and hence a static method is extracted from that region. The assignment to a local variable s at line 4 must be transformed so that the value assigned at line 4 is reflected on the value of s at line 6. Before the aspect is woven by the original weaver of abc, therefore, the toBeAdvised method is transformed into the code shown in Figure 3.10. The class LocalStorage$toBeAdvised is a helper class generated during this transformation. The static method extracted for the region from line 6 to 12 receives the value of *$localStorage* as an argument.

A helper class such as LocalStorage$toBeAdvised is generated per code region. Each field has the same type as the corresponding local variable. We do not use a java.util.HashMap object or an Object array for *$localStorage*. These are more generic but type conversion or boxing/unboxing is needed

when a value is stored and obtained from them.

Jumps to the outside of the region

Jumps — break, continue, and return statements — must be also transformed when a static method is extracted from a code region selected by a regioncut. The destination of these jumps may be out of that region. Figure 3.11 shows a program including jumps to out of the selected region. Note that, in Jimple and Java bytecode, break and continue statements are represented by goto instructions.

For the transformation, each jump instruction from the inside to the outside of the region is given a unique identification (id.) number. Next, each jump instruction is replaced with the following instructions:

1. Save the id. number into a local variable (jump id. variable).
2. Jump to the tail of the region.

Furthermore, at the tail of the region, a switch statement is inserted. It branches to the destination specified by the jump id. variable. Figure 3.12 shows the resulting program after the transformation above. Then our extended abc compiler extracts a static method from the code region from line 5 to 16 in Figure 3.12. The abc compiler is responsible to maintain the consistency of the value of the jump id. variable $i between the extracted static method and the original method includeJump. A return statement is transformed in a similar way.

Current Limitation

Two regioncuts may select two code regions intersecting each other. Our compiler cannot implement around advices for those two regions. In Figure 3.13 the region from beginA() to endA() and the region from beginB() to endB() are intersection. We cannot apply around advice these regions. The compiler can implement if one of the selected regions is nestedly contained in the other region. We should define the priority rule of intersected regions to implement around advices for such regions. For example, if we define a rule — a precedent region should be enlarged to contain following regions —, we can solve this problem. However, we should study what rule is useful for programmers.

Regioncuts are fragile when the code is refactored. In order to make regioncuts more robust, we plan to inline method calls while analyzing. We

think, by this inlining, regioncuts may still pick out regions after the code is modified using *extract method* refactoring. However, it is not enough for other refactorings. We will introduce an assertion for advice in next section, for refactorings.

Java code

```
1 public static int fact(int n){
2   if(n<=0){
3     return 1;
4   }else{
5     return n * fact(n−1);
6   }
7 }
```

Pseudo Jimple code

```
1  public static int fact(int n){
2    BLOCK:begin:0;
3      IF:begin:1;
4          STMT:begin:2;
5            if n > 0 goto BLOCK:begin:5;
6          STMT:end:2;
7        BLOCK:begin:3;
8          STMT:begin:4;
9            return 1;
10         STMT:end:4;
11       BLOCK:end:3;
12       BLOCK:begin:5;
13         STMT:begin:6;
14           $i0 = n − 1;
15           $i1 = fact($i0);
16           $i2 = n * $i1;
17           return $i2;
18         STMT:end:6;
19       BLOCK:end:5;
20     IF:end:1;
21   BLOCK:end:0;
22 }
```

Figure 3.7: Example of *markers*

Before Extraction

```
1   public void bar(int n){
2           a();
3           b(n);
4           c();
5   }
6
7   void around(int num): call(* *.b(int)) && args(num){
8           // do something
9           proceed(num);
10          // do something
11  }
```

After Extraction

```
1   public void bar(int n){
2           a();
3           anAdvice(n); // call an advice
4           c();
5   }
6
7   void anAdvice(int num){
8           // do something
9           aAhadow(num); // call join point shadow
10          // do something
11  }
12
13  static void aShadow(int num){
14          b(num);
15  }
```

Figure 3.8: Example of Extraction of Join Point Shadow

```
1  public void toBeAdvised(int x){
2      String s="initial␣string";
3      a();
4      s="string␣was␣replaced";
5      b();
6      System.out.println(s); // what is the value of s?
7  }
8
9  void around():
10     region[
11         call(* *.a()),
12         call(* *.b())
13     ]
14 {
15     proceed();
16 }
```

Figure 3.9: A local variable is updated within a code region

```
1  public void toBeAdvised(int x){
2      String s="initial␣string";
3      $localStorage = new LocalStorage$toBeAdvised();
4
5      $localStorage.s=s;
6      nop; //label for the beginning of the region
7      s=$localStorage.s;
8      a();
9      s="string␣was␣replaced";
10     b();
11     $localStorage.s=s;
12     nop; // label for the end of the region
13     s=$localStorage.s;
14
15     System.out.println(s);
16 }
```

Figure 3.10: A transformed version of the method in Figure 3.9

```
1  public void includeJump(){
2      labelOfFor:
3      for(;;){
4          while(true){
5              a();
6              if(innerBreak()){
7                  break; //goto label0;
8              }else{
9                  break labelOfFor; //goto label1;
10             }
11             b();
12         }
13         //label0:
14     }
15     //label1;
16 }
17
18 void around(): region[call(* *.a()),call(* *.b())] {
19     proceed();
20 }
```

Figure 3.11: A program including jumps to out of a selected code region

```
1  public void includeJump(){
2      labelOfFor:
3      for(;;){
4          while(true){
5              nop; //label for beginning of the shadow
6              a();
7              if(innerBreak()){
8                  $i=0;
9                  goto endLabel;
10             }else{
11                 $i=1;
12                 goto endLabel;
13             }
14             b();
15             endLabel:
16             nop; //label for end of the shadow
17             switch($i){
18             case 0: goto label0;
19             case 1: goto label1;
20             }
21         }
22         label0:;
23     }
24     label1:;
25 }
```

Figure 3.12: A transformed version of the method in Figure 3.11

```
1  void withConflict(){
2    beginA();
3    beginB();
4    endA();
5    endB();
6  }
7
8  void around():
9    region[call(* *.beginA()),call(* *.endA())]
10 {
11   proceed();
12 }
13
14 void around():
15   region[call(* *.beginB()),call(* *.endB())]
16 {
17   proceed();
18 }
```

Figure 3.13: Example of conflict of regioncuts

Chapter 4

An Assertion for Advice

To solve the problems mentioned in Section 2.2, we also propose a new assertion mechanism for AspectJ. We call this assertion mechanism *assertion for advice*. This assertion enables programmers to test an assumption that a certain advice is woven and it modifies the program behavior at some execution point. This mechanism is useful in particular for an advice with a regioncut, which tends to be fragile. Even small changes of a base program may make the regioncut not to match the code region where the advice must be woven. For example, if the join point specified in the regioncut is moved into another method by*Extract Method* refactoring, the advice with regioncut failed to be woven. The proposed mechanism would make programmers less reluctant to use an aspect to implement an alternative feature such as a synchronization concern.

4.1 Overview

We propose two kinds of annotations: @AssertAdvised and @SolveProblem. @AssertAdvised annotation annotates a method. @SolveProblem annotation annotates an advice. Figure 4.3 and Figure 4.4 show examples. @AssertAdvised declares that the behavior of the annotated method, for example, the foo method in Figure 4.3, must be modified for implementing some concern by an advice annotated by @SolveProblem, for example, the advice in Figure 4.4. The argument to @SolveProblem represents which problem in which class the advice is expected to solve. For example, @SolveProblem("A.name_of_problem") solves the problem of the method with @AssertAdvised("name_of_problem") in class A.

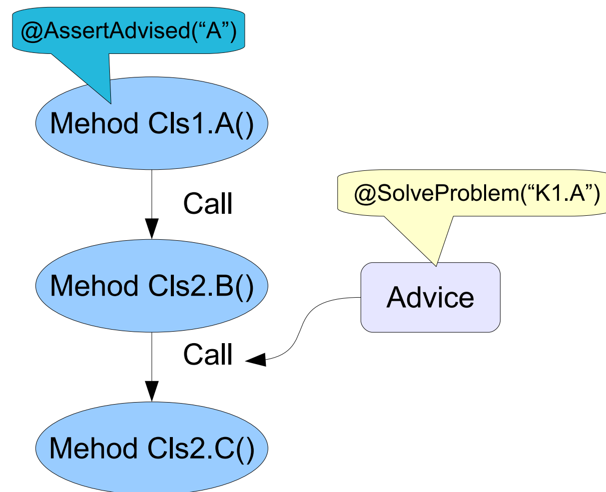

Figure 4.1: Advice with @SolveProblem is invoked by method with @AssertAdvised

@AssertAdvised tests if the annotated method satisfies the following assumption just before the method returns:

- the method is directly or indirectly invoked (through proceed) from the @SolveProblem advice (*i.e.* the method is under the control of the advice, shown in Figure 4.1), or
- the method directly or indirectly invokes the @SolveProblem advice while the method is being executed (*i.e.* the part of the method body is under the control of the advice, shown in Figure 4.2).

Here the @SolveProblem advice is an advice with the @SolveProblem annotation corresponding to the @AssertAdvised of the method. If the test fails, then @AssertAdvised will be thrown.

We did not choose simpler design, in which @AssertAdvised tests if a specific advice is woven at a specific join point. The reason is to allow refactoring on the advice. For example, a synchronization concern can be implemented with different policies. Thus, programmers might replace an original synchronization aspect with a new one they write. The new aspect might be woven at a different join point. @AssertAdvised must consider such a new aspect is woven at a different join point. Our design of @AssertAdvised presented in this paper uses a higher level abstraction and accept such

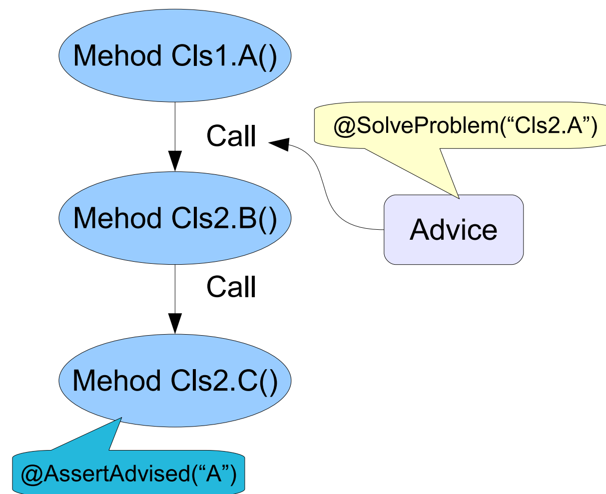

Figure 4.2: Method with @AssertAdvised is invoked by advice with @SolveProblem

```
1 class A{
2   @AssertAdvised("name_of_problem")
3   void foo(){
4     bar();
5   }
6 }
```

Figure 4.3: An annotation for a method

refactoring on aspects.

Note that an @AssertAdvised annotation is not inherited by a subclass. Suppose that a class has a method annotated with @AssertAdvised and its subclass overrides that method. The @AssertAdvised annotation is not added to the overriding method in the subclass unless another @AssertAdvised annotation is explicitly written for the overriding method. This is because the implementation of the overriding method might be different and thus the advice is not needed any more or another kind of advice is needed.

```
1 @SolveProblem("A.name_of_problem")
2 void around(): call(* *.bar()) {
3   //do something
4 }
```

Figure 4.4: An annotation for an advice

4.2 Implementation

In order to test the rule of assertion for advice, we implemented the two testers.

4.2.1 Static Tester

We implemented the static tester of assertion for advice, based on Soot [22]. Soot is a Java optimization framework, and it can produce call graph from java bytecode. We analyze and test assertion for advice using call graph.

Fisrt the tester loads class files and creates the call graph. Then we checks all methods whether it has an @AssertAdvised annotation. When we find the method which is annotated with @AssertAdvised, we search the caller and callee advice. If there are no caller and callee advice which has same problem name, we shows warning message.

4.2.2 Dynamic Tester

The dynamic tester of assertions for advice are implemented by program transformation. First, the following three variables are declared for each pair of @AssertAdvised and @SolveProblem:

- *rm*: the current depth of the nested calls to the annotated method
- *ra*: the current depth of the nested invocations of the annotated advice
- *ca*: true if the annotated advice has been executed since the annotated method started running.

These variables are declared per thread. They are stored in a java.lang.ThreadLocal object. Then the program is transformed to test these variables and throw an exception if necessary. For example, the program in Figure 4.3 and Figure 4.4 is transformed into Figure 4.5.

The runtime test for the assertion is executed only when the assertion mechanism is generally enabled ('-ea' option is given to the java command). The variable $assertionsDisabled is a static final field and it is made true if the assertion mechanism is disabled. Hence, if it is disabled, we expect that the if statements we inserted are eliminated by the runtime optimizer of the JVM.

4.3 Limitation

The assertion for advice is declared for a method by adding @AssertAdvised; it is not directly for a join point. Thus, if a thread of control only conditionally reaches the join point, the assertion may throw false alert. To suppress this, an empty around advice must be redundantly woven.

Figure 4.6 is an example. A synchronization advice is woven for the code region from line 8 to 10. Its policy is to put the object creation at line 9 into a synchronized statement. This is a right policy and the implementation is correct. On the other hand, @AssertAdvised declares that the whole body of the method getInstance or part of it is advised by some synchronization advice. Hence, if the getInstance method is invoked, @AssertAdvised tests the assumption at the return statements. Since singletonEnabled is false, the thread never reaches line 9 or the synchronization advice. Although this is right behavior, @AssertAdvised throws an exception since the advice was not executed.

To suppress this exception, another advice must be woven as well on the object creation at line 13. It does not have to do anything except invoking proceed but it must have the annotation @SolveProblem("Singleton.synchronizeCache").

```
1  class A{
2    void foo(){
3      if(!$assertionsDisabled){
4          rm++;
5      }
6      try{
7          bar();
8      }finally{
9          if(!$assertionsDisabled){
10             rm−−;
11             if(ra==0 && !ca)
12                 throw new AssertionError();
13             if(rm==0)
14                 ca=false;
15         }
16     }
17   }
18 }
19
20 void around(): call(∗ ∗.bar()) {
21     if(!$assertionsDisabled){
22         ra++;
23     }
24     try{
25         //do something
26     }finally{
27         if(!$assertionsDisabled){
28             ra−−;
29             if(rm>0)
30                 ca=true;
31             else
32                 ca=false;
33         }
34     }
35 }
```

Figure 4.5: The implementation of the annotations for advices

```
1  public class Singleton{
2    private static Singleton instance;
3    static final boolean singletonEnabled=false;
4
5    @AssertAdvised("synchronizeCache")
6    public static Singleton getInstance() {
7      if (singletonEnabled) {
8          if(instance==null){
9              instance=new Singleton();
10         }
11         return instance;
12     }else
13         return new Singleton();
14   }
15 }
16
17 aspect SynchornizeCache{
18   @SolveProblem("Singleton.synchronizeCache")
19   void around():
20     region[
21       get(static Singleton Singleton.instance),
22       set(static Singleton Singleton.instance) ]
23     {
24         synchronized(Singleton.class){
25             proceed();
26         }
27     }
28 }
```

Figure 4.6: An example of false alert by @AssertAdvised

Chapter 5

Preliminary Evaluation

We implemented regioncut and assertions for advices by extending the AspectBench 1.3.0 compiler with the JastAdd frontend [13, 9] running on Sun JVM 1.6. Then we evaluated the design of the regioncut and the assertion for advice by applying them to two open-source software products. We use the dynamic tester for assertion for advice in this chapter.

5.1 Javassist

We wrote synchronization aspects for Javassist by using the proposed constructs along the scenario in Section 2.2. One aspect implements fine-grained synchronization. Figure 5.1 shows an equivalent program in which fine-grained synchronization code is embedded by hand. Two synchronized statements are embedded in the createClass2 method. Figure 5.2 shows an equivalent program written by hand for coarse-grained synchronization. One synchronized statement is embedded in the createClass method, which calls createClass2. We could successfully separate the synchronization code into aspects by using regioncut. Figure 5.3 and Figure 5.4 show (the advices in) the aspects.

The Javassist users can easily switch synchronization policies by selecting either of the two aspects. Switching the policies caused performance differences according to our experiment. We ran the benchmark test posted with the bug report [1]. It is a client-server program, in which Javassist is used for the client-side code running 20 threads. For our experiment, we used machines with Intel Xeon (2.83 GHz), Linux 2.6.28 (x64), and Sun JVM 1.6.0. The client machine had 8GB memory and the server one had 4GB memory. They are connected through 1Gbps Ethernet. We disabled

	Time (sec.)	**Std. Deviation**
—quad core		
fine-grain (by aspect)	5.70	0.13
fine-grain (by hand)	5.63	0.13
coarse-grain (by aspect)	7.77	0.26
coarse-grain (by hand)	7.87	0.33
—dual core		
fine-grain (by aspect)	9.94	0.21
fine-grain (by hand)	9.94	0.21
coarse-grain (by aspect)	8.70	0.20
coarse-grain (by hand)	8.76	0.24

Table 5.1: The execution time of the Javassist benchmark

assertion for advice while running this benchmark.

Table 5.1 lists the results. We used two kinds of client machine: one with 4 cores and the other with 2 cores. The numbers are the average of the execution time measured 500 times. The results revealed that using a fine-grained synchronization aspect is better on the 4 core machine while using a coarse-grained one is better on the 2 core machine. The overheads due to using an aspect were negligible.

We could add @AssertAdvised to the createClass method for enforcing synchronization. However, to suppress false alert mentioned in Section 4.3, we had to weave an empty advice on the call to createClass3.

5.2 Hadoop

Hadoop [21] is an open-source framework for distributed computing; it provides a distributed file system and programming supports for the MapReduce computing model [8]. We rewrote the program of Hadoop 0.16.4 in AspectJ with our proposed constructs.

5.2.1 Separating Synchronization concerns by Regioncut

First, we separated synchronization concerns into aspects from the TaskTracker class (2357 LOC) in org.apache.hadoop.mapred package of Hadoop. Table 5.2 lists the result of our experiment. The TaskTracker class contains 21 synchronized statements. We separated all the statements into aspects.

synchronized statements	21
ones separated by ordinary pointcuts	9
ones separated by regioncuts	12

Table 5.2: The number of synchronization concerns in the TaskTracker class

Among them, 9 statements could be separated into aspects by using ordinary pointcut designators such as call, get, set, or execution. We needed the regioncut to separate the rest of the synchronized statements into aspects. Figure 5.5 and Figure 5.6 show the example methods and advice which need regioncut. Note that we did not modify the original source program of the TaskTracker class. If we performed refactoring to extract a new method from the synchronized block, then we would need less regioncuts for separating synchronized statements into aspects.

We also evaluated the necessity of more than two arguments to a regioncut. Recall that a regioncut can take more than two pointcuts as arguments to distinguish similar code regions in the same method body. Among 12 synchronization concerns in the TaskTracker class, 5 concerns needed regioncuts that take more than two pointcuts as arguments. Furthermore, 4 concerns needed our proposed context exposure mechanism.

5.2.2 Assertion for Advices

To evaluate the assertion for advice, we added @AssertAdvised and @SolveProblem annotations for the synchronization aspects separated by regioncuts from the TaskTracker class. We then updated the program from Hadoop version 0.16.4 to 0.18.3. Finally, we compiled the TaskTracker class with the same aspects and ran unit tests, which are bundled with the distribution of Hadoop, to find the aspects that were not correctly woven any more.

Table 5.3 lists the advices after the update. Among 12 advices using a regioncut, 6 advices were not correctly woven for the new version of the program. For 2 of these 6 advices, @AssertAdvised threw an exception during the unit tests. Unfortunately, the rest of the advices were not detected by the assertion for advices. However, it turned out that the unit tests did not invoke the methods annotated with @AssertAdvised for the remaining 4 advices.

the advices with regioncuts in the old version	12
ones correctly woven	6
ones detected by the assertion	2
ones not detected	4

Table 5.3: The number of synchronization advices with regioncuts after the update to Hadoop 0.18.3

	Time (ms)	**Std. Deviation**
assertion disabled	24.7	0.1
assertion enabled	1321.4	34.8

Table 5.4: The performance comparison of assertion for advice

5.3 Performance of Assertion for Advice

We measured the performance of assertion for advice. Figure 5.7 is a benchmark program we used. This benchmark program increments the `counter` field 100,000,000 times, and the advice also increments the same field. For this benchmark, we used the machine with quad-core Intel Xeon (2.83 GHz), with 4 GB memory, Linux 2.6.28 (x64), and Sun JVM 1.6.0.

We ran the benchmark parogram with and without assertion for advice. Table 5.4 is the result. This result shows that, in this case, the speed of the program enabled the assertion for advice is 53.5 times slower than one disabled the assertion.

```
1  private static WeakHashMap proxyCache;
2  private void createClass2(ClassLoader cl) {
3    CacheKey key = new CacheKey(superClass, interfaces, methodFilter
         , handler);
4    synchronized (proxyCache) {
5      HashMap cacheForTheLoader = (HashMap)proxyCache.get(cl);
6      if (cacheForTheLoader == null) {
7        cacheForTheLoader = new HashMap();
8        proxyCache.put(cl, cacheForTheLoader);
9        cacheForTheLoader.put(key, key);
10     } else {
11       CacheKey found = (CacheKey)cacheForTheLoader.get(key);
12       if (found == null)
13         cacheForTheLoader.put(key, key);
14       else {
15         key = found;
16         Class c = isValidEntry(key); // no need to synchronize
17         if (c != null) {
18           thisClass = c;
19           return;
20         }
21       }
22     }
23   }
24
25   synchronized (key) {
26     Class c = isValidEntry(key);
27     if (c == null) {
28       createClass3(cl);
29       key.proxyClass = new WeakReference(thisClass);
30     }
31     else
32       thisClass = c;
33   }
34 }
```

Figure 5.1: Fine-grained Synchronization by hand

```
1  public Class createClass() {
2    if (thisClass == null) {
3      ClassLoader cl = getClassLoader();
4        synchronized (proxyCache) {
5          if (useCache)
6            createClass2(cl);
7          else
8            createClass3(cl);
9        }
10   }
11
12   return thisClass;
13 }
```

Figure 5.2: Coarse-grained Synchronization by hand

```
1  void around():
2    region[
3      call(∗ WeakHashMap.get(..)),
4      call(∗ WeakHashMap.put(..))
5    ]
6  {
7    synchronized(ProxyFactory.class){
8      proceed();
9    }
10 }
11
12 void around(Object key):
13   region[
14     call(∗ ∗.isValidEntry(∗)) && args(key),
15     set(∗ ∗.proxyClass)
16   ]
17 {
18   synchronized(key){
19     proceed(key);
20   }
21 }
```

Figure 5.3: The advices for fine-grained synchronization

```
1  void around():
2    region[
3      get(static boolean *.useCache),
4      call(* *.createClass2(..))
5    ]
6  {
7    synchronized(ProxyFactory.class){
8      proceed();
9    }
10 }
```

Figure 5.4: The advice for coarse-grained synchronization

```
1  private synchronized void purgeJob(KillJobAction action) throws
       IOException {
2    String jobId = action.getJobId();
3    LOG.info("Received␣'KillJobAction'␣for␣job:␣" + jobId);
4    RunningJob rjob = null;
5    synchronized (runningJobs) {
6      rjob = runningJobs.get(jobId);
7    }
8
9    if (rjob == null) {
10     LOG.warn("Unknown␣job␣" + jobId + "␣being␣deleted.");
11   } else {
12     synchronized (rjob) {
13       for (TaskInProgress tip : rjob.tasks) {
14         tip.jobHasFinished(false);
15       }
16       if (!rjob.keepJobFiles){
17         fConf.deleteLocalFiles(SUBDIR + Path.SEPARATOR +
               JOBCACHE +
18                                  Path.SEPARATOR + rjob.getJobId
                                      ());
19       }
20       rjob.tasks.clear();
21     }
22   }
23
24   synchronized(runningJobs) {
25     runningJobs.remove(jobId);
26   }
27 }
28
29 //synchronization advice for the formaer synchronized statement
30 //in purgeJob()
31 void around(Object syncObj):region[
32   get(Set *.tasks) && target(syncObj),
33   call(* JobConf.deleteLocalFiles(String)),
34   call(* java.util.Set.clear())
35
36 ] && withincode(* TaskTracker.purgeJob(KillJobAction))
37 {
38   synchronized(syncObj){
39     proceed(syncObj);
40   }
41 }
```

Figure 5.5: Example Code from Hadoop and Its Synchronization Aspect 1

```
1   private void killOverflowingTasks() throws IOException {
2     long localMinSpaceKill;
3     synchronized(this){
4       localMinSpaceKill = minSpaceKill;
5     }
6     if (!enoughFreeSpace(localMinSpaceKill)) {
7       acceptNewTasks=false;
8       synchronized (this) {
9         TaskInProgress killMe = findTaskToKill();
10
11        if (killMe!=null) {
12          String msg = "Tasktracker running out of space." +
13            " Killing task.";
14          LOG.info(killMe.getTask().getTaskId() + ": " + msg);
15          killMe.reportDiagnosticInfo(msg);
16          purgeTask(killMe, false);
17        }
18      }
19    }
20  }
21
22   void around(TaskTracker t):region[
23    call(TaskInProgress TaskTracker.findTaskToKill()),
24    call(void TaskTracker.purgeTask(..))
25  ] && withincode(* TaskTracker.killOverflowingTasks()) && this(t)
26  {
27    synchronized(t){
28      proceed(t);
29    }
30  }
```

Figure 5.6: Example Code from Hadoop and Its Synchronization Aspect 2

```
1  public class AATest{
2    public int counter=0;
3    void inc(){
4      counter++;
5    }
6    @AssertAdvised("needDoubleInc")
7    public void run(){
8      for(int i=0;i<100000000 ;i++){
9        inc();
10     }
11   }
12
13
14   public static void main(String[] args){
15     AATest aa=new AATest();
16     long start=System.nanoTime();
17     aa.run();
18     System.out.println(System.nanoTime()-start);
19   }
20 }
21
22 aspect DoubleInc{
23     @SolveProblem("AATest.needDoubleInc")
24     void around(AATest self):
25         call(void AATest.inc()) && target(self)
26     {
27         proceed(self);
28         self.counter++;
29     }
30 }
```

Figure 5.7: The benchmark program for assertion for advice

Chapter 6

Conclusion

This paper proposed two language constructs to separate regions, especially with synchronizations, as aspects in AspectJ. *Regioncut* is a new pointcut designator, which helps selecting regions as join points. It allows AspectJ programmers to select arbitrary code regions. Synchronization concerns can be treated in AspectJ by regioncut. An *assertion for advice* allows programmers to detect an unwoven advice. Such absence of advice can be detected quickly by running unit tests with this assertion or using the static checker. We implemented these two constructs by modifying the AspectBench compiler and Soot. We mentioned the issues when implenmenting them.

We applied regioncut to Javassist and showed that our aspects help switching synchronization policies for improving performance. We also wrote aspects for separating synchronization code in Hadoop. We found that regioncut was necessary for separating synchronization code and unwoven advices could be detected when the program was updated as far as the target methods were called in unit tests. Regioncut needs complex information for picking up a region, so its drawback is fragility. However, we think assertion for advice reduces a drawback of regioncut.

Bibliography

[1] : [#JASSIST-28] javassist enhancement failed on deserializing hibernate proxies - jboss.org JIRA, http://jira.jboss.org/jira/browse/JASSIST-28.

[2] Adding trace matching with free variables to AspectJ, New York, NY, USA, ACM, pp. 345–364 (2005).

[3] When to use features and aspects?: a case study, New York, NY, USA, ACM, pp. 59–68 (2006).

[4] abc: an extensible AspectJ compiler, New York, NY, USA, ACM, pp. 87–98 (2005).

[5] Scaling step-wise refinement, Washington, DC, USA, IEEE Computer Society, pp. 187–197 (2003).

[6] EJFlow: taming exceptional control flows in aspect-oriented programming, New York, NY, USA, ACM, pp. 72–83 (2008).

[7] : Load-Time Structural Reflection in Java, London, UK, Springer-Verlag, pp. 313–336 (2000).

[8] MapReduce: simplified data processing on large clusters, Berkeley, CA, USA, USENIX Association, pp. 10–10 (2004).

[9] The JastAdd extensible Java compiler, New York, NY, USA, ACM, pp. 884–885 (2007).

[10] Aspect-Oriented Programming is Quantification and Obliviousness, Addison-Wesley, pp. 21–35 (2005).

[11] : Implementing product-line features by composing aspects, Norwell, MA, USA, Kluwer Academic Publishers, pp. 271–288 (2000).

[12] A join point for loops in AspectJ, New York, NY, USA, ACM, pp. 63–74 (2006).

[13] JastAdd: an aspect-oriented compiler construction system, Sci. Comput. Program., Vol. 47, No. 1, pp. 37–58 (2003).

[14] Feature-oriented domain analysis (foda) feasibility study, Carnegie-Mellon University Software Engineering Institute (1990).

[15] FORM: A feature-oriented reuse method with domain-specific reference architectures, Annals of Software Engineering, Vol. 5, pp. 143–168 (1998).

[16] An Overview of AspectJ, London, UK, Springer-Verlag, pp. 327–353 (2001).

[17] Back to the Future: Pointcuts as Predicates over Traces, ACM (2005).

[18] Aspect-oriented application-level scheduling for J2EE servers, New York, NY, USA, ACM, pp. 1–13 (2007).

[19] Compilation semantics of aspect-oriented programs, ACM (2002).

[20] Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software, Washington, DC, USA, IEEE Computer Society, pp. 653–656 (2005).

[21] : Welcome to Apache Hadoop!, `http://hadoop.apache.org/`.

[22] Soot - a Java bytecode optimization framework, IBM Press, p. 13 (1999).

[23] Implementing protocols via declarative event patterns, New York, NY, USA, ACM, pp. 159–169 (2004).

[24] A synchronized block join point for AspectJ, New York, NY, USA, ACM, pp. 39–39 (2008).

[25] Aspect-oriented support for synchronization in parallel computing, New York, NY, USA, ACM, pp. 1–5 (2009).

[26] : FlexSync: An aspect-oriented approach to Java synchronization, Washington, DC, USA, IEEE Computer Society, pp. 375–385 (2009).