Extending AspectJ for Separating Regions

Shumpei Akai Shigeru Chiba
Tokyo Institute of Technology
akai@csg.is.titech.ac.jp chiba@is.titech.ac.jp

Abstract

Synchronization is a good candidate for an aspect in aspect-oriented programming (AOP) since programmers have to choose the best granularity of synchronization for the underlying hardware to obtain the best execution performance. If synchronization is an aspect, programmers can change the synchronization code independently of the rest of the program when the program runs on different hardware. However, existing AOP languages such as AspectJ have problems. They cannot 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. 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 and assertion for advice enforces weaving a mandatory advice. We implemented these constructs by extending the AspectBench compiler. We evaluated the design of our constructs by applying them to two open-source software products, Javassist and Hadoop.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords Aspect-Oriented Programming, Feature-Oriented Programming, Region, Synchronization

1. Introduction

A synchronization concern is a candidate for being implemented as a separate module. 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]. 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 [13, 14] 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 [15] 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.

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. Programmers must perform refactoring on their programs so that a synchronization aspect can be written within the confines of AspectJ. If we use java.util.concurrent.locks.Lock and before and after advices, we can synchronize the execution of a region in most cases. However, this approach will not release a lock when an exception is raised within the region. 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. It is valid to run the program without a synchronization aspect. However, a synchronization concern is mandatory and a synchronization aspect is an alternative feature. At least one of aspects must be woven.

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. An assertion for advice tests at 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 different join points.

In the rest of this paper, Section 2 shows our motivating example, which is an aspect for customizing lock granularity for synchronization. Then, Section 3 proposes regioncut. Section 4 proposes assertions for advice. Section 5 shows case study. Section 6 describes related work. Section 7 concludes this paper.

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.

2. Motivating Example

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; 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 [17], 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. Thus, for the users who run their software on such a relatively slow machine, we should have modified Javassist to make the 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 Limitations in AspectJ

AspectJ can modularize various concerns such as logging and Observer pattern [15] by aspects but it has two problems for modularizing synchronization concerns. The first one is the granularity of the join points in AspectJ. A join point selected by call, get, or set pointcut corresponds to a 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. 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.

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. 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. 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 [19]. This is a general problem of using an AspectJ aspect for implementing an alternative feature in feature modeling [13, 14]. 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.

3. Regioncut

To solve the problems mentioned in the previous section, 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. 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 and then it selects the code region including the join points selected by every pointcut designator in that order. Only the call, get and set pointcut designators are available for a regioncut, which 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.

The following code shows an example of regioncut.

```
1 pointcut rc1():
2   region[
3     call(Object List.get(int)),
4     get(int Foo.bar)
5   ];
```

The parameter to this 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:

```
1 pointcut rc2():
2   region[
3     call(Object List.get(int)),
4     set(* Foo.foo),
5     get(int Foo.bar)
6   ];
```

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.

Specifying an intermediate join point like set(* Foo.foo) is useful to distinguish similar code regions in the same method. For example, the pointcut rc1 shown above matches two regions in the method in Figure 1. If we want to select only the region from line

```
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 1. A method including two similar regions

```
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 2. The selected region in the first phase does not fit control structure

4 to 6, we must use the pointcut rc2 instead. The region from line 10 to 12 is excluded.

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. 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 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, a join-point sequence splits into two; one goes to the *then* block and the other goes to the *else* block. A while statement (and other loop statements) also splits a sequence. Both first visit the conditional expression and one goes into the loop body but the other skips the whole loop body. Loop iteration is ignored. A try-finally statement is treated as a normal sequence of statements; it goes into a try block first and then a finally block. It never visits a catch clause included in that try statement. The compiler constructs an independent join-point sequence for that catch clause. It is separately tested to find a matching region. A return 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 rc2 shown in the previous subsection 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 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 2, a regioncut rc1 selects the region from a() (line 9) to b() (line 11) 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.

The 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 curry brackets, the resulting method body is still syntactically valid. In Figure 2, 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 11).

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. 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. Then, for each direct child node of the root of that sub-tree 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.

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. If a locked object is in an argument 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

```
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. Context exposure by a regioncut

region is passed to an advice body as an argument. Otherwise, if the variable is not available at that point, or if the argument to args or target is a compound expression, then a compile error is reported.

Figure 3 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 [21]. We use 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. To perform the region expansion described in Section 3.2.2, we extended Jimple to make this information available.

We introduced a new Jimple instruction *marker*. A marker has two properties. One is a kind of the structure, which is either statement, block, if, while, or others. The other is whether the marker represents 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. 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 [18], for example, one selected by execution and initialization pointcuts. The values of method parameters and so on at the join point shadow are passed as arguments to the static method. We 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

```
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 4. 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 5. A transformed version of the method in Figure 4

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.

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 4, 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 5. 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.

```
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 6. 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 7. A transformed version of the method in Figure 6

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 6 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 end of the region.

Furthermore, at the end of the region, a switch statement is inserted. It branches to the destination specified by the jump id. variable. Figure 7 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 7. 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. The compiler can implement them if one of the selected regions is nestedly contained in the other region. We will define a precedence rule for intersecting regions so that the compiler can allow around advices for such regions. For example, if a region with higher precedence is expanded to contain the other intersecting regions at the phase described in Section 3.2.2, the compiler can implement around advices for those regions. However, the practicality of this rule is not clear yet.

Regioncuts are more fragile than ordinary pointcuts when the code is refactored. To make regioncuts more robust, we plan to extend our compiler to perform inter-procedural analysis when constructing a join-point sequence at the phase described in Section 3.2.1. This will make regioncuts less fragile for *extract method* refactoring. However, regincuts will be still fragile for other kinds of refactoring. We believe that using the assertion shown in the next section is more pragmatic for reducing the fragility of regioncuts.

4. An Assertion for Advice

To solve the problems mentioned in Section 2, we also propose a new assertion mechanism for AspectJ. This 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. 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. The former annotates a method and the latter annotates an advice. Figure 8 and Figure 9 show examples. @AssertAdvised declares that the behavior of the annotated method, for example, the foo method in Figure 8, must be modified for implementing some concern by an advice annotated by @SolveProblem, for example, the advice in Figure 9. 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.

During runtime, @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), 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).

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

Figure 8. An annotation for a method

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

Figure 9. An annotation for an advice

Here the @SolveProblem advice is an advice with the @SolveProblem annotation corresponding to the @AssertAdvised of the method. If the test fails, then java.lang.AssertionError 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 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.

4.2 Implementation

The 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 8 and Figure 9 is transformed into Figure 10.

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.

```
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 10. The implementation of the annotations for advices

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 11 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").

5. Preliminary Evaluation

We implemented regioncut and assertions for advices by extending the AspectBench 1.3.0 compiler with the JastAdd frontend [12, 9] running on Sun JVM 1.6. Then we evaluated the design of the

```
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 /* an aspect for synchronization */
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 11. An example of false alert by @AssertAdvised

regioncut and the assertion for advice by applying them to two open-source software products.

5.1 Javassist

We wrote synchronization aspects for Javassist by using the proposed constructs along the scenario in Section 2. One aspect implements fine-grained synchronization. Figure 12 shows an equivalent program in which fine-grained synchronization code is embedded by hand. Two synchronized statements are embedded in the createClass2 method. Figure 13 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 14 and Figure 15 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 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.

```
1  private static WeakHashMap proxyCache;
2  private void createClass2(ClassLoader cl) {
3    CacheKey key = new CacheKey(...);
4    synchronized (proxyCache) {
5      HashMap cacheForTheLoader=proxyCache.get(cl);
6      if(cacheForTheLoader==null){
7        cacheForTheLoader = new HashMap();
8        proxyCache.put(cl, cacheForTheLoader);
9        cacheForTheLoader.put(key, key);
10     }else{
11       ...
12     }
13   }
14
15   synchronized (key) {
16     Class c = isValidEntry(key);
17     if(c == null){
18       createClass3(cl);
19       key.proxyClass= new WeakReference(thisClass);
20     }else{
21       thisClass = c;
22     }
23   }
24 }
```

Figure 12. 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    return thisClass;
11  }
```

Figure 13. Coarse-grained Synchronization by hand

	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 1. The execution time of the Javassist benchmark

5.2 Hadoop

Hadoop [20] is an open-source framework for distributed computing; it provides a distributed file system and programming supports

```
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): region[
13   call(* *.isValidEntry(*)) && args(key),
14   set(* *.proxyClass)]
15 {
16   synchronized(key){
17     proceed(key);
18   }
19 }
```

Figure 14. 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 15. The advice for coarse-grained synchronization

for the MapReduce computing model [8]. We rewrote the program of Hadoop 0.16.4 in AspectJ with our proposed constructs.

Separating Synchronization concerns by Regioncut First, we separated synchronization concerns into aspects from the TaskTracker [1] class (2357 LOC) of Hadoop. Table 2 lists the result of our experiment. The TaskTracker class contains 21 synchronized statements. We separated all the statements into aspects. 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. 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.

Assertion for Advices To evaluate the assertion for advice, we added @AssertAdvised and @SolveProblem annotations for the

[1] org.apache.hadoop.mapred.TaskTracker

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

Table 2. The number of synchronization concerns in the TaskTracker class

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

Table 3. The number of synchronization advices with regioncuts after the update to Hadoop 0.18.3

	Time (ms)	**Std. Deviation**
with assertion	24.7	0.1
without assertion	1321.4	34.8

Table 4. The performance comparison of assertion for advice

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 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.

5.3 Performance of Assertion for Advice

We measured the performance of assertion for advice. Figure 16 is a micro 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 this micro benchmark parogram with and without assertion for advice. Table 4 lists the result. This result shows that, in this experiment, the execution performance of the program with the assertion for advice is more than 50 times slower than one without the assertion (by the -da option of java command, which disables the assertion mechanism of Java).

6. Related Work

Declarative event patterns [22] 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. However, these history-based pointcuts are not appropriate for implementing a synchronization concern. To synchronize program execution, a lock must be acquired by a synchronized statement or a java.util.concurrent.Lock at the beginning of the execution history the pointcut matches on. It is extremely difficult to do this at the time when the first event of the history just occurs since the pointcut refers to *future* events (not occurring yet at that time) as in GAMMA[16]. A drawback of the history-based point-

```
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):call(void AATest.inc())
           && target(self){
25         proceed(self);
26         self.counter++;
27     }
28 }
```

Figure 16. The benchmark program for assertion for advice

cuts is that they only support before and around advices invoked at the last join point in the matching execution history. They do not support before or around advices invoked at the first join point in the execution history, which are necessary for synchronization.

LoopsAJ [11] 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. If a method body contains multiple loops, LoopsAJ cannot distinguish these loops. On the other hand, our regioncuts can distinguish them.

Xi et al. proposed *synchronized block join points* [23, 24]. The contribution of their work is to enable selecting synchronized statements as join points. With their work, programmers can select existing synchronized statements, for example, to change synchronization policies but they cannot change the granularity of synchronization or add new synchronization code within a method body including no synchronized statement. With regioncuts, programmers can implement various kinds of granularity of synchronization.

Cacho et al. proposed *EJFlow* [6]. EJFlow provides AOP constructs for exception handling. It allows programmers to select the flow of an exception. Since regioncuts enable programmers to select an arbitrary code region, they can also separate exception-handling concerns as well as synchronization concerns.

FlexSync [25] is an aspect-oriented synchronization library. It helps switching synchronization mechanisms: lock, block-level atomicity, and software transactional memory, without modifying the base program. FlexSync requires that a synchronization block is a method body since it uses an ordinary pointcut language. If the base program does not satisfy this requirement, the programmer must transform the synchronization block into a method by using a refactoring tool or by hand. On the other hand, our regioncut is an extension to the pointcut language of AspectJ. It was designed to be able to identify an arbitrary synchronization block, which may not be a method body.

7. 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. 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. We implemented these two constructs by modifying the AspectBench compiler.

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 during unit tests.

References

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

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM.

[3] S. Apel and D. Batory. When to use features and aspects?: a case study. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 59–68, New York, NY, USA, 2006. ACM.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. 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.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[6] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. Ejflow: taming exceptional control flows in aspect-oriented programming. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 72–83, New York, NY, USA, 2008. ACM.

[7] S. Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, London, UK, 2000. Springer-Verlag.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 884–885, New York, NY, USA, 2007. ACM.

[10] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

[11] B. Harbulot and J. 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.

[12] G. Hedin and E. Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical

report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[14] K. C. Kang, S. Kim, J. Lee, K. Kim, G. J. Kim, and E. Shin. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[16] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *FOAL 2005: Foundations of Aspect-Oriented Languages 2005*. ACM, 2005.

[17] K. Kourai, H. Hibino, and S. Chiba. Aspect-oriented application-level scheduling for j2ee servers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 1–13, New York, NY, USA, 2007. ACM.

[18] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *FOAL 2002: Foundations Of Aspect-Oriented Languages - Workshop at AOSD 2002* -. ACM, 2002.

[19] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[20] The Apache Software Foundation. Welcome to apache hadoop! `http://hadoop.apache.org/`.

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[22] R. J. Walker and K. 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.

[23] C. Xi, B. Harbulot, and J. R. Gurd. A synchronized block join point for aspectj. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 39–39, New York, NY, USA, 2008. ACM.

[24] C. Xi, B. Harbulot, and J. R. Gurd. Aspect-oriented support for synchronization in parallel computing. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution*, pages 1–5, New York, NY, USA, 2009. ACM.

[25] C. 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.