遠隔ポイントカット
– 分散アスペクト指向プログラミングのための言語機構

Remote Pointcut
– A Language Construct for Distributed Aspect-Oriented Programming

by

西澤 無我

Muga Nishizawa

02M37259

January 2004

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 Information Science and Engineering.

Supervisor: Shigeru Chiba

Abstract

This paper presents our extensions to the AspectJ language that enables to separate crosscutting concerns in distributed systems. Aspect-Oriented Programming (AOP) is technology that can capture concerns that cut across multiple modules, called crosscutting concerns, such as logging, synchronization and security. AOP allows developers to modularize a crosscutting concern as an aspect, to develop maintainable and understandable systems.

This paper shows that some crosscutting concerns in distributed systems cannot be modularized in existing AOP languages as simple aspects. Rather, aspects modularizing such a concern tend to be in code spread over multiple hosts and explicitly communicate across the network. Here, the simple aspect means maintainable and readable aspect. This paper illustrates this fact with an example of testing a distributed program in AspectJ with Java RMI. AspectJ is an implementation of AOP for Java.

To address this problem of the complexity due to network communication, this paper proposes an AOP language called DJcutter, which is the extension to AspectJ for distributed computing. The most significant difference between DJcutter and existing AOP languages is that DJcutter provides a new language construct named remote pointcuts. Pointcut is a language construct provided by AspectJ, and it has a function to pick out the program's operations, called join points, such as method calls, field accesses, and exception events in the program flow. However, the pointcut language provided by AspectJ is only for identifying join points on a single host, and it doesn't have functions enough to implement some crosscutting concerns in distributed systems. Remote pointcuts provided by DJcutter allow developers to designate join points on remote host. Thereby, DJcutter enables to write a simple aspect that can modularize crosscutting concerns distributed on multiple hosts. Moreover, to examine the execution performance of remote pointcuts, we compared the execution time of the programs in DJcutter and AspectJ using Java RMI.

Acknowledgments

I profoundly thank my supervisor, Professor Shigeru Chiba at Tokyo Institute of Technology. His suggesions and supports greatly helped me write this work. I am also thankful to Dr. Kenichi Kourai at Tokyo Institute of Technology, Dr. Michiaki Tatsubori at IBM Tokyo Research Laboratory, and Yoshiki Sato at Tokyo Institute of Technology, who gave me great advice and comments. I also thank my colleagues in our research group at Tokyo Institute of Technology, in particular Chikayasu Uzaki, Kiyoshi Nakagawa, and Ryo Kurita. I am greateful for english support by Shannon Jacobs at IBM Tokyo Research Laboratory and Emi Seto at University of Nihon.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modularizing software systems is one of significant demands in software industry. Developers should decompose software into a number of small independent programs so that they can develop software of high quality, which is easy to understand, modify, and maintain [21]. To do that, several paradigms have been proposed. One of them is *Object-Oriented Programming* (OOP) technology. It is a widely used paradigm for developing software systems because the object modeling, which is a central concept of this paradigm, provides a better fit with real domain problems. Therefore, a large number of OOP based systems, languages, and applications have been developed so far.

Recently, to separate *crosscutting concerns* in software, *Aspect-Oriented Programming* (AOP) have been proposed [6, 16, 36, 27]. Crosscutting concerns, such as logging, cut across the boundary between modules, and they decrease maintainability and understandability of software. Whereas OOP is not powerful technology enough to separate such concerns, AOP is technology that can modularize crosscutting concerns as *aspects*. AOP allows developing maintainable and understandable systems.

Motivating Problem

Many crosscutting concerns, such as logging, transaction, and security, arise in distributed software systems. Some of them cannot be modularized in exising AOP languages as simple aspects. Rather, aspects modularizing such concerns tend to be in code spread over multiple hosts and explicitly communicate across the network. Here, the simple aspect means maintainable and readable aspect. This paper illustrates this fact with an example of testing a distributed program in AspectJ with Java RMI. AspectJ is an

implementation of AOP for Java [15].

The source of these problems is that existing AOP languages do not accommodate to distribution or network processing. Combination of the AOP language and an existing framework for distributed software, such as Java RMI (remote method invocation)[35] is not a solution. The existing frameworks extend language constructs for object-orientation, such as method calls, so that they can accommodate distribution. They do not support language constructs for aspect orientation.

Solution by This Thesis

To address this problem of the complexity due to network communication, this paper proposes a new AspectJ-like language named *DJcutter* . In this language, several language constructs, in particular pointcuts, have been extended for distributed software. The extended pointcuts of DJcutter are called *remote pointcuts*. Although a pointcut in AspectJ identifies execution points on the local host, a remote pointcut can identify them on a remote host. This language construct can simplify the code of a component implementing a crosscutting concern in distributed software. DJcutter also provides another language construct named *remote inter-type declaration*, which allows developers to declare a new method and field in a class on a remote host. The aspect weaving in DJcutter is performed at load time on each participating host. When a class is loaded from a local file system, it is transformed according to an aspect sent from a remote host. This architecture is useful for distributed unit testing since the users do not have to deploy a woven program to each host whenever they change the description of the aspects.

The Structure of This Thesis

From the next chapter, we present background, language specifications, and implementation issues of DJcutter. The structure of the rest of this thesis is as follows:

Chapter 2: Distributed AOP

At first, we explain motivation and contribution of AOP that is the underlying technology of our research. In particular, we explain the detail of the AspectJ language. Next, we present motivating problem and example of this work. At last, we present existing languages, tools, and systems for modularizing crosscutting concerns in distributed software as related work.

Chapter 3: Language Specification

To address the motivating problem, we propose a new AspectJ-like language named DJcutter and explain the language specification. Moreover, we show several examples of apects in DJcutter.

Chapter 4: Implementation Issues

We present implementation issues of DJcutter. The DJcutter implementation mainly consists of two parts: the one is a compiler, the other is runtime infrastructure. We explain the implementation issues of each part.

Chapter 5: Experiment

At first, to examine the execution performance of remote pointcuts, we compared the execution time between DJcutter and AspectJ using Java RMI. Next, we show that an aspect in DJcutter is easy to understand and maintain.

Chapter 6: Concluding Remarks

We conclude this thesis in chapter 6. Moreover, we present future work.

Chapter 2

Distributed AOP

Recently, to modularize concerns that are scattered throughout modules, called *crosscutting concerns*, *Aspect-Oriented Programming* (AOP) have been proposed [6, 16, 36, 27]. Whereas Object-Oriented Programming (OOP) is not powerful technology enough to modularize such concerns, AOP is technology that can separate the crosscutting concerns as *aspects*. It builds on and complements OOP. Users can develop maintainable and understandable systems involving such aspects.

We challenged to modularize crosscutting concerns in distributed systems. To modularize distributed software, a large number of tools, languages and middleware have been proposed and developed. For example, CORBA (Common Object Request Broker Architecture)[24] and the Java RMI (Remote Method Invocation) framework [35] are provided as part of the standard in Java. They allow developers of distributed systems to simplify stub-code generation. Thereby, the developers don't need to use raw sockets for network processing. However, most of them do not support for modularizing crosscutting concerns in distributed software systems. Moreover, existing AOP based tools, languages, and middleware don't have sufficient functions enough to modularize crosscutting concerns in distributed systems yet.

In the rest of this chapter, at first we explain an AOP that is the underlying technology of our research. Moreover, we simply present the AspectJ language specification. Next, we present motivating problem and example of this work. At last, we explain existing tools, languages and middleware for modularizing crosscutting concerns in distributed software as related work.

2.1 Aspect-Oriented Programming

2.1.1 Overview

Aspect-Oriented Programming (AOP) has been proposed as technology for improving separation of concerns in software. However Object-Oriented Programming (OOP) is the technology that can fundamentally modularize software systems, it is not sufficient technology enough to separate concerns that are scattered throughout modules. We call such concerns *crosscutting concerns*. They decrease maintainability and understandability of software systems. Because the crosscutting implementation is scattered throughout modules, the developers must understand and change each modules in software even if they change and maintain the crosscutting implementation. For example, code for logging is scattered within code whose primary responsibility is something else. Even if the developers change and remove the codes for logging, they must change and remove every scattered codes for logging. AOP can pull the widespread crosscutting concern into a single module. These modules are termed *aspects*. AOP builds on several technologies, such as procedural programming and OOP, that have already made significant improvements in software modularity.

Several general-purpose AOP languages, tools, and systems have been developed already. AspectJ [15] is a simple and practical AO extension to Java. Adaptive Programming [20] provides a special-purpose language, called DemeterJ [23], for writing class structure traversal specifications. DemeterJ prevents knowledge of the complete class structure from becoming tangled throughout the code. Composition filters object model [6] provides control over messages received and sent by an object. The composition filters mechanism provides an aspect language that can beused to control a number of aspects inlcuding synchronization and communication. ComposeJ [39] is an extension of the Java language that adds composition filters to Java classes through inlining. Multi-dimensional separation of concerns (Subject-Oriented Programming) [36] provides for composing and integration disparate class hierarchies, each of which might represent different concerns. Hyper/J [14] supports separation and integration of concerns along multi-dimensional in standard Java software. JAC [29] is a Java framework for dynamic AOP. Unlike other lanugages such as AspectJ, JAC does not require any langugae extensions to Java.

2.1.2 AspectJ

AspectJ is an implementation of AOP for Java [15, 11, 28] [1]. But concerns, such as logging and security, cut across the classes in Java, the crosscutting concerns are not easily turned into classes precisely. AspectJ allows developers to implement this concerns in Java.

AspectJ adds to Java just one new concept, a *join point* – and that's really just a name for an existing Java concept. It adds to Java only a few new constructs: *pointcuts*, *advice*, *inter-type declarations* and *aspects*. Pointcuts and advice dynamically affect a thread of control, inter-type declarations statically affects a program's class heirarchy, and aspects encapsulate these new constructs.

Join Points

A *join point* is a program's operation in the program flow. As an example of operations, there are method calls, method executions, field accesses, constructor calls, constructor executions, and exception events in the program flow.

Pointcuts

Pointcuts pick out certain join points in the program flow. For example, the pointcut:

```
call(void Point.setX(int))
```

picks out join points that is a method call of the signature void Point.setX(int) in the program flow. The call is one of pointcut designators, identifies each join points that are call of the specified method. In table 2.1, we listed several pointcut designators that AspectJ is provided. Also, a pointcut can be built out of other pointcuts with and (&&), or (||), and not (!). For example, the pointcut:

```
call(void Point.setX(int)) || call(void Point.setY(int))
```

picks out join points that is method call of the signature void Point.setX(int) or void Point.setY(int).

[1] In this paper, we call AspectJ 1.0.6 "AspectJ". Currently, AspectJ 1.1.1 have been released.

Table 2.1: Several pointcut designators of AspectJ

designator	join points
within(*TypePattern)*	the join points included in the declaration of the types matching *TypePattern*
target(*Type or Id*)	the join points where the target object is an instance of *Type* or the type of *Id*
this(*Types or Ids*)	the join points when the currently executing object is an instance of *Type* or *Id*'s type
args(*Types or Ids*)	the join points where the arguments are instances of *Types* or the types of of the *Ids*
call(*Signature*)	the calls to the methods matching *Signature*
execution(*Signature*)	the execution of the methods matching *Signature*
cflow(*Pointcut*)	all join points that occur between the entry and exit of each join point specified by *Pointcut*

Advice

Advice is a method-like mechanizm used to declare that certain code should execute at each join point in a pointcut. Advice consists of two parts: the one is a pointcut, the other is the code. AspectJ has *before*, *after* and *around* advice. *Before* advice runs before the thread of control reaches each join point in a pointcut. *After* advice runs after the thread of control reaches identified join points.

Aspects

Aspects are modular units of crosscutting implementation, wrap up pointcuts and advice. It is defined like a class, can have member methods and fields. As an example, the aspect:

```
aspect LoggingAspect {
   pointcut move():
      call(void Point.setX(int)
        || void Point.setY(int));
   before(): move() {
      System.out.println("about to move");
   }
}
```

prints a message whenever the setX() and setY() method in the Point class are called.

Inter-type Declarations

Inter-type declarations (formerly called the *introduction*) are declarations that cut across classes and their hierarchies in AspectJ. We can declare those in an aspect.

2.2 Motivating Problem

2.2.1 Complicated Network Processing

Several AOP languages such as AspectJ are useful programming languages for developing distributed software. They enable modular implementation even if some crosscutting concerns are included in the implementation. However, developers of distributed software must consider the deployment of the executable code. Even if some concerns can be implemented as an aspect at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub-components or sub-processes running on each host. Since Java (or most of the AOP languages) does not provide variables or fields that can be shared among multiple hosts, the implementation of such a concern would include complicated network processing for exchanging data among the sub-components.

Programming frameworks such as Java RMI (remote method invocation) do not solve this problem of complication. Although they make details of network processing implicit and transparent from the programmers' viewpoint, the programmers still must consider distribution and they are forced to implement the concern as a collection of several distributed subcomponents exchanging data through remote method calls. The programmers cannot implement such a concern as a simple, non-distributed monolithic component without concerns about network processing. This is never desirable with respect to aspect orientation since it means that the programmers must be concerned about distribution when implementing a different concern.

2.2.2 Example

A program written in existing AOP languages with distributed middelware often includes complicated network processing in the description of some

aspects. As an example to address this problem, we explains testing code written in AspectJ with Java RMI in this section.

We illustrate this situation with an example of unit testing[2] for distributed software. The importance of testing frameworks is becoming widely accepted. Writing test code for automating unit tests is an important development process that the XP (Extreme Programming) community [5] recommends. The automation results in cleaner code, encourages refactoring, and makes rapid development possible. Recently, simple regression test frameworks such as JUnit [25] and Cactus [1] have been getting popular for the automated unit testing.

Many crosscutting concerns arise during unit testing of software systems. Testing code for non-distributed software includes typical crosscutting concerns that AspectJ can deal with [18]. However, if we use AspectJ to modularize the testing code for distributed software, the code develops the comlexities mentioned above.

Unit Test for Authentication Service

As an example, we present test code for a distributed authentication service. The implementation of this service consists of two components: a front-end server AuthServer on a host W and a database server DbServer on another host D. This is a typical architecture for enterprise Web application systems. If a client application needs to register a new user, it remotely calls registerUser() on the front-end server using Java RMI. Then the registerUser() method remotely calls addUser() on the database server, which will actually access the database system to update the user list.

To unit-test the registerUser() method, the test code would first remotely call the registerUser() method and then confirm that the addUser() method is actually executed by the database server. Note that since the test code must confirm that remote method invocation is correctly executed, it must confirm not only that registerUser() on the host W calls addUser() but also that addUser() starts running on the host D after the call.

The test code would be simple and straightforward if the examined program is not distributed. We below show the test code written in AspectJ (figure 2.1). Although this is not complete code due to the space limitations, the readers would understand the overall structure of the test code. The main part of the test code is testRegisterUser() (lines 3 to 9). It calls the registerUser() method and then confirms the wasAddUserCalled field is

[2]Some might think this example should be called not unit testing but end-to-end testing.

```
 1: aspect AuthServerTest extends TestCase {
 2:   boolean wasAddUserCalled;
 3:   void testRegisterUser() {
 4:     wasAddUserCalled = false;
 5:     String userId = "muga", password = "xxx";
 6:     AuthServer auth = new AuthServer();
 7:     auth.registerUser(userId, password);
 8:     assertTrue(wasAddUserCalled);
 9:   }
10:   before():
11:     execution(void DbServer.addUser(String,
12:                                     String)) {
13:     wasAddUserCalled = true;
14:   }
15: }
```

Figure 2.1: The test code for non-distributed software in AspectJ

true. This field is set to true by the before advice (lines 10 to 14) when the addUser() method is executed.

Test Code in AspectJ

Unfortunately, the test code becomes more complicated if the examined program is distributed. The test code shown below is a distributed version (again, it is not complete code. Access modifiers such as public and constructors are not shown).

```
 1: // on host T
 2: class AuthServerTest extends TestCase {
 3:   boolean wasAddUserCalled;
 4:   void testRegisterUser() {
 5:     Naming.rebind("test", new RecieverImpl());
 6:     wasAddUserCalled = false;
 7:     String userId = "muga", password = "xxx";
 8:     AuthServer auth
 9:       = (AuthServer) Naming.lookup("auth");
10:     auth.registerUser(userId, password);
11:     assertTrue(wasAddUserCalled);
12:   }
```

```
13:   class ReceiverImpl
14:       extends UnicastRemoteObject
15:       implements NotificationReceiver {
16:     void confirmCall() {
17:       wasAddUserCalled = true;
18:     }
19:   }
20: }
21:
22: interface NotificationReceiver
23: { // on both hosts
24:   void confirmCall();
25: }
26:
27: aspect Notification { // on host D
28:   before():
29:     execution(void DbServer.addUser(String,
30:                                     String)) {
31:       NotificationReceiver test
32:         = (NotificationReceiver)
33:             Naming.lookup("test");
34:       test.confirmCall();
35:   }
36: }
```

The test code now consists of three sub-components: AuthServerTest, ReceiverImpl, and Notification (Figure 2.2). Although the overall structure is the same, the AuthServerTest and ReceiverImpl objects run on a testing host T but the Notification aspect runs on the host D, where the DbServer is running. The host T is different from W or D.

The testRegisterUser() method (lines 4 to 12) on T remotely calls registerUser() on W and then confirms that the wasAddUserCalled field is true. This field is set to true by the confirmCall() method in ReceiverImpl, which is remotely called by the before advice (lines 28 to 35) of Notification running on D. The confirmCall() method cannot be defined in AuthServerTest since AuthServerTest must extend the TestCase class whereas Java RMI requires that remotely-accessible classes extends the UnicastRemoteObject class.[3]

[3]This is not precisely accurate. Technically, a confirmCall() can be defined in AuthServerTest by using certain programming tricks. However, the test code would be significantly more complicated.

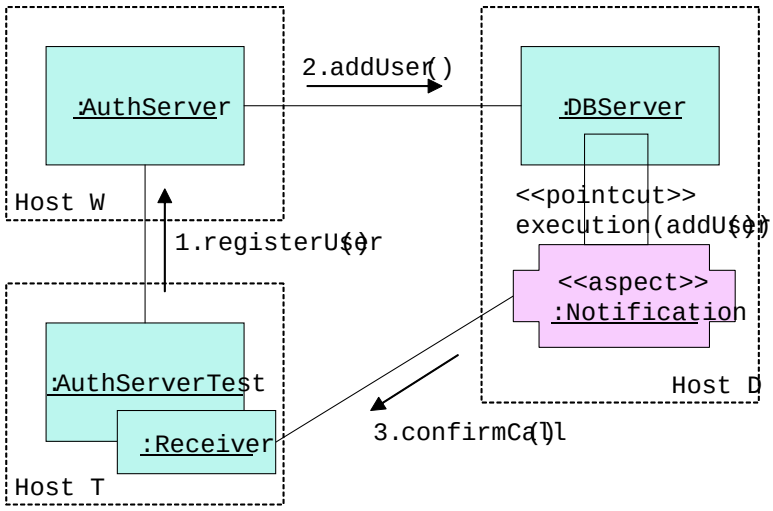

Figure 2.2: The testing code in AspectJ

As we can see, even this simple testing concern is implemented by distributed sub-components and hence we had to write complicated network processing code using Java RMI despite that it is not related to the testing concern. In particular, the Notification aspect is used only for notifying confirmCall() on the host T beyond the network that the thread of control on the host D reaches addUser(). The Notification aspect is a sub-component that are necessary only because confirmCall() and addUser() are deployed on different hosts. This means that the component design of the unit testing is influenced by concerns about distributed. Furthermore, this notification code is similar to what the AspectJ compiler produces for implementing the pointcut-advice framework. It should not be hand-coded, but implicit within the language constructs provided by an AOP language.

2.3 Related Work

Soares et al reported that they could use AspectJ for improving the modularity of their program written using Java RMI [32]. Without AspectJ, the program must include the code following the programming conventions required by the Java RMI. AspectJ allows separation of that code from the rest into a distribution aspect. However, the ability of AspectJ is limited with respect to modularization for distributed programs and thus the resulting

programs are often complicated and difficult to maintain. To address these complications, we propose remote pointcuts and the inter-type declaration as extended language constructs for distributed aspect-oriented programs.

Although Java RMI is the standard framework, several researchers have been proposing other systems such as cJVM [2], our own Addistant [37] and J-Orchestra [38]. These systems provide a single virtual machine image on several hosts connected through a network. They allow for the distributed execution of a program originally written as a non-distributed one, without code modification for the distribution. An alternative to the approach presented here might be to write a program in AspectJ and run it on these systems, which would appropriately translate local pointcuts into remote pointcuts at the implementation level. We did not take this approach since our target applications are for the unit testing of enterprise server software, and these programs are inherently designed and implemented as distributed software. Therefore, we do not have to translate such software to distributed software by, for example, using Addistant, except for the modules implemented as aspects. If we translate all the modules of such software, the unnecessary indirections due to the proxy objects would cause significant performance penalties, since such software has already included indirections for remote accesses. On the other hand, DJcutter can be regarded as a system that translates only aspects to enable transparent remote accesses. Although Addistant allows the programmers to specify translation only for the classes generated by the AspectJ compiler from the aspects, the programmers must manually describe these specifications. DJcutter provides better syntax so that these specifications can be simple or implicit within the language constructs.

Distribution is a well known crosscutting concern and several systems have been proposed to support such concerns. For example, the D language [22] allows the programmers to separately describe how a parameter is passed to a remote procedure. Such work has explores new crosscutting concerns in distributed programs whereas our work explore general-purpose language constructs for distributed aspect-oriented programs. The goal of our work is to develop language constructs so that programs written in an AspectJ-like language can be simple and easy to maintain.

JAC [29, 26] is a powerful framework for dynamic AOP in Java. Unlike other lanugages such as AspectJ, JAC does not require any langugae extensions to Java. An aspect of JAC is implemented by a set of aspect objects. JAC also supports Java API that easily implements crosscutting concerns in distributed systems such as the codes changing consistency protocol on a set of replications and implementing load-balancing for developers. But,

using JAC, even if developers will separate the crosscutting concerns during unit testing, complicated network processing is not necessarily solved. The significant difference JAC and DJcutter is that DJcutter provide the remote pointcut.

DADO (Distributed Adaplets for Distributed Objects) [40] provides a CORBA-like programming model, which comprises several languages, tools, and runtime environment, to support crosscuting concerns in distributed heterogeneous systems. This programming model enables the developers to separate crosscutting implementation that arised in application components on both client and server side such as security, caching. In particular, the DADO programming model has two languages. One of these languages, DADO deployment language, is based on AspectJ and specifies how a QoS feature interacts with an underlying application. However these languages allow modeling the communications between client and server side, don't support remote pointcuts provided by DJcutter.

Chapter 3

DJcutter Language Specification

In this chapter, to address the problems of the previous chaper, We propose an extension to the AspectJ language for distributed software systems. The proposed language allows developers to implement crosscutting concerns as an aspect that does not incude explicit network processing, even if that the concerns cut across multiple components on different hosts. As a result, the developers can develop distributed software of high quality, which is easy to understand, modify, and maintain. The proposed language is called *DJcutter* (**D**istributed **J**oin point **cutter**).

In the rest of this chapter, first, we present core technology of DJcutter, called *remote pointcuts*. Then we explain the specification of runtime infrastructure for realizing the remote pointcut mechanism. At second, we present the DJcutter language specification, such as pointcut designators and aspect methods. In particular, we explain the difference between AspectJ and DJcutter so that DJcutter is an extension to AspectJ.

3.1 Remote Pointcut

The contribution of DJcutter is that it provides *remote pointcuts*. A remote pointcut is a function for identifying join points in the execution of a program running on a remote host. In other words, when the thread of control reaches the join points identified by a remote pointcut, the advice body associated with that remote pointcut is executed on a remote host different from the one where those join points occur. Remote pointcuts are analogous to remote method calls, which invoke the execution of a method body on a remote host.

```
1: aspect LoggingAspect {
2:   pointcut setter(int x):
3:      args(x) && call(void Point.setX(int));
4:   before(int x): setter(x) {
5:      System.out.println("set x: " + x);
6:   }
7: }
```

Figure 3.1: Introductory simple example in DJcutter

Unfortunately, existing AOP languages such as AspectJ do not provide such a pointcut. An advice body in these languages is executed on the same host as where the join points identified by a pointcut occur.

The remote pointcut is a crucial language construct for distributed aspect-oriented programming, corresponding to remote method invocation (RMI) for distributed object-oriented programming. RMI is the technology that can *transparently* access distributed objects same as objects on local host. RMI enables users to describe to distributed software simply and easily. The remote pointcut mechanism is also the technology that *transparently* can identify join points on remote hosts same as join points on local host.

3.1.1 Introductory Simple Example

To explain the remote pointcuts, we show a simple aspect written in DJcutter (figure 3.1), which conforms to the regular AspectJ syntax: This aspect, named LoggingAspect, prints a message whenever the setX() method that is defined in the Point class is called on each participating host.

The pointcut, named setter, (lines 2 to 3) in the aspect:

```
call(void Point.setX(int))
```

identifies each join point that is a call to the setX method in the Point class. Unlike pointcuts provided by AspectJ, however, this pointcut in DJcutter allows identifing each of join points matching the signature on every host even if the advice body is not deployed on the host.

On the other hand, the body of the advice (line 5):

```
System.out.println("set x: " + x);
```

is executed just before each call to the setX(), but it is executed on a host different from the host where the caller thread is running. If the thread

of control reaches the join point, it implicitly sends a message through the network to an *aspect server* running on a different host so that the aspect server will execute the advice body. The detail of aspect server's behavior is explained in following section refsec:spec:aspectserver. As an example, the Point objects are executed on each of hosts A and B. The aspect server is running on a host C. When the thread of control reaches the designated join point, that is a method call setX(), in the program flow on the host A (or B), the before advice is executed by the aspect server on the host C – that is, an argument of method call Point.setX(int) on A (or B) is printed as the valiable x on C. Then, just after the advice is executed, the program flow on A (or B) restarts from the method call setX().

3.1.2 Aspect Server

An *aspect server* is one of DJcutter's runtime library, and it only can execute the body of advice in the library [1]. If the thread of control reaches each of join points that are identified in pointcut, it implicitly sends a message through the network to an aspect server running on a different host[2] so that the aspect server will execute the advice body. The thread of control that sent the message blocks until the aspect server finishes the execution of the advice body. Since all of the advice bodies are executed by the aspect server on the central host, they can easily exchange values by storing data in the fields defined in the aspect. These fields are locally accessible from the advice bodies. Note that, in AspectJ, the advice body is executed on the same host where the caller thread is running. Thus it may have to explicitly send values through the network to exchange them with other advice bodies executing on other hosts.

To explain the behavior of aspect server, we use the example that mentioned above (figure 3.1) again. Figure 3.2 show the behavior of it. When the thread of control reaches a method call Point.setX(int) on A, it sends a message so that the aspect server on C will execute the code of advice (Step 1). Then, the aspect server executes the advice (Step 2). When the aspect server finished executing the code of advice on C, it sends a message so that the thread of control on A will restart (Step 3).

[1] Note that, a *local aspect* (in section 3.6) makes an exception of this specification.
[2] Technically, the aspect server might be running on the same host.

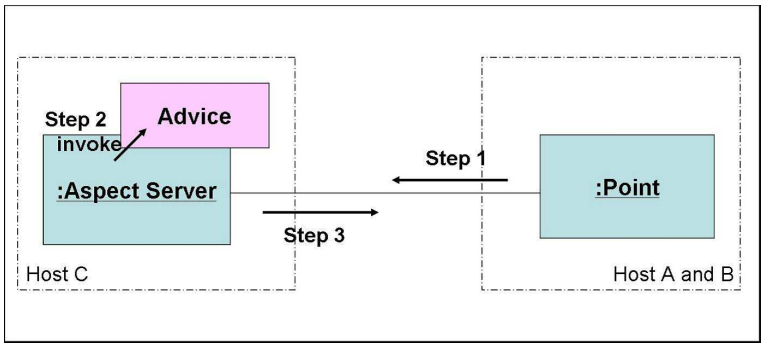

Figure 3.2: The behavior of aspect server

3.1.3 Load-time Weaving

DJcutter performs *load-time weaving*. This meaning is that the normal Java classes and the compiled aspects that constitute distributed software on each participating host are woven (composed) at load-time[3]. The normal Java classes on each host must be loaded by the class loader provided by DJcutter [19]. This class loader weaves the aspects and Java classes on the fly. Both of this class loader and the aspect server are the significant components for implementing the load-time weaving mechanism[4].

The compiled aspects are stored in the aspect server. The parts of the compiled code except for the advice bodies are automatically distributed by the aspect server to each host, so the lastest aspects can be woven when the classes are loaded. The users of DJcutter do not have to manually deploy the compiled aspects to every host.

3.1.4 Remote Inter-type Declaration

This fact improves the usefulness of the inter-type declaration (formerly called *the introduction*) in DJcutter. An aspect can declare that it will respond to certain methods and field-access requests on behalf of other objects. In DJcutter, these methods and fields can be declared other objects on multiple remote hosts. Since the description of the inter-type declaration is automatically distributed from the aspect server to every host, declaring a method or field to classes on remote hosts is simple. The users only

[3]Aspects are compiled by DJcutter compiler in advance. The compiled aspects are normal Java-bytecode.

[4]The detail of weaving mechanism is explained in section 4.3.

Table 3.1: The pointcut designators of DJcutter

designator	join points
within(*TypePattern)*	the join points included in the declaration of the types matching *TypePattern*
target(*Type or Id*)	the join points where the target object is an instance of *Type* or the type of *Id*
args(*Types or Ids*)	the join points where the arguments are instances of *Types* or the types of of the *Ids*
call(*Signature*)	the calls to the methods matching *Signature*
execution(*Signature*)	the execution of the methods matching *Signature*
cflow(*Pointcut*)	all join points that occur between the entry and exit of each join point specified by *Pointcut*
hosts(*Host, ...*)	the join points in the execution on *Hosts*

have to install the compiled aspect on the aspect server. Unlike in AspectJ, they do not have to manually deploy the woven aspect and classes to every host. This automatic deployment is useful in the context of distributed unit testing. We will revisit this issue in the section 3.7.

3.2 Pointcut Designators

The pointcut designators provided by the implementation of DJcutter are listed in 3.1. Most of the pointcut designators are from AspectJ. As an example that mentioned the previous figuare 3.1, the call in AspectJ is designator that can identify each join point that is a call to the specified method as an argument though, one in DJcutter is designator that can identify each of join points matching the speficied signature on every host. In the rest of section, we explain unique pointcut designators that AspectJ doesn't support.

3.2.1 Hosts

A pointcut designator unique to DJcutter is hosts. It identifies the join points in the execution on the designated hosts. Although DJcutter can deal with all the join points on every participating host, this pointcut designator is used to identify the join points on particular hosts.

For example, the users of DJcutter can describe the following pointcut with the hosts pointcut designator:

```
1: pointcut sample(): call(void Point.setX(int))
2:                    && hosts(hostId1, hostId2);
```

This pointcut identifies join points that are calls to the setX() method in the Point class on the hosts with the names specified by hostId1 or hostId2.

These parameters HostId1 and hostId2 are not each of host names, these are parameters given by the users when the program starts running. These runtime parameters allow the developers to avoid embedding particular host names in the source-code. Thereby, the source-code is flexible.

3.2.2 Cflow

DJcutter extends the cflow pointcut designator to handle the control flows of distributed software. Cflow identifies join points that occur between the start of the method specified by cflow and the return. It identifies only the join points visited by the thread executing the method specified by cflow. In AspectJ, cflow cannot pick out join points on a remote host since the control-flow data needed to implement cflow is stored in a ThreadLocal variable but the ThreadLocal variable is never passed through a network.

DJcutter provides a custom socket class sothat the ThreadLocal variable can be passed through a network. If network communication is performed with this custom socket class, then cflow can pick out join points on a remote host. For example, if Java RMI is used for network communication, the following program exports a remote object to make it available to receive incoming calls, using the custom socket class:

```
PointImpl p0 = new PointImpl();
Point p = (Point)
   UnicastRemoteObject.exportObject(p0,
                                    40000,
                                    new DJCClientSocketFacotry(),
                                    new DJCServerSocketFacotry());
```

This program exports a PointImpl object, which is accessible from a remote host through the Point interface. The DJCClientSocketFactory and DJC-ServerSocketFactory classes are the factory classes provided by DJcutter for creating the custom socket. DJcutter also provides a convenient method with which the program shown above can be rewritten as follows:

```
PointImpl p0 = new PointImpl();
Point p = (Point) DJcutter.exportObject(p0, 40000);
```

3.3 Access to Aspect Methods

Although aspects are executed on the aspect server, normal Java classes can remotely call a method declared in the aspects. To make an aspect accessible from remote hosts, the aspect must implement an interface that declares the exported methods.

Suppose that we want to export a displayLog() method to remote hosts. The definition of the aspect should be as follows:

```
interface Logger extends AspectInterface {
   void displayLog(Point p, int x);
}
aspect LoggingAspect implements Logger {
   void displayLog(Point p, int x) {
      System.out.println("set x: " + x);
   }
   ...
}
```

The Logger interface declares the displayLog() method, which is exported to remote hosts. On the remote hosts, normal Java classes can remotely call the displayLog() method as follows:

```
Logger logger = (Logger) Aspect.get("LoggingAspect");
logger.displayLog();
```

Aspect is the class provided by DJcutter. The get method in Aspect returns a remote reference to the aspect with the specified name (in this example, LoggingAspect). The type of the remote reference is the interface type implemented by that aspect. If a method is called on the *proxy* object represented by the remote reference [12], then the corresponding method in the aspect is invoked on the aspect server.

This architecture using the proxy objects is the same as that of Java RMI. [5] The reason why methods in aspects must be called through an interface type is that this architecture enables separate compilation. The developers can compile normal Java classes without aspects, provided that the interface type is avvailable. This is quite helpful in the development of distributed software. Furthermore, this architecture allows the developers to implement components independently of each other. For example, they can start describing a normal class that remotely calls a method in an aspect

[5] This architecture in detail explained in section 4.3.3.

before they have finished describing the aspect, if the interface declaring the exported method is already available.

3.4 Pointcut Parameters

Like AspectJ, DJcutter allows pointcuts to expose the execution context of the join points they identify. For example, the args pointcut designator can expose method paramemters and the target pointcut designator can expose the target object. Each part of the exposed context is bound to a pointcut parameter, which is accessible within the body of the advice. For example,

```
pointcut setter(int x):
   call(void Point.move(int, int))
   && args(x, *);
```

This setter pointcut exposes the first int-type parameter to the move method through a pointcut parameter x.

In DJcutter, since remote pointcuts identify join points on remote hosts, the pointcut parameters should refer to data on the remote hosts. By default, they refer to a copy of that data constructed on the aspect server. The runtime system of DJcutter first serializes the data on the remote hosts, transfers it through the network, and constructs a copy from the serialized data. The pointcut parameters available in the advice body refer to that copy.

Pointcut parameters can be specified as remote references instead of local references to the copies. If the configuration file specifies that pointcut parameters of class type C are remote references, then the runtime system of DJcutter dynamiclly generates a proxy class for C. From the implementation viewpoint, the pointcut parameters are made to refer to instances of that proxy class on the aspect server. If the advice body calls a method on that proxy object, then the method is invoked on the master object on the remote host where the join point occurs. To generate proxy classes, DJcutter uses the replace approach we developed for Addistant. For example, if the remote object associated with a proxy object is a Widget object, then the proxy class is also named Widget. On the aspect server, this proxy class is loaded instead of the original Widget class. The proxy-class generation is performed with our bytecode engineering library Javassist [7].

Remote references are used not only for pointcut parameters but also references to instances of aspects. As shown in the section 3.3, normal Java classes can call methods declared in aspects. The references to the

Table 3.2: Part of methods in JoinPoint class

Methods in JoinPoint	Description
Object[] getArgs()	returns the arguments at this join point.
Signature getSignature()	returns the signature at this join point.
Object getTarget()	returns the target object.
Object getThis()	returns the currently executing object.
String getHost()	returns the name of the host.

instances of the aspects are also remote references implemented using the same approach as for the pointcut parameters. In addition, the parameters of the methods called on the remote object indicated by a remote reference can be also remote references.

3.5 Reflection by thisJoinPoint

As in AspectJ, DJcutter provides the thisJoinPoint special variable for reflective access to join points. This variable refers to an object representing the context of the current join point or advice. It available within the body of the advice code. In the advice, thisJoinPoint is an instance of djcutter.runtime.JoinPoint type. In table 3.2, we show parts of methods defined in the JoinPoint class.

Unlike in AspectJ, This thisJoinPoint variable provided by DJcutter has a getHost() method to acquire the name of the host where the identified join point is located. For example, to change an attribute value of the Point object according to hosts that allocates each join point, users can write an aspect as follows:

```
 1: String lastCallerHost;
 2: void around(Point p, int x):
 3:    target(p)
 4:    && args(x)
 5:    && call(void Point.setX(int))
 6: {
 7:    lastCallerHost = thisJoinPoint.getHost();
 8:    if (lastCallerHost.equals("D"))
 9:       point.setX(x + 5);
10:    else if (lastCallerHost.equals("T"))
11:       point.setX(x - 5);
```

```
12:    else
13:       point.setX(x);
14: }
```

The x variable is an argument of Point.setX(int) method call on each host. In this example, we show that the users can change the argument of the setX() method according to each of hosts. Of course, this code is equivalent to the following code with the hosts pointcut designator.

```
 1: void around(Point p, int x):
 2:    target(p)
 3:    && args(x)
 4:    && hosts(hostId_D)
 5:    && call(void Point.setX(int)) {
 6:    point.setX(x + 5);
 7: }
 8: void around(Point p, int x):
 9:    target(p)
10:    && args(x)
11:    && hosts(hostId_T)
12:    && call(void Point.setX(int)) {
13:    point.setX(x - 5);
14: }
15: void around(Point p, int x):
16:    target(p)
17:    && args(x)
18:    && !hosts(hostId_T, hostId_D)
19:    && call(void Point.setX(int)) {
20:    point.setX(x);
21: }
```

3.6 Local Aspect

In DJcutter, the developers can specify that copies of an aspect are distributed to each participating host and that body of advice in the aspect is locally executed on the same host as where the join points exists. These types of aspects, which are called *local aspects*, are equivalent to the aspects available in AspectJ. Since a local aspect is instantiated on each host, the fields declared in the aspect are not shared among the hosts. A value assigned to such a field on one host is not visible on the other hosts. To

exchange data among the hosts, the data must be explicitly transferred through the network, for example, by using Java RMI.

3.7 Examples

In this section, we show two example programs written in DJcutter to illustrate how remote pointcuts and inter-type declaration can be used for distributed unit testing.

3.7.1 The Use of Remote Pointcut

The testing code presented in chapter 2 was complicated compared to the non-distributed version of the testing code. If we rewrite that testing code in DJcutter, then the resulting code becomes as simple as the non-distributed version:

```
 1: // on host T
 2: aspect AuthServerTest extends TestCase {
 3:   boolean wasAddUserCalled;
 4:   void testRegisterUser() {
 5:     wasAddUserCalled = false;
 6:     String userId = "muga", password = "xxx";
 7:     AuthServer auth
 8:       = (AuthServer) Naming.lookup("auth");
 9:     auth.registerUser(userId, password);
10:     assertTrue(wasAddUserCalled);
11:   }
12:   before(): // remote pointcut
13:     cflow(
14:       call(void AuthServer.registerUser(String,
15:                                         String)))
16:     && execution(void DbServer.addUser(String,
17:                                        String))){
18:     wasAddUserCalled = true;
19:   }
20: }
```

Unlike the code in AspectJ, the testing code in DJcutter is not divided into distributed sub-components (figure 3.3). Although the before advice (lines 12 to 19) is executed when the thread of control reaches the addUser()

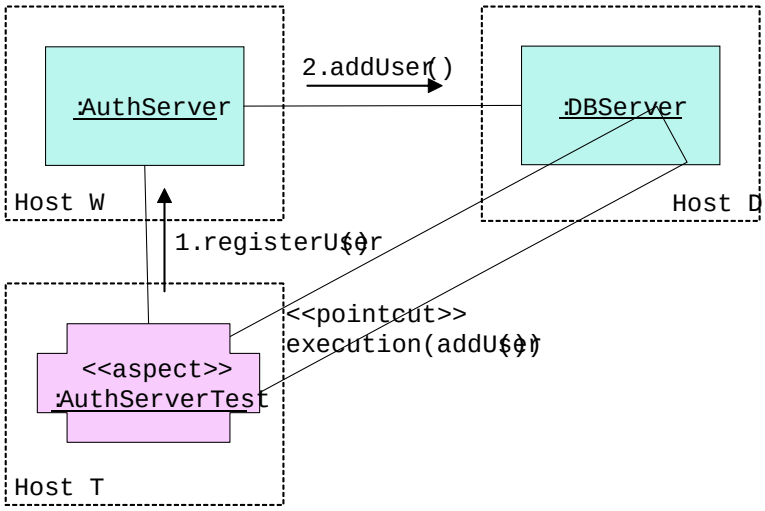

Figure 3.3: The testing code in DJcutter

method on the host D, where the DbServer is running, the execution of the before advice is on a different host T, where the testRegisterUser() method is running. Thus the before advice can directly set wasAddUserCalled to true. All the network processing for reporting the execution of the addUser() method to the host T needs not be explicitly described.

Note that the before advice contains the cflow pointcut designator, since DJcutter provides cflow across multiple hosts if the components communicate with the Java RMI. This improves the accuracy of the testing code. The code can examine not only whether or not addUser() is executed, but also whether the caller to addUser() is registerUser().

The testing code in DJcutter has another advantage. Since DJcutter automatically distributes the definitions of the aspects to each participating host and weaves them at load time, the programmers do not have to manually deploy the compiled and woven code to the hosts whenever the definitions of the aspects are changed for different tests.

3.7.2 The Use of Remote Inter-type Declaration

Unit testing sometime requires accessor methods for inspecting the internal state of objects. AspectJ can be used to append such accessor methods just for testing if these methods are not defined in the original program. For example, the developer may want to confirm that the data sent by the

registerUser() method is actually stored in the database by the addUser() method. To do this, an accessor method containsUser() must be appended to the DbServer class so that the testing code can examine whether the added user entry is contained in the database.

The remote inter-type declaration of DJcutter simplifies such unit testing. If the developers use AspectJ, they have to recompile all the programs and deploy the compiled and woven code to the participating hosts whenever they change the inter-type declaration in the aspect. On the other hand, DJcutter can simplify this deployment. Since DJcutter automatically distributes the new definitions of the aspect to the hosts and weaves it at load time, the new aspect is reflected in the programs if the programs are simply restarted.

The following is the testing code written in DJcutter. It appends containsUser() to the DbServer class (lines 13 to 18). The testRegisterUser() method first confirms that the user muga is not recorded in the database (line 9) and then it calls the registerUser() method (line 10). After that, it confirms that the user muga is recorded in the database (line 11).

```
 1: // on host T
 2: aspect AuthServerTest extends TestCase {
 3:   void testRegisterUser() {
 4:     String userId = "muga", password = "xxx";
 5:     AuthServer auth
 6:       = (AuthServer) Naming.lookup("auth");
 7:     DbServer db
 8:       = (DbServer) Naming.lookup("db");
 9:     assertTrue(!db.containsUser(userId));
10:     auth.registerUser(userId, password);
11:     assertTrue(db.containsUser(userId));
12:   }
13:   boolean DbServer.containsUser(String
14:                                     userId) {
15:     // this method returns true if the user
16:     // entry specified by userId is found
17:     // in the database.
18:   }
19: }
```

Chapter 4

DJcutter Implementation Issues

In this chapter, we present implementaion issues of DJcutter. The DJcutter implementation mainly consists of two parts: the one is a compiler, the other is runtime infrastructure. The role of the compiler is to parse aspects in DJcutter and to generate the normal Java-bytecodes representing the aspects. We call the generated Java class the *compiled aspect*. DJcutter uses *Javassist* [7, 8] for editing and translating the normal Java-bytecode. On the other hand, the role of the runtime infrastructure is to *weave* (compose) the compiled aspects and the Java classes in distributed software on each host at load-time. Moreover, the runtime infrastructure executes distributed software on each of hosts. The aspect server and the class loader that mentioned above (in section 3.1) are included in this runtime.

In the rest of this chapter, at first, we simply explain Javassist, which is a toolkit for the Java-bytecode translators. It is widely used within the classes that are consisted by DJcutter. At second, we present how the compiler of DJcutter generates the Java-bytecodes representing aspects. At last, we explain the implementation of the DJcutter runtime.

4.1 Java-bytecode Translation by Javassist

The compiler and runtime infrastructure of DJcutter edit and translate the Java-bytecode by using Javassist. Javassist is our bytecode engineering library, and a reflection-based toolkit for developing the Java bytecode translators. It is a class library in the Java language for transforming Java class files (bytecode) at compile-time or load-time. Unlike other similar libraries

[10, 3] that are not based on reflection, it provides a source-level view of bytecode for the developers, who can manipulate bytecode without detailed knowledge of the bytecode or the internal structure of the Java bytecode. Javassist is easier to use than other naive toolkits as a source-level debugger is easier to use than an assembly-level debugger. On the other hand, Javassist restricts the ability to modify bytecode. It does not allow bytecode modification that is difficult to express with a source-level view.

The Javassist users can first translate a Java class file into several objects representing a class, field, or method. The users' programs can access these *meta objects* for transformation. Introducing a super interface, a new field, and so on, to the class is described with these objects. The modifications applied to these metaobjects are finally translated back into the modifications of the class file so that the transformation is reflected on the class file. Since Javassist does not expose internal data structures contained in a class file, such as a constant pool item and a method_info structure, the developers can use Javassist without knowledge about Java class file or bytecode.[1] On the other hand, other libraries such as BCEL[10] provide objects directly corresponding to a constant pool item and a method_info structure.

4.1.1 Structual Reflection

The CtClass object is an object provided by Javassist for representing a class obtained from the given class file. It provides the almost same functionality of introspection as the java.lang.Class class of the standard reflection API. Introspection means to inspect data structures, such as a class, used in a program. For examle, the getName method returns the CtClass object representing the super class. GetFields, getMethods, and getConstructors return CtField, CtMethod, CtConstructor objects representing fields, methods, and constructors, respectively. These objects parallel java.lang.reflect.Field, Method, and Constructor. They provide various methods, such as getName and getType, for inspecting the definition of the member. Since a CtClass object does not exist at run-time, the newInstance method is not available in CtClass unlike in java.lang.Class. For the same reason, the invoke method is not available in CtMethod and so forth.

Unlike the standard reflection API, Javassist allows developers to alter the definition of classes through CtClass objects and the associated objects (Table 4.1, 4.2, and reftab:ctmember). For example, the setSuperclass

[1] For practival reasons, Javassist also provides another programming interface to directly access the internal data structures in a class file. However, normal users do not have to use that interface.

Table 4.1: Part of methods for modifying a class

Methods in CtClass	Description
void setName(String name)	changes the class name.
void setModifiers(int m)	changes the class modifiers.
void setSuperclass(CtClass c)	changes the super class.
void setInterfaces(CtClass[] i)	changes the interfaces.
void addField(CtField f, String i)	adds a new field.
void addMethod(CtMethod m)	adds a new method.
void addConstructor(CtConstructor c)	adds a new constructor.
void instrument(ExprEditor e)	modifies the bodies of all methods and constructors declared in the class.

Table 4.2: Part of methods for modifying a field

Methods in CtField	Description
void setName(String name)	changes the field name.
void setModifiers(int m)	changes the class modifiers.
void setType(CtClass c)	changes the field type.

method in CtClass changes the super class of the class. The addMethod method adds a new method to the class. The definition of the new method is given in the form of String object representing the source code. Javassist compiles the sourcecode into bytecode on the fly and adds it into the class file. The addField method adds a new field. Javassist compiles the source-code and inserts it in the constructor body so that the field is appropriately initialized.

The setName method in CtClass changes the name of the class. To keep consistency, several methods like setName perform more than changing one attribute field in a class file. For example, setName also substitutes the new class name for all occurrences of the old class name in the class definition. The occurrences of the old class name in method signatures are also changed.

4.1.2 Bytecode Edition with a Source-level View

Javassist newer than version 2.4 allows modifying only expression included in a method body by using javassist.expr.ExprEditor class. Javassist.expr.ExprEditor class is an editor for replacing an expression in a method body. The Javassist

Table 4.3: Part of methods for modifying a method or constructor

Methods in CtMethod, CtConstructor	Description
void setName(String name)	changes the method name.
void setModifiers(int m)	changes the class modifiers.
void setExceptionTypes(CtClass[] t)	sets the types of the exceptions that the method may throw.
void setBody(String b)	changes the method body.
void instrument(ExprEditor e)	modifiers the method body.

users can define a subclass of ExprEditor class to customize how to modify a method body. The overall architecture is similar to the *strategy pattern*[13].

As an example, supposing to print a message, when the method setX() defined in Point class is called. Javassist user should describe the following code.

```
CtMethod cm = ... ;
cm.instrument(
   new ExprEditor() {
      public void edit(MethodCall m)
               throws CannotCompileException
      {
         if (m.getClassName().equals("Point")
             && m.getMethodName().equals("setX"))
            m.replace("{
                        System.out.println(\"setX is called\");
                        $_ = $proceed($$);
                      }");
      }
   });
```

To run an ExprEditor object, the users must call instrument() in CtMethod or CtClass. This program searchs the method body represented by cm and replaces all calls to setX in class Point with a block:

```
{
   System.out.println("setX is called");
   $_ = $proceed($$);
}
```

The method instrument() searches a method body. If it finds an expression such as a method call, field access, object creation, and exception handling, then it calls edit() on the given ExprEditor object. In this example, since the users want to found the expression that is called the setX() method, they should use the MethodCall object that represents a method call. The parameter to edit() is an object representing the found expression. The edit() method can inspectand replace the expression through that object. Calling replace() on the parameter to edit() substitutes the given statement or block for the expression. If the given block is an empty block, that is, if replace("") is executed, then the expression is removed from the method body. The second statement:

```
$_ = $proceed($$);
```

is the code that calls a method setX() actually. Then, the first statement enables user to insert the code that prints the message setX is called before calling the method setX().

4.2 Compiler

The role of DJcutter compiler is to parse aspects in DJcutter and to generate Java-bytecodes representing the source files of aspects that developers described. When parsing the aspects, the compiler doesn't create and edit Java-source files (.java files). Instead, It generates directly the Java-bytecodes (.class files) by using Javassist (in section 4.1). By using the compiled aspects that the compiler generated in advance, the runtime infrastructure of DJcutter on each host runs distributed software. Basically, the compiler of DJcutter outputs the Java-bytecodes same as the code that the compiler of AspectJ 1.0.6 outputs.

In the rest of this section, we explain how each member in an aspect is compiled by the compiler.

4.2.1 Aspect Declarations

Even if the compiler parses an aspect declaration that DJcutter users described, it creates an array of bytes (bytecode) of the Java class declaration. The compiler generates a class file of the same name as the aspect name. As an example, supposing that the compiler parses the following aspect.

```
aspect LoggingAspect {
}
```

Then, by referring the aspect declaration, the compiler generates an array of bytes of the following class.

```
public class LoggingAspect {
}
```

The Java class declaration representing an aspect is always pertained the public modifier by the compiler. This reason is that the compiled aspects and the normal Java classes in distributed software are always executed on each different Java Vertual Machines (JVMs) in the current implementation of DJcutter. Even though the compiler parses a *local aspect* declaration, it generates same class declaration as default aspect.

4.2.2 Advice Declarations

Even if the compiler parses advice declarations in an aspect, it makes static methods as advice in the compiled aspect. In body of generated static method, the compiler inserts the body of advice without modification. For example, supposing that the compiler reads the advice:

```
aspect LoggingAspect {
   before(): setter() {
      System.out.println("set x");
   }
}
```

Then it generates the static method:

```
public class LoggingAspect {
   public static void before_$0() {
      System.out.println("set x");
   }
}
```

A static method declaration representing an advice is always pertained the public modifier. Moreover, the name of static method, before_$0, is automatically given by the compiler. The reason why this name before_$0 is given is that the parsed advice is before advice. As if a parsed advice were after advice, the name of static method were after_$0. The charactor with that before_$0 ends, 0, represents number of advice. Even if two before advice declarations are defined in an aspect by developers, the compiler generates two static methods named before_$0 and before_$1.

4.2.3 Member Methods and Fields

Member methods and fields that DJcutter users defined in an aspect are directly inserted by the compiler, that is same signatures, same modifiers, and same method bodies, within the compiled aspect.

4.2.4 Pointcut Declarations

When the compiler parses pointcut declarations in an aspect, it doesn't reflects the information of pointcut declarations in the compiled aspect. Instead, it creates objects of the Pointcut type by using the information of pointcut declarations. Then, it passes the created Pointcut objects to the aspect server of DJcutter runtime (in section 4.3.1).

The Pointcut object has two fields: joinpoints and adviceNames. The joinpoints field is a reference of JoinPoints class defined in DJcutter runtime. The pointcut declaration that users describe:

```
within(Point) && call(void Point.setX(int))
```

is assigned to the joinpoints field by the compiler when the compiler parsed the declaration. On the other hand, the type of adviceNames field is an array of the java.lang.String objects. Each java.lang.String object is names of advice (static method) that is executed by DJcutter runtime when the thread of control reaches each of join points in pointcut (e.g. before_$0).

4.2.5 Inter-type Declarations

The compiler doesn't reflect information of the inter-type declarations in the compiled aspect same as pointcut declarations. The compiler creates objects of the InterTypeDecl class defined in DJcutter by using the information of inter-type declarations. Then it passes the created InterTypeDecl object to the aspect server of DJcutter runtime (in section 4.3.1).

4.3 Runtime Infrastructure

4.3.1 Load-time Weaving

Aspect Server

The roles of aspect server are mainly as follows:

– First one is to execute the bodies of advice defined in an aspect on same JVM that the aspect server is running.

– Second one is to register information of pointcut declarations and inter-type declarations parsed by the compiler of DJcutter.

The Execution of bodies of Advice using Reflection. Aspect server executes the codes of advice defined in an aspect, when the thread of control reached each of join points in pointcut on every hosts. By default, the code of advice in the aspect is executed by the aspect server only [2]. We implement it using reflection technology [33]. As an example, we show the code of advice execution defined in the aspect server as follow:

```
Class aspectClass = Class.forName(aspectName);
Method advice = aspectClass.getDeclaredMethod(adviceName,
                                                parameterTypes);
Object result = advice.invoke(null, parameterValues);
```

At first, the aspect server receives the values of variables, aspectName, adviceName, parameterTypes, and parameterValues as a message, through the network from DJcutter runtime on other JVMs so that it will execute an advice. The value of aspectName is the name of aspect, the value of adviceName is the name of static method as advice, the value of parameterTypes is type of pointcut parameters, and the value of parameterValues is the arguments of pointcut parameters. Next, the aspect server creates a Class object of the aspect specified by aspectName name. Then it creates a Method object representing the static method as the advice specified by adviceName. At last, it executes the invoke() method defined in Method object in order to execute the advice. [3]

Registering the Information of Aspect Declarations. Once the compiler of DJcutter parsed aspects and required the information of pointcuts and inter-type declarations in the aspects, the compiler registers the information into the aspect server. DJcutter runtime on each host doesn't know which join points are designated in pointcuts and which classes are appended as inter-type declaration on startup. So DJcutter runtime on each host asks the aspect server the information on load-time and acquires it. Then the runtime weaves the Java classes and the compiled aspects by using it.

The aspect server has two fields pointcuts and interTypeDecls. The type of each field is java.util.Vector. Pointcuts is the field for registering the information of pointcut declarations, that is Pointcut objects. InterTypeDecls

[2] In section 3.6, we showed that aspect server executed advice bodies in aspects excepting a *local aspect* declaration.

[3] The aspect server technically can execute the codes of advice, without using reflection technology.

is the field for registering the inter-type declarations, that is InterTypeDecl objects. DJcutter runtime on each host choose some objects from the Pointcut objects and InterTypeDecl objects that the aspect server registers on load-time. Then it weaves the Java classes by using those objects.

Class Loader provided by DJcutter

DJcutter provides unique class loader for implementing remote pointcuts mechanism. As mentioned above (in section 4.3.1), DJcutter runtime on each host doesn't know which join points are identified in pointcuts and which classes are declared in inter-type declarations on startup. Therefore, the class loader provided by DJcutter asks the aspect server those information, receives the information matching join points within a loading class, and weaves (translates) a loading class by using the information. It is called *load-time weaving*.

In the Java language, developers can implement subclass of java.lang.ClassLoader class in order to extend the manner in which the JVM dynamically loads classes [19]. We implement the findClass method defined in ClassLoader class for achieving load-time weaving. The following code is the findClass() method defined in the class loader provided by DJcutter runtime.

```
 1: Class findClass(String className) {
 2:     ...
 3:     byte[] classFile = ...;
 4:     ...
 5:     Pointcut[] pcs = getPointcuts(className);
 6:     InterTypeDecl[] itds = getInterTypeDecls(className);
 7:     ...
 8:     classFile = weaver.weave(classFile, pcs, itds);
 9:     ...
10:     return defineClass( ... );
11: }
```

GetPointcuts() is the method that DJcutter runtime ask the aspect server whether join points within the loading class are designated in pointcut or not. It returns an array of Pointcut objects even if join points are designated in pointcut(line 5). GetInterTypeDecl() is also the method that DJcutter runtime ask the aspect server whether the fields and methods of loading class are declared in aspects or not. It returns an array of InterTypeDecl objects even if the fields and methods are declared (line 6). The processing on line 8 is that an *aspect weaver* of DJcutter runtime, the weaver variable, recieves

the arguments pcs and itds from the class loader and weaves (translates) the loading class by using the information of pointcuts and inter-type declarations defined in aspects. After weaving the loading class, the aspect weaver returns the array of bytes of loading class to the class loader (line 8). Next, the class loader converts the array of bytes into an instance of class java.lang.Class using the defineClass method defined in class ClassLoader (line 10).

Aspect Weaver

The most significant role of weave() in an aspect weaver is to insert the codes for invoking the bodies of advice by bytecode translation at load-time. As mentioned in section 4.3.1, the aspect weaver is executed by the class loader. Then it modifies the array of bytes of the class by using the Pointcut objects and InterTypeDecl objects passed by the class loader.

The aspect weaver checks whether each of join points within the loading class, such as a method call, a field access, an object creation, and exception event, matches the designated join points in pointcuts or not. Even if a join point within the loading class matches the designated join points, the aspect weaver immediately starts weaving. We show an example of weaving the Line class and LoggingAspect aspect as follows:

```
 1: class Line {
 2:    void moveX(int dx) {
 3:       int x1 = p1.getX();
 4:       p1.setX(x1 + dx);
 5:       int x2 = p2.getX();
 6:       p2.setX(x2 + dx);
 7:       observer.paint();
 8:    }
 9: }
10: aspect LoggingAspect {
11:    pointcut setterX():
12:       within(Line)
13:       && call(void Point.setX(int));
14:    before(): setterX() {
15:       System.out.println("set x");
16:    }
17: }
```

Note that the compiler of DJcutter compiles the LoggingAspect aspect (lines

10-17), and then it generates the following code.

```
class LoggingAspect {
   static void before_$0() {
      System.out.println("set x");
   }
}
```

In this example, join points designated by Pointcut object is the Point.setX method call. The aspect weaver of DJcutter checks whether each join point within moveX() of the Line class is equal to the Point.setX method call or not. First join point within moveX() is the Point.getX method call (line 3). The Point.getX method call is not equal to the designated join point. Next, join point within moveX() is the Point.setX method call (line 4). As this join point is equal to the designated join point as pointcut, the weaver inserts the code that invokes LoggingAspect.before_$0() before calling Point.setX(). On the same way, the weaver checks whether join points of lines 5 to 7 are equal to the designated join point or not. As a result, the weaver modifies the body of moveX() as follow.

```
void moveX(int dx) {
  int x1 = p1.getX();
  LoggingAspect.before_$0();
  p1.setX(x1 + dx);
  int x2 = p2.getX();
  LoggingAspect.before_$0();
  p2.setX(x2 + dx);
  observer.paint();
}
```

The class loader provided by DJcutter receives the array of modified bytes from the weaver, and then loads those.

The Dynamic Join Point Model

The design of DJcutter language is the dynamic join point model same as AspectJ language. The aspect weaver of DJcutter can insert not only the code for executing the bodies of advice but also a code for evaluating a dynamic context of the original join point. Note that, Wether the aspect weaver inserts or not is independent on the pointcut declarations that users wrote.

Even if the users wrote the aspect in DJcutter:

```
aspect LoggingAspect {
   pointcut testEquality(Point p):
      target(Point)
      && args(p)
      && call(boolean equals(Object));
   before(Point p): testEquality(p) {
      System.out.println("set x");
   }
}
```

, the aspect weaver inserts the following code into each of join points.

```
if (o instanceof Point) {
   LoggingAspect.before_$0((Point) o)
}
```

In this case, if the argument passed to equals() is not Point type, the advice (before_$0 method) is not executed.

4.3.2 Design and Implementation of thisJoinPoint

Whereas the design of thisJoinPoint variable that developers use in advice declaration is same as AspectJ 1.0.6, the implementation of thisJoinPoint is different between AspectJ and DJcutter. The thisJoinPoint variable provided by DJcutter is an object of type JoinPoint (JoinPoint is an interface).

In the implementation of DJcutter runtime, DJcutter must pass the information of designated join point, that is JoinPoint object, to the aspect server. To do that, DJcutter edits and translates the bytecode of compiled aspects. The aspect weaver of DJcutter adds one parameter to the parameters of static method representing the advice. And it edits for giving the information of designated join point to the added parameter, that is, first parameter-type of a static method representing an advice is the JoinPoint type.

For example, the aspect weaver of DJcutter translates the advice:

```
static void before_$0(Point p, int x) {
}
```

into the following code.

```
static void before_$0(JoinPoint jp, Point p, int x) {
}
```

4.3.3 Remote Reference Implemented *Proxy* Pattern

In the current implementation of DJcutter runtime, even if the compiled aspects and the normal Java classes in distributed software are allocated on a single host, these are performed on different Java Vertual Machines (JVMs). The specification is same as the Enterprise Java Bean (EJB) 1.1 [34]. In the EJB 1.1 speficiation, even if some beans are allocated on a single host, each of beans are performed on different processes (JVMs). DJcutter runtime must implement inter-process communication. However, DJcutter users need not to implement explicit network processing such as Java RMI or socket programming.

Proxy-Master Model

DJcutter automatically implements the references to remote objects by bytecode translation at load-time, in order to implicitly archieve the inter-process communication. To run the translated software, no custom JVM is needed. DJcutter employs the *proxy-master model*, which is also known as the *remote proxy pattern*[12], so that a remote method can be transparently invoked with the same syntax as a local method. In this model, an object whose methods can be invoked from a remote host is associated with an object called *proxy* existing on that remote host. For distinction, we call the former object *master*. A proxy provides the same set of methods as its master and delegates every method invocation to its master. It encapsulates details of network communication necessary for the remote method invocations.

In the rest of this paper, we write A_proxy as remote reference to master object A, that is proxy object.

Cases that Remote References are generated

DJcutter runtime automatically generates remote references at load-time. Cases that remote references are generated are as follows:

– First one is a case that the thread of control reaches the identified join point in pointcuts. If the pointcut parameter-types are reference types, those parameters are translated remote references and are passed to the aspect server so that the aspect server will use those parameters.

  As an example of this case, we explain the following aspect.

```
aspect LoggingAspect {  // on A
   pointcut setter(Point p, int x):
```

```
      target(p)
      && args(x)
      && execution(void Point.setX(int));
   before(Point p, int x) {
      System.out.println("set x: " + x);
   }
}
```

The aspect prints a message (executes the before advice) just before being setX() method executed. Note that, The aspect is translated the following Java class by the DJcutter compiler (in section 4.2) already.

```
class LoggingAspect {  // on A
   static void before_$0(Point p, int x) {
      System.out.println("set x: " + x);
   }
}
```

However, the compiled aspect and the Point object are executed on each of diffenrent processes. Therefore, the LoggingAspect class is translated the following code by DJcutter runtime at load-time.

```
class LoggingAspect {  // on A
   static void before_$0(Point_proxy p, int x) {
      System.out.println("set x: " + x);
   }
}
```

The Point_proxy object is the remote reference of Point object. The Point_proxy class is automatically created by DJcutter runtime on demands at load-time.

– Second one is a case that the distributed Java application classes require an reference to compiled aspect by using Aspect.get() method. The Aspect.get() method explained in section 3.3. Aspects and Java classes that are cosisted distributed systems are executed on each of diffenrent processes, by default. Therefore, the Aspect.get() method returns a remote reference to the compiled aspect with the specified name.

As an example of the *aspect reference* case, suppose that developers use the displayLog() method defined in the LoggingAspect aspect. LoggingAspect is run on the host A. In this case, they should describe the following code.

```
Logger logger = (Logger) Aspect.get("LoggingAspect");
logger.displayLog(p);  // The variable p is the Point object
```

So that the program code mentioned above is executed on different host from the host that the LoggingAspect object is allocated, the variable logger receives the remote reference LoggingAspect_proxy to LoggingAspect object. At load-time, If LoggingAspect_proxy class doesn't exist on the host A, DJcutter runtime create this class by bytecode translation on the fly. Note that the remote reference LoggingAspect_proxy has to implement the Logger interface.

– Third one is a case that DJcutter runtime executes the call of member method in an aspect within distributed Java application classes. If parameter-types of the member method defined in an aspect are reference types, remote references of parameters are passed to the method through the network.

```
class LoggingAspect {
   void displayLog(Point p) {
      System.out.println("point: " + p);
   }
}
```

As an example, supposing that the displayLog() method is defined in the LoggingAspect aspect. This aspect is translated at load-time as follows:

```
class LoggingAspect{
   void displayLog(Point_proxy p) {
      System.out.println("point: " + p);
   }
}
```

– Fourth is a case that the DJcutter runtime executes member methods defined in proxy class. As mentioned above, DJcutter users enable

to use remote references as the pointcut parameters within the bodies of advice. When DJcutter runtime executes the member methods defined in the remote references, even if the parameter-types of member methods are reference types, those parameters are translated to remote references.

Several Approaches for Implementing Proxy

Unfortunately, any single implementation approach of the proxy-master model cannot deal with all kinds of classes. Each approach convers only the classes satisfying the criteria peculiar to that approach. The developers cannot choose a single approach and enforce the criteria on the whole program. For example, one of the approaches needs to modify the declaration of the class of master objects. Since the JVM does not accept modified system classes, if an instance of a system class is a remote object, that approach cannot be used. A different approach must be used for that case.

To avoid this problem, Addistant [37] ,which is our old work, provides several different approaches for implementing the proxy-master model. It provides four approaches; *replace*, *rename*, *subclass*, and *copy*. The developers can choose one from the four for each class of master. The differences among the four approaches are mainly how a proxy class is declared, how caller-side code, that is, expressions of remote method invocations, is modified, and how a master class is modified. The four approaches cover most of case.

DJcutter allows developers to specify a policy of proxy implementaion for each of classes. The developers can declare that every instances of remote reference of the class are implemented the specified approach. The policy declaration is written in a policy file in an XML syntax. DJcutter runtime receives the policy file that the developers write on startup, it parses the file as an XML document. Then, DJcutter runtime automatically implements remote references of each classes on demands at load-time. As an example, the policy file:

```
<policy>
   <class name="Point" proxy="replace">
   <class name="java.lang.String" proxy="copy">
   <aspect name="LoggingAspect" proxy="replace">
</policy>
```

means that every instances of remote reference of the Point class is implemented by using *replace* approach, java.lang.String class is implemented by

using *copy* approach, and LoggingAspect class representing the LoggingAspect aspect is implemented by using *replace* approach. Currently, DJcutter allows to choose one from *replace* and *copy* approaches.

4.3.4 Deadlock Avoidance with ThreadLocal Object

Any host can invoke a method on a remote object and receive a method invocation from a remote object in DJcutter runtime. Therefore, a remote method call from a host A to an other host B may cause another method call back from B to A. In this case, the latter method call must be handled by the same thread that requested the former method call on the host A. Otherwise, a deadlock may occur if the methods are synchronized ones.

In order to ensure the same thread executes all the methods called back, DJcutter establishes a one-to-one communication channel between the thread executing a method on the host B and the thread executing a method that is handled the method on the host A. This communication channel is stored in a thread-local variable implemented with java.lang.ThreadLocal. The ThreadLocal class provides thread-local variables for the Java developers. These variables differ from their normal counterparts in that each thread that accesses one has its own, independently initialized copy of the variable.

For example, the following code getCommChannel() (lines 2 to 11) is that DJcutter runtime on each host establishes a communication channel.

```
 1: static ThreadLocal commChannel = new ThreadLocal();
 2: CommChannel getCommChannel(Isolate remoteVM) {
 3:    CommChannel channel = (CommChannel) commChannel.get();
 4:    if (channel != null)
 5:       return channel;
 6:    ...
 7:    channel = CommChannel.createSocketChannel(this, remoteVM);
 8:    setCommChannel(channel);
 9:    ...
10:    return channel;
11: }
12: void setCommChannel(CommChannel channel) {
13:    commChannel.set(channel);
14: }
```

The CommChannel class is defined in DJcutter runtime, an object of this class represents a communication channel between a local thread and an

other thread (representing the remoteVM variable). The remoteVM variable is the object of Isolate class in DJcutter runtime, we implemented Isolate class according to [4][9]. Even if channel is registered in the commChannel variable already (line 2), getCommChannel() returns channel (line 3). Otherwise (that is , if channel is not registered), getCommChannel() creates a new communication channel, and then returns it (lines 7 to 10).

A thread always uses the same channel for every remote method call and it waits for not only the result of the invocation but also another request of invocation from a remote thread sharing the same channel. A deadlock is avoided.

4.3.5 Tuning RMI using Thread Pool

One simplistic model for building a RMI server program would be to create a new thread each time a request arrives and serivce the request in the new thread. This approach actually works fine for prototyping though, has significant disadvantages that would become apparent if you tried to deploy the RMI server program that worked this way. One of the disadvantages of the thread-per-request approach is that the overhead of creating a new thread for each request is significant, the server that created a new thread for each request would spend more time and consume more system resources creating and destroying threads than it would processing actual user requests. In addition to the overhead of creating and destroying threads, active threads consume system resources. Creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption.

To resolve this problem, current DJcutter tries to reduce the overhead of creating and destroying threads by using the thread pool [17]. Actually, we implement a thread pool (ThreadPool class) (lines 1 to 17)that combined with a fixed group of worker threads (Worker class) (lines 18 to 35). The thread pool uses wait() (line 25) and notify() (line 14) to signal waiting threads that new work has arrived.

```
1: class ThreadPool {
2:    final int threadNum;
3:    final Worker[] threads;
4:    LinkedList pool;
5:    ThreadPool(int num) {
6:       this.threadNum = num;
7:       pool = new LinkedList();
```

```
 8:       threads = new Worker[this.threadNum];
 9:       ... // initializes and starts each Worker
10:    }
11:    void execute(Runnable r) {
12:       synchronized(pool) {
13:          pool.addLast(r);
14:          pool.notify();
15:       }
16:    }
17: }
```

The ThreadPool class has a field pool (line 4), which monitors the worker threads and implements as LinkedList class. Objects of the Worker class implement the Runnable interface.

```
18: private class Worker extends Thread {
19:    public void run() {
20:       Runnable r;
21:       while (true) {
22:          synchronized(pool) {
23:             while (pool.isEmpty()) {
24:                ...
25:                pool.wait();
26:                ...
27:             }
28:             r = (Runnable) pool.removeFirst();
29:          }
30:          ...
31:          r.run();
32:          ...
33:       }
34:    }
35: }
```

The thread pool in DJcutter meets the requirements for safely using notify(). The thread pool offers a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks. As a bonus, because the thread already exists when a request arrives, the delay introduced by thread creation is eliminated. Thus, the request can be serviced immediately, rendering the application more responsive. Furthermore, by properly tuning the number of threads in the thread pool, you

can prevent resource thrashing by forcing any requests in excess of a certain threshold to wait until a thread is available to process it.

Chapter 5

Experiment

5.1 Performance Measurement of RMI

To examine the execution performance of remote pointcuts, we compared the execution time of the programs in DJcutter and AspectJ using Java RMI. For this experiment, we used the testing programs shown in section 2.2.2 (AspectJ using Java RMI) and section 3.7.1 (DJcutter). These programs examine whether registerUser() in AuthServer remotely calls addUser() in DbServer. We measured the elapsed time of the testRegisterUser() method for each program. The body of the addUser() method was empty. In this experiment, the AuthServer and the AuthServerTest ran on the same host while DbServer ran on another host. The AuthServer host was a Sun Blade 1000[1] and the DbServer was a Sun Fire V480[2]. These hosts were connected through a 100 BaseTX network. We used Sun JDK 1.4.0_01 and AspectJ 1.0.6.

Table 5.1: The elapsed time (msec.) of testRegisterUser()

Pointcut parameters	()	(String)	(String,String)
Java + Java RMI	5.9	5.9	6.0
AspectJ + Java RMI	5.9	6.0	6.0
DJcutter	4.8	4.9	5.0
DJcutter without cflow	4.8	4.9	4.9

Table 5.1 lists the results of our measurement. Although the program in

[1]Dual UltraSPARC III 750 MHz with 1 GB of memory and Solaris 8.
[2]UltraSPARC III Cu 900 MHz ×4 with 16 GB of memory and Solaris 9.

DJcutter was slightly faster than in AspectJ, this result does not mean DJcutter is considerably faster than AspectJ using Java RMI. In the program in AspectJ (see section 2.2.2), when the body of the before advice is executed, a remote reference test (lines 31 to 33) is obtained for calling confirmCall(). On the other hand, this remote reference is not obtained in DJcutter during the measurement. It is implicitly obtained by the runtime system in advance. Since obtaining this remote reference needs remote access to the registry server, this difference caused about 1 milli-second ahead of DJcutter in the measurement. We confirmed this fact by other experiment.

The programs shown in section 2.2.2 and 3.7.1 do not use pointcut parameters. To evaluate effects by sending pointcut parameters through a network, we also examined the programs in that the before advice (in DJcutter) or the confirmCall() method (in AspectJ) receives one or both of the parameters to the addUser() method (DJcutter). The type of the parameters is the String class. The results of our measurement showed that the performance impacts by pointcut parameters are small.

For fair comparison, we also measured the elapsed time of the program written in DJcutter without cflow since the program in AspectJ did not use cflow. The results were similar to those of the program using cflow since the overhead due to cflow across a network is not significant.

5.2 High Readability and Maintainability of an Aspect

In this section, to show high readability and maintainability of an aspect in DJcutter, we compared aspects in DJcutter and AspectJ with Java RMI. Figure 5.1 is each of aspects written in DJcutter (the left-side code) and AspectJ using Java RMI (the right-side code). As comparing with the aspect in AspectJ with Java RMI, the aspect written in DJcutter is a brief description. Whereas the aspect in AspectJ includes several complicated codes for explicit network processing, these complicated codes surely reduce within the aspect in DJcutter.

```
1: // on host T
2: aspect AuthServerTest extends TestCase {
3:   boolean wasAddUserCalled;
4:   void testRegisterUser() {
5:     wasAddUserCalled = false;
6:     String userId = "muga", password = "xxx";
7:     AuthServer auth
8:       = (AuthServer) Naming.lookup("auth");
9:     auth.registerUser(userId, password);
10:    assertTrue(wasAddUserCalled);
11:  }
12:  before(): // remote pointcut
13:    cflow(
14:      call(void AuthServer.registerUser(String,
15:                                        String)))
16:    && execution(void DbServer.addUser(String,
17:                                       String))){
18:    wasAddUserCalled = true;
19:  }
20: }
```

```
1: // on host T
2: class AuthServerTest extends TestCase {
3:   boolean wasAddUserCalled;
4:   void testRegisterUser() {
5:     Naming.rebind("test", new RecieverImpl());
6:     wasAddUserCalled = false;
7:     String userId = "muga", password = "xxx";
8:     AuthServer auth
9:       = (AuthServer) Naming.lookup("auth");
10:    auth.registerUser(userId, password);
11:    assertTrue(wasAddUserCalled);
12:  }
13:  class ReceiverImpl
14:      extends UnicastRemoteObject
15:      implements NotificationReceiver {
16:    void confirmCall() {
17:      wasAddUserCalled = true;
18:    }
19:  }
20: }
21:
22: interface NotificationReceiver
23: { // on both hosts
24:   void confirmCall();
25: }
26:
27: aspect Notification { // on host D
28:   before():
29:     execution(void DbServer.addUser(String,
30:                                     String)) {
31:       NotificationReceiver test
32:         = (NotificationReceiver)
33:           Naming.lookup("test");
34:       test.confirmCall();
35:   }
36: }
```

Figure 5.1: Comparing each aspect in DJcutter and AspectJ with Java RMI The left-side code is the aspect in DJcutter. The right-side code is the aspect in AspectJ with Java RMI.

Chapter 6

Concluding Remarks

This paper presented DJcutter, which provides remote pointcuts as a new language construct for distributed aspect-oriented programming. A remote pointcut is a function for identifying join points in the execution of a program running on a remote host. It can simplify the description of aspects with respect to network processing if the aspects implement a crosscutting concern spanning over multiple hosts. To illustrate this situation, this paper used the example of a program written in DJcutter for distributed unit testing. The remote pointcut is a crucial language construct for distributed aspect-oriented programming, corresponding to remote method invocation (RMI) for distributed object-oriented programming.

Although we adopted load-time weaving for DJcutter, runtime weaving is more appropriate if we use DJcutter as a testing framework for distributed software. This allows the developers to change the testing code written as an aspect without restarting the target software they are testing. Extending DJcutter is our future work to enable runtime weaving such as exists PROSE [30] and Wool [31]. Another area of our future work is performance improvement. Although the advice bodies in all aspects are currently executed in the aspect server, this centralized approach might be a performance bottleneck. We will extend DJcutter to allow multiple aspect servers for better performance.

Bibliography

[1] Apache Software Foundation: *CACTUS*, Online publishing, URI http://jakarta.apache.org/cactus/ (2000).

[2] Aridor, Y., Factor, M. and Teperman, A.: cJVM: A Single System Image of a JVM on a Cluster, *International Conference on Parallel Processing 1999 (ICPP 1999)*, pp. 4–11 (1999).

[3] Back, G.: DataScript - A Specification and Scripting Language for Binary Data, *Generative Programming and Component Engineering 2002 (GPCE 2002)*, LNCS 2487, Springer (2002).

[4] Balfanz, D. and Gong, L.: Experience with Secure Multi-Processing in Java, *The 18th International Conference on Distributed Computing Systems (ICDCS 1998)*, pp. 398–405 (1998).

[5] Beck, K.: *Extreme Programming Explained: Embrace Change*, Addison-Wesley, chapter 4 (1999).

[6] Bergmans, L. and Aksits, M.: Composing crosscutting concerns using composition filters, *CACM*, ACM Press (2001).

[7] Chiba, S.: Load-time Structural Reflection in Java, *European Conference on Object-Oriented Programming 2000 (ECOOP 2000)*, LNCS 1850, Springer Verlag, pp. 313–336 (2000).

[8] Chiba, S. and Nishizawa, M.: An Easy-to-Use ToolKit for Efficient Java Bytecode Translators, *Generative Programming and Component Engineering 2003 (GPCE 2003)*, LNCS 2830, SV, pp. 364–376 (2003).

[9] Czajkowski, G.: Application Isolation in the Java Virtual Machine, *Object-Oriented Programming Systems, Languages, and Applications 2000 (OOPSLA 2000)*, Springer Verlag, pp. 354–366 (2000).

[10] Dahm, M.: Byte Code Engineering, *Java Information Tage (JIT 1999)* (2000).

[11] Eclipse Organization: *aspectj project*, Online publishing, URI http://www.eclipse.org/aspectj/ (2001).

[12] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, chapter 4 (1995).

[13] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, chapter 5 (1995).

[14] IBM Research: *Hyper/J: Multi-Dimensional Separation of Concerns for Java*, Online publishing, URI http://www.research.ibm.com/hyperspace/HyperJ/hyperJ.html.

[15] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *European Conference on Object-Oriented Programming 2001 (ECOOP 2001)*, LNCS 2072, Springer, pp. 327–353 (2001).

[16] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C. V., Maeda, C. and Mendhekar, A.: Aspect-Oriented Programming, *European Conference on Object-Oriented Programming 1997 (ECOOP 1997)*, LNCS 1241, Springer, pp. 220–242 (1997).

[17] Lea, D.: *Concurrent Programming Second Edition Design Principles and Patterns*, Addison Wesley Java Series, chapter 4 (2000).

[18] Lesiecki, N.: *Test flexibly with AspectJ and mock objects*, IBM developerWorks, Online publishing, URI http://www-106.ibm.com/developerworks/java/library/j-aspectj2/ (2000).

[19] Liang, S. and Bracha, G.: Dynamic Class Loading in the Java Virtual Machine, *Object-Oriented Programming Systems, Languages, and Applications 1998 (OOPSLA 1998)*, ACM SIGPLAN Notices, pp. 36–44 (1998).

[20] Lieberherr, K. J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston (1996). ISBN 0-534-94602-X.

[21] Liskov, B. and john Guttag: *Program Development in Java - Abstraction, Specification, and Object-Oriented Design*, Addison Wesley, chapter 1 (2001). ISBN 0-202-65768-6.

[22] Lopes, C.: *D: A Language Framework for Distributed Programming*, PhD Thesis, College of Computer Science, Northeastern University (1997).

[23] Northeastern University: *Demeter/Java(DARPA supported EDCS project)*, Online publishing, URI http://www.ccs.neu.edu/home/lieber/Demeter-and-Java.html.

[24] Object Management Group, Inc: *CORBA*, Online publishing, URI http://www.corba.org/.

[25] Object Mentor: *JUnit.org*, Online publishing, URI http://www.junit.org/index.htm (2001).

[26] ObjectWeb Consortium: *The JAC Project*, Online publishing, URI http://jac.objectweb.org/index.html (1999).

[27] Orleans, D. and Lieberherr, K.: DJ: Dynamic Adaptive Programming in Java, *International Conference on Meta-level Architectures and S eparation of Crosscutting Concerns 2001 (Reflection 2001)*, Springer Verlag, pp. 73–80 (2001).

[28] Palo Alto Research Center: *AspectJ*, Online publishing, URI http://www.parc.com/research/csl/projects/aspectj/.

[29] Pawlak, P., Seinturier, L., Duchien, L. and Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java, *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns 2001 (Reflection 1997)*, LNCS 2192, Springer, pp. 1–24 (2001).

[30] Popovici, A., Gross, T. and Alonso, G.: Dynamic Weaving for Aspect-Oriented Programming, *Aspect-Oriented Software Development 2002 (AOSD 2002)*, ACM Press, pp. 141–147 (2002).

[31] Sato, Y., Chiba, S. and Tatsubori, M.: A Selective, Just-In-Time Aspect Weaver, *Generative Programming and Component Engineering 2003 (GPCE 2003)*, LNCS 2830, SV, pp. 189–208 (2003).

[32] Soares, S., Laureano, E. and Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ, *Object-Oriented Programming*

Systems, Languages, and Applications 2002 (OOPSLA 2002), ACM SIGPLAN Notices, pp. 174–190 (2002).

[33] Sun Microsystems, Inc: *Reflection*, Online publishing, URI http://java.sun.com/j2se/1.4.1/docs/guide/reflection/.

[34] Sun Microsystems, Inc: *Enterprise JavaBeans Technology*, Online publishing, URI http://java.sun.com/products/ejb/ (1995).

[35] Sun Microsystems, Inc: *Java Remote Method Invocation (RMI)*, Online publishing, URI http://java.sun.com/products/jdk/rmi/ (1995).

[36] Tarr, P., Ossher, H., Harison, W. and Jr, S. M. S.: N degrees of separation: multi-dimensional separation of concerns, *International Conference on Software Engineering 1999 (ICSE 1999)*, IEEE Computer Society Press, pp. 107–119 (1999).

[37] Tatsubori, M., Sasaki, T., Chiba, S. and Itano, K.: A Bytecode Translator for Distributed Execution of Legacy Java Software, *European Conference on Object-Oriented Programming 2002 (ECOOP 2002)*, LNCS 2072, Springer, pp. 236–255 (2001).

[38] Tilevich, E. and Smaragdakis, Y.: J-Orchestra: Automatic Java Application Partitioning, *European Conference on Object-Oriented Programming 2002 (ECOOP 2002)*, Springer (2002).

[39] University of Twente: *ComposeJ*, Online publishing, URI http://trese.cs.utwente.nl/prototypes/composeJ/ (1999).

[40] Wohlstadter, E., Jackson, S. and Dvanbu, P.: DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems, *International Conference on Software Engineering 2003 (ICSE 2003)*, IEEE Computer Society Washington, DC, USA, pp. 174–186 (2003).