Region Pointcut for AspectJ

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

Shigeru Chiba
Tokyo Institute of Technology
chiba@is.titech.ac.jp

Muga Nishizawa
Rakuten, Inc.
muga.nishizawa@mail.rakuten.co.jp

ABSTRACT

This paper proposes a new pointcut called *region pointcut*, which has the ability to pick out regions as join points. This pointcut allows programmers to modularize synchronization, exception handling and parallelization. Although these are typical crosscutting concerns, the join point that we need to advise for such a concern is not a *point* but a region. The existing aspect-oriented programming language, including AspectJ, provide only the means to pick out points. We modified the AspectBench compiler and implemented region pointcuts as an extension to the AspectJ language.

Categories and Subject Descriptors

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

General Terms

Languages, Design

1. INTRODUCTION

Aspect-oriented programming (AOP) helps programmers separate and modularize crosscutting concerns such as logging and Observer pattern. Although synchronization, exception handling, and parallelization should be also crosscutting concerns that AOP could help to modularize, doing so is difficult in existing AOP languages including AspectJ [5]. These AspectJ-like languages provide pointcut primitives that pick out a point in execution time, for example, when a method is called. Advices are invoked at that "join" point. However, synchronization, exception-handling, and parallelization concerns need advices that are invoked when an arbitrary code region specified by the programmer is executed. The existing languages do not allow programmers to pick out the period in time while an arbitrary code region is executed.

To address this limitation, this paper proposes a new kind of pointcut, which we call *region pointcut*. Its argument is a *region-match pattern*, which is used to select an arbitrary code region. For example, a region-match pattern can match the code region between one method call and another method call. A region-match pattern is conscious of the block structure of the target program. If it matches a region that intersects the existing block hierarchy of the program, the region is extended to fit the block hierarchy.

The region pointcut picks out a join point that is the code region selected by the region-match pattern. An around advice for this join point is executed instead of the original code region. Thus it can implement, for example, a synchronization concern; it can lock an object by a synchronized statement and proceed. Since the join points that the region pointcut picks out are not really points but regions, the region pointcut extends the join point model of the language. To capture the runtime contexts necessary for implementing a synchronization concern, the region pointcut can also obtain the value of a local variable available at the beginning of the region selected by the region pointcut. The region pointcut is implemented as an extension to the Aspect Bench Compiler (abc) [8]. This paper also mentions a few issues of this implementation and the results of experiments using this implementation.

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 the region pointcuts. Section 4 mentions our implementation of the region pointcut. Section 5 describes related work. Section 6 concludes this paper.

2. MOTIVATING EXAMPLE

Synchronization and exception handling as well as transaction and security enforcement are typical non-functional concerns known as good candidates of aspects. For modularizing an exception-handling concern, AspectJ provides an after-throwing advice, which is executed when the corresponding join point (such as a method call) abruptly terminates and throws an exception. It also provides a handler pointcut, which selects a join point when an exception of the specified type is caught by a catch clause of a try statement. However, AspectJ programmers cannot write an advice that is executed when an exception is thrown within the code region specified by the programmers, for example, from a method call to beginTransaction until a call to endTransaction. A handler pointcut deals with an exception thrown only within an existing try block. An after-throwing advice is executed when an exception is thrown anywhere within the whole body of the specified method. If the code region

ACP4IS'09, March 2, 2009, Charlottesville, Virginia, USA.

that a programmer wants to advise is not either a try block or a whole method body, they cannot write an aspect for the exception handling.

This limitation is not only for exception handling. AspectJ does not provide pointcut primitives for selecting an arbitrary region as a join point. The reason for this is that because the design principle of AspectJ is that a join point is a *point* in time ; it is not a *region* in time [7]. Although an exception caught by a handler pointcut comes from the code region of the try block, that pointcut selects the point in time when the exception is thrown. It never selects the region in time while the try block is being executed. This is also true for an after-throwing advice, which selects the point in time when the method completes by throwing an exception.

To overcome this limitation, there have been a few proposals. For synchronization, Xi et al. proposed a synchronized block join point for AspectJ [10], which corresponds to the code block of a synchronized statement. Programmers can select that join point and write an around advice for it. For a loop statement, Harbulot and Gurd proposed a join point representing a loop body [4]. It also corresponds to a code region. However, these proposals do not allow programmers to directly specify a code region that they want to write an aspect for. They have to choose the block of a synchronized statement or the loop body of a for (or while) statement.

Synchronization aspect

To illustrate the need of directly specifying a code region for aspects, we below show an example. In 2006, we received a bug report for Javassist [1]. Javassist [3] 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. Figure 1 shows the method after we modified it to fix the bug. It is part of the javassist.util.proxy.ProxyFactory class. We inserted two synchronized statements into the method (line 4 and 25).

Although the two synchronized statements fixed the bug, it was not clear that this solution is the best with respect to execution performance. As we discussed in our previous paper [6], excessive concurrency often has negative impact on performance. In year 2006, low-end servers were still single-processor machines and 4-way multi-processor 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 work load. 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, for example, by putting the whole method body of createClass2 into a synchronized statement that locks the lock of proxyCache, which is a static field of the ProxyFactory class.

If we could implement this synchronization as an aspect, we could avoid this dilemma of lock granularity. We could write two aspects, one for small granularity and the other for large, and let the users choose either one of the two aspects to fit their execution environment. Javassist could be distributed as synchronization aspects (and maybe other performance tuning aspects) and the jar (Java archive) file containing the rest of the code. The users do not have to directly modify the source code of Javassist to improve the performance, or we do not have to maintain two versions of Javassist, each of which is customized for a different type of machines. However, writing such an aspect is difficult, for example, in AspectJ because we cannot select an arbitrary code region as a join point to write a synchronization advice for that.

```
1  private void createClass2(ClassLoader cl) {
2    CacheKey key = new CacheKey(
3      superClass, interfaces, methodFilter, handler);
4    synchronized (proxyCache) {
5      HashMap cacheForTheLoader =
6          (HashMap)proxyCache.get(cl);
7      if(cacheForTheLoader==null){
8        cacheForTheLoader = new HashMap();
9        proxyCache.put(cl, cacheForTheLoader);
10       cacheForTheLoader.put(key, key);
11     }else{
12       CacheKey found =
13         (CacheKey)cacheForTheLoader.get(key);
14       if(found == null){
15         cacheForTheLoader.put(key, key);
16       }else{
17         key = found;
18         Class c = isValidEntry(key);
19         if(c != null){
20           thisClass = c;
21           return;
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     }else{
31       thisClass = c;
32   } }
33 }
```

Figure 1: The modified method in javassist.util.proxy.ProxyFactory class

Separating a synchronization concern into an aspect has another benefit. It enables the programmers to easily overview when objects are locked for synchronization. Choosing an object that will be locked by synchronized statements (and synchronized methods) is not a simple task because programmers must avoid deadlock. For example, another bug report to Javassist was that two method calls were not surrounded with a synchronized statement and thus the code was not thread-safe. To fix this bug, before inserting a synchronized statement, we had to make sure that the called methods do not lock other objects, or, at the least, the objects they lock do not form a cycle.

3. REGION POINTCUT

In this section, we propose a new pointcut primitive for AspectJ. This pointcut named *region pointcut* addresses the problem mentioned in the previous section.

3.1 Region Adjustment

The region pointcut picks out a join point that is not really an execution point but a region — the period while a specific code region is executed. The region that the region pointcut

picks out can be an arbitrary code block within a method body. The region pointcut specifies the beginning and the end of the code block, which are given by the region-match patterns presented in Section 3.2. The beginning and the end of the code block are (the shadow of [7]) simple join points that represent an execution *point*, for example, ones picked out by the call pointcut.

Region pointcuts support before, after, and around advices. An around advice for the join point that the region pointcut picks out substitutes for the original code block. It is executed instead of that block, which is not executed unless the advice explicitly invokes the original block by proceed.

Region pointcuts are conscious of the block structure of the program. The selected region is implicitly extended to fit the block hierarchy of the program if the originally selected region intersects different levels of blocks. Otherwise, an around advice for such a region would be difficult to use due to its complicated semantics.

For example, in Figure 2, suppose that a region pointcut specifies the call to beginCriticalSection as the beginning of the region and the call to endCriticalSection as the end of it. The beginning of the region is within the body of the for statement while the end of the region is the next statement to that for statement. If the region pointcut naively picked out the region between the two method calls, how does an around advice for that region change the behavior of the method foo? The numbers of calls to beginCriticalSection and endCriticalSection are different. It is difficult to define the behavior of the method foo so that the programmers can naturally expect and understand it.

```
1  public void foo() {
2      ...
3      for(int i=0;i<100;i++){
4          bar();
5          beginCriticalSection(); // begin
6          ...
7      }
8      endCriticalSection(); // end
9      ...
10 }
```

Figure 2: Intersecting the block structure of the program

To avoid such confusion, our region pointcut implicitly extend the region. In the case of Figure 2, the selected code block is extended to include the entire for statement and the method-call expression to endCriticalSection. It starts at the variable declaration:

```
int i = 0;
```

and it ends just after the statement for calling the method endCriticalSection.

A region pointcut selects the code block between the two statements including the join point (shadow) representing the beginning and the end of the region. If the two statements are not at the same level of the block structure, they (or one of them) are superseded by upper-level statements that include those statement and are at the same level. In the example of Figure 2, the statements at line 5 and 8 were initially selected. Since the level of the two statements are different, the statement at line 5 is superseded by the for statement including that method-call statement. The for statement and the other statement at line 8 are at the same level. Thus, the region pointcut selects the code block from line 3 to 8.

3.2 Region-match Patterns

We propose a new pointcut designator called *region pointcut* to pick out regions as join points. The argument of the region pointcut is a *region-match pattern*, which determines the starting and ending join points by pattern matching.

A region-match pattern specifies a possible sequence of events. Here, the events are join points that a simple pointcut such as call can pick out. A region-match pattern specifies the join point where the region starts and it also specifies the last join point where the region ends. It can also specify join points that must be included in the region. If the control flow includes a branch by a statement such as if and while, the pattern matches if there is at least one control path that matches the event sequence specified by the pattern. For example,

```
1  a();
2  if(flag)
3      b();
4  else
5      c();
```

the pattern may match the region that starts when the method a is called and ends when the method b is called (between line 1 and 3) although the region will be extended to cover line 5 as we mentioned in Section 3.1. The value of flag might be false at runtime but this fact is not considered during the pattern matching. Note that this pattern matching considers only the join point shadow and static types. The control flow is also dealt with in the same approach; the pattern matches if the program execution *may* cause a specified sequence of join points.

For example, the following region pointcut:

```
1  pointcut p(): region[call( * *.a()); call( * *.c())];
```

matches the entire method body of foo:

```
1  void foo(){
2      a(); b(); c();
3  }
```

The semicolons delimiting events work as a wild card * because they match any join points such as the call to the method b.

All pointcut

Region-match patterns can include a pseudo-pointcut: all. It is a special pointcut designator available only with a region pointcut.

The all pointcut takes event-sequence patterns as its argument. The argument can be multiple patterns separated by comma. This pointcut matches the shortest join-point sequence that includes at least one sub-sequence matched by every pattern passed as the argument. For example, this region pointcut:

```
1  pointcut p3(): region[all(call( * *.a()), call( * *.c()))];
```

matches the shortest code block that includes the method call expressions to a and c in any order. Furthermore, the code block may include multiple method-call expressions to either a or c (it will include only one expression for calling the other method because the block is the shortest). Hence, the

region pointcut above will match c();c();a();, which includes two expressions for calling the method c.

The all pointcut makes a region pointcut less fragile. The order of two method calls in a code block may change if the code is refactored in the future. The all pointcut enables the same code block to be selected even after the order of method calls within that block changes.

Sequence modifier

An event sequence in the region-match pattern may have a modifier @pointcut.

The @pointcut modifier is used to specify a join-point sequence surrounding the code block that we want to select. See Figure 3. There are two region pointcuts pc1 and pc2. @pointcut is included only in pc2. Although the code block between line 9 and 11 in methodB are selected by pc1 and pc2, the code block between line 2 and 4 in methodA is selected only by pc1. The region pointcut pc2 selects the code block that the pattern passed to @pointcut matches but that code block must follow a method call to x. The join-point sequence that the argument to @pointcut matches must be a sub-sequence of the sequence that the entire region-match pattern matches.

```
1  public void methodA() {
2      a();
3      ...
4      b();
5  }
6  public void methodB(){
7      x();
8      ...
9      a();
10     ...
11     b();
12 }
13
14 pointcut pc1():
15    region[
16      call(* *.a());
17      call(* *.b());
18    ];
19 pointcut pc2():
20     region[
21         call(* *.x());
22         @pointcut[call(* *.a()); call(* *.b());];
23     ];
```

Figure 3: The @pointcut modifier

Context Passing

In Java, a synchronized statement takes an object that will be locked. To implement the synchronization aspect shown in Figure 2, if an around advice executes a synchronized statement, it must take a lock object as a parameter. The region pointcuts provide several means to obtain a lock object.

A region pointcut can be combined with the this or args pointcut by &&. If it is combined with this, the argument of this is bound to the value of the this variable available in the region that the region pointcut picks out. If the region pointcut is combined with args, the arguments of args are bound to the values of the arguments passed to the method containing the region that the region pointcut picks out.

The args and target pointcuts can be used within the region-match pattern, *i.e.* the argument to the region pointcut. Figure 4 shows an example. For example, the pointcut expression at line 17 contains args. The argument to args is bound to the value passed to the method add at line 3 in bar. This behavior is the same as in AspectJ.

```
1  public void bar() {
2      List s = ... ;
3      s.add(Integer.valueOf(1));
4      if(cond()){
5          s.contains(Integer.valueOf(2));
6      }
7      for(int i = 0; i < 100; i++){
8          beginCriticalSection();
9          Object o=s.get(i);
10         endCriticalSection();
11     }
12     ...
13 }
14
15 pointcut pc1(Integer intObject) :
16     region[
17         call(* List+.add(*)) && args(intObject);
18         @pointcut[
19             call(* *.beginCriticalSection());
20             call(* *.endCriticalSection());];
21     ];
```

Figure 4: Region pointcuts with args

4. IMPLEMENTATION ISSUES

We implemented region pointcuts in AspectJ as extension to the AspectBench Compiler (abc) [8]. Abc uses Jimple [9] as an intermediate language. We also use Jimple for pattern matching.

4.1 Analysis of blocks and statements

Jimple has no information where blocks, statements and control structures start and end. In order to achieve the region adjustment described in Section 3.1, we should recognize this information.

We chose the approach that the information is embedded in Jimple instruction sequences. We introduced a new Jimple instruction *marker*. A marker has a type, such as *statement*, *block*, *if*, *while* and so on, and it has information about beginning or ending of the statement. We modified the Java-to-Jimple compiler, to make pairs of markers surrounding the instructions of statements.

After the join points representing the beginning and the end of the region are determined by region matching, we proceed to the region adjustment phase. In this phase, a tree of the Jimple instructions for each method is constructed. The nodes of the tree are block or control structures and the leaves are statements. Then the inserted markers will be removed before Jimple instructions are converted into Java bytecode.

4.2 Around advice support

In abc, some join point shadows, such as execution and initialization join point shadows, are separated and moved into static methods, when weaving an around advice. Figure 5 shows a simplified weaving, where an around advice is applied to an execution join point. Local variables available at join point shadows are passed as arguments to the static methods. We use this mechanism for region pointcuts. However, region pointcuts deal with an arbitrary code block within a method body. This fact causes several issues when around advices are woven into regions.

Before Weaving

```
1  public void toBeAdvised(int x){
2      System.out.println("before␣a()");
3      a(x);
4      System.out.println("after␣a()");}
5
6  void around():execution(* *.toBeAdvised()){
7      System.out.println("before␣proceed()");
8      proceed();
9      System.out.println("after␣proceed()");}
```

After Weaving

```
1  public void toBeAdvised(int x){
2      advice(x);}
3  public static void advice(int x){
4      System.out.println("before␣proceed()");
5      shadow(x); //corresponding to proceed();
6      System.out.println("after␣proceed()");}
7  public static shadow(int x){
8      System.out.println("before␣a()");
9      a(x);
10     System.out.println("after␣a()");}
```

Figure 5: Weaving an around advice into an execution join point

4.2.1 Assignments to local variables declared out of the region

When a region (*shadow*) is separated into a static method, assignments to local variables cause trouble. If there is an assignment in the region to a local variable declared outside the region, this assignment dose not correctly update the values of the local variable. The variables outside and inside the region are now different variables because they are in different methods.

To solve this problem, we transform the code of the method. We use an object that contains the values of local variables in its fields. This object is shared between the region and the original method by being passed as an argument between them.

4.2.2 Jumps to the outside of the region

Jumps — break or continue statements — are also adversely affected by the separation of a region. If there are jumps whose targets are outside of the region, their targets will be in another method. Since JVM dose not support such a inter-method jumps, these jumps will fail.

To solve this problem, we assign identification (id) numbers to each jump to the outside. Next, each jump is replaced by the following instructions:

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

After the execution of the region, the jump id variable is checked, and then the thread jumps to the target specified by the jump id variable. The jump id variable will be shared between the region and the original method by the mechanism mentioned in Section 4.2.1.

5. RELATED WORK

LoopsAJ [4] provides a pointcut for picking out a loop join point, which corresponds to a loop body. It allows parallelizing the execution of the specified loop body. However, if the loop has several successor nodes, LoopsAJ cannot apply an around advice to that loop. Also, when there are multiple loops in the body of a method, LoopsAJ dose not provide a means to select only one loop among those loops. On the other hand, our region pointcut enables programmers to select it.

Tracematches [2] provides a history-based pointcut mechanism. For tracematches, *regular patterns* are used to describe the join points that programmers want to pick out. An advice is invoked when the history of the runtime events matches the given regular pattern. Although this approach has a limited support for an around advice, our region pointcut supports an around advice because it can statically determines the region that it selects.

6. CONCLUSION

This paper proposed region pointcuts to provide a means to deal with code regions as join points in AspectJ and we implemented region pointcuts by modifying the AspectBench compiler. A region pointcut supports before, after and around advices. This pointcut helps separating and modularizing synchronization, parallelization, and exception handling. A region selected by our region pointcut might be appropriately extended to be consistent with the block hierarchy of the program.

A drawback of our region pointcut is its fragility. Although complex information is often necessary for picking up a region, it increases the dependency of the pointcut on the target program. This makes the pointcut fragile and even simple refactoring of the target program may change the semantics of the pointcut. Reducing this drawback is our future work.

7. 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 et al. Adding trace matching with free variables to aspectj. In *OOPSLA '05*, pages 345–364. ACM, 2005.

[3] S. Chiba. Load-time structural reflection in java. In *ECOOP '00*, pages 313–336. Springer-Verlag, 2000.

[4] B. Harbulot and J. R. Gurd. A join point for loops in aspectj. In *AOSD '06*, pages 63–74. ACM, 2006.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01*, pages 327–353. Springer-Verlag, 2001.

[6] K. Kourai, H. Hibino, and S. Chiba. Aspect-oriented application-level scheduling for j2ee servers. In *AOSD '07*, pages 1–13. ACM, 2007.

[7] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In *APLAS*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2006.

[8] P. Avgustinov et al. abc: an extensible aspectj compiler. In *AOSD '05*, pages 87–98. ACM, 2005.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99*, page 13. IBM Press, 1999.

[10] C. Xi, B. Harbulot, and J. R. Gurd. A synchronized block join point for aspectj. In *FOAL '08*, page 39. ACM, 2008.