# Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph

Takashi Ishio, Shinji Kusumoto, Katsuro Inoue
Graduate School of Information Science and Technology,
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
{t-isio, kusumoto, inoue}@ist.osaka-u.ac.jp

## Abstract

*Aspect-Oriented Programming (AOP) introduces a new software module unit named aspect to encapsulate crosscutting concerns. While AOP modularizes crosscutting concerns to improve maintainability and reusability, AOP introduces a new factor of complexity. It is difficult to find defects caused by an aspect modifying or preventing the behavior of other objects and aspects. In this paper, we examine a method to support a debugging task in aspect-oriented software development. We propose an application of a call graph generation and program slicing to assist in debugging. A call graph visualizes control dependence relations between objects and aspects and supports the detection of an infinite loop. On the other hand, program slicing shows the user changes of dependence relations caused by aspects. We implement a program-slicing tool for AspectJ and apply it to certain programs. The experiment illustrates how our approach effectively helps developers understand the influence of aspects in a program.*

## 1. Introduction

Aspect-Oriented Programming (AOP) proposes a new module unit, or *aspect*, for encapsulating crosscutting concerns such as logging and synchronization [17]. In Object-Oriented Programming, program code implementing crosscutting concerns is normally scattered among objects related to the concerns. In AOP, one crosscutting concern can be written in a single aspect. AOP improves maintainability and reusability of objects and aspects.

The goal of Aspect-Oriented Programming (AOP) is to separate concerns in software. While the hierarchical modularity of object-oriented languages are extremely useful, they are inherently unable to modularize crosscutting concerns, such as logging and synchronization. AOP provides language mechanisms that explicitly capture the crosscutting structure. Encapsulating the crosscutting concern as a module unit *aspect*, which is easier to develop, maintain and reuse is possible. Aspects separated from an object-oriented program are composed by *Aspect Weaver* to construct the program with a crosscutting structure.

In AspectJ, an aspect represents a crosscutting concern as a set of *advices*. An advice is a method-like unit consisting of a procedure and a condition used to execute the procedure. The condition of an advice execution is specified by a *pointcut*. A pointcut is defined by a subset of *join points*, which are well-defined events during program execution, such as method calls and field accesses. Using join points, a programmer can separate crosscutting concerns from objects. Various applications of AOP have been reported [16, 12, 21].

Although AOP is useful, it introduces a new complexity into a program. Since an aspect modifies the behavior of objects, a programmer must inspect objects and related aspects to understand system behavior; otherwise, the programmer may inject a defect, which is hard to detect, such as accidental advice executions and inter-aspect problems. A typical inter-aspect problem occurs when two aspects prevent each other's behavior, while each aspect behaves correctly when the aspect stands alone [19]. Early detection of inter-aspect problems is crucial to support the debugging tasks.

In this paper, we propose an application of a call graph generation and program slicing to support a debugging task for aspect-oriented software development. A call graph is a directed graph whose vertices and edges represent methods and method call relations, respectively. We add advice vertices and advice execution relations into a call graph for detection of infinite loops and accidental advice executions. On the other hand, *program slicing* is a very promising approach to localize faults in a program [25]. By definition,

program slicing is a technique which extracts all statements that may possibly affect a certain set of variables in a target program. We extend a *DC slicing* [18], which is a program-slicing method based on static and dynamic dependence relations in a program.

We implement a call graph calculation and a program-slicing tool as an Eclipse [6] plug-in. When a programmer runs a program and finds the incorrect value of a variable using a debugger, he/she calculates a program slice based on the variable to find the statements which have affected the incorrect value.

We conduct two experiments to evaluate the tool. In one experiment, we apply the tool to certain programs and show that program slicing visualizes changes of dependence relations caused by aspects. In the other experiment, we have students debug a program of AspectJ using a program slice. As a result, we show program slicing is appropriate for the debugging of aspect-oriented programs.

The structure of this paper is as follows: in Section 2, we present a brief overview of Aspect-Oriented Programming. In Section 3, we describe infinite loop detection using a call graph. In Section 4, we present an extension of program slicing for an aspect-oriented program. In Section 5, we evaluate the proposed method and discuss experimental results. In Section 6, we conclude our discussion with remarks regarding plans for future work.

## 2. Aspect-Oriented Programming

### 2.1. Features of Aspect-Oriented Programming

Aspect-Oriented Programming is an improved programming paradigm based on the other module mechanisms such as procedural programming and Object-Oriented Programming. In OOP, an object implements a part of the system's functionality. Objects interact with each other via messages (or method calls) to achieve the system's goal. While the hierarchical modularity of object-oriented languages is extremely useful, they are inherently unable to modularize crosscutting concerns, such as logging and synchronization. Since such concerns are implemented as an interaction of related objects, program code must be scattered among objects. Scattered code causes the following problems:

- When a specification of a crosscutting concern is changed, programmers must modify all related objects.
- Programmers cannot reuse an object independently of other objects since objects are connected to each other with a crosscutting concern.
- Programmers cannot reuse implementation of a concern independently of objects. If another set of objects interacts in the same way, programmers must re-implement the concern.

```
class SomeClass {
  public void doSomething(int x) { ... }
}

aspect LoggingAspect {
  before(): call(void SomeClass.doSomething(..)) {
    Logger.logs(thisJoinPoint);
} }

aspect ParameterValidationAspect {
  before(int x):
    args(x) && call(void *.doSomething(..)) {
    if ((x < 0)||(x > Constants.X_MAX_FOR_SOMETHING)) {
      throw new RuntimeException("invalid parameter!");
} } }
```

**Figure 1. Aspect examples: Logging and parameter checking**

AOP introduces a new module unit named 'aspect' to encapsulate a crosscutting concern. In AOP, one concern can be written in a single aspect. An aspect consists of some advices. An advice is a method-like unit consisting of a procedure and a condition used to execute the procedure. A condition to execute an advice is specified by a pointcut. A pointcut is defined by a subset of join points, which are well-defined events during program execution, such as method calls and field accesses. Using join points, a programmer can separate crosscutting concerns from objects. Modularized crosscutting concerns have good maintainability and reusability.

A part of available join points are shown as follows:

- A method call to an object,
- A method execution of an object after dynamic binding,
- A field access of an object, and
- Exception handling.

Advices are linked to objects by three types of forms: *before* (immediately before join points), *after* (immediately after), and *around* (replacement of join points). Advices can access runtime context information, for example, a called object, a caller object, and parameters of a method call.

A sample code of aspects is shown in Figure 1. *LoggingAspect* logs a method call to *SomeClass.doSomething*. *ParameterValidationAspect* validates all method calls whenever the method name is *doSomething*, and throws an exception if the validation fails. In this example, when the specification of the parameter validation is changed, programmers change only the aspect instead of all callers of *doSomething*. On the other hand, both aspects are executed when *SomeClass.doSomething* is called. In such a case, a

compiler (or an interpreter) serializes advices being executed. In AspectJ, programmers write the precedence of aspects to adjust the execution sequence of advices.

Various applications of AOP have been reported. In OOP, design patterns are design components describing how objects should interact [8]. Since an interaction of objects is a kind of crosscutting concern, programmers can write a pattern as an aspect. Aspects implementing design patterns are reusable components [12]. On the other hand, it is also useful for applications to support debugging and to write crosscutting concerns in a distributed software environment [16, 21].

### 2.2. Complexity of Aspects

Although AOP is useful, AOP introduces new complexity as follows:

**(a)** Multiple advices may be executed at the same join point. An execution sequence of advices may affect the result of calculation. An example is the program in Figure 1. When the parameter validation aspect throws an exception, the output depends on whether or not the logging aspect is executed before the parameter validation.

**(b)** An advice may be activated during another advice execution. In such a case, an aspect may change the behavior of another aspect. When two advices are activated during an execution of each other advice, the advices cause an infinite loop.

**(c)** An incorrect pointcut definition causes accidental advice executions. It is impossible to predict the behavior of an advice accidentally executed.

**(d)** In the software evolution process, a pointcut definition may become obsolete according to the changes of objects.

Problems (a) and (b) are known as inter-aspect issues [19]. A research regarding how to solve such issues exists [5]. Problems (c) and (d) are partially supported by the Integrated Development Environment in AspectJ [4]. For problem (d), aspect-aware refactoring is proposed since aspects may conflict with refactoring techniques in OOP [11].

Detecting inter-aspect problems and accidental advice executions is difficult since aspect interference is required in certain cases. For example, *ParameterValidationAspect* must validate method calls in other aspects. Therefore, we focus on the debugging defects caused by aspects instead of focusing on the safe composition rules of aspects [15]. We propose a debugging support based on a call graph and program slicing. Program slicing focuses on the problems (a) and (c), and call graph construction focuses on the problem (b).

## 3. Loop Detection using Call Graph

Carelessly defined, incorrect pointcuts cause accidental advice executions. Accidental advice executions are hard to detect since a programmer who inspects a code fragment is hard to recognize whether or not the code fragment is modified by aspects when the fragment is viewed in isolation [22]. A typical result of an incorrect pointcut is an infinite loop [3]. Infinite loops should be statically detected in a compilation process instead of runtime.

A call graph is a simple way to visualize advice executions and to detect an infinite loop. A call graph is a directed graph representing the calling relationships between the program's methods [9]. When a cyclic path from a vertex $v$ to the vertex $v$ itself exists, the path represents a candidate of an infinite loop.

We use a simple extension of a call graph for AOP. We treat an advice as a method in the same way AspectJ compiles an advice into a standard Java method [14]. When a join point specified by a pointcut of an advice exists in a method body, we regard the advice execution as a method call from the method to the advice. We construct a call graph whose vertices and edges represent methods and advices, and method call relations and advice execution relations, respectively. If a path from a vertex $v_m$ corresponding to a method $m$ to a vertex $v_{adv}$ corresponding to an advice $adv$ exists, the advice $adv$ may be called during the execution of the method $m$.

A key point of the call graph construction is how to handle control flow that is dynamically determined in AOP. In AspectJ, such control flow is caused by the polymorphic methods of objects and the dynamic pointcut designators of aspects. In order to resolve such dynamic elements, we construct a call graph in the following steps.

First, the class hierarchy of a program is extracted from source code. This class hierarchy includes the method lookups modified by the inter-type declaration of the aspects in the program[22].

Next, we construct a method call graph whose vertices and edges represent methods and method calls, respectively. We resolve a polymorphic method as follows: When the method `m` of the class `c` overrides the method `m` defined in the superclass `d` and the method `n` calls `d.m`, a method call edge from $v_n$ to $v_{c.m}$ and another edge from $v_n$ to $v_{d.m}$ are connected.

Finally, advices vertices and advice call edges are added to the graph. Dynamic pointcut designators such as `cflow` and `if` are dynamically checked in the program execution. We regard *join point shadows*[14] which may trigger an advice execution as an advice call. `before` and `after` advices are simply replaced to method calls as shown in Figure 2. An `around` advice is handled in another way since an `around` advice replaces join points. Figure 3 shows that

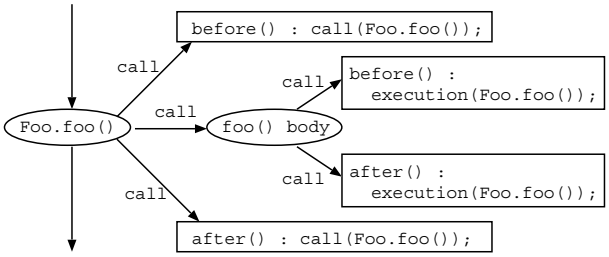


**Figure 2. Before advice call and after advice call handled as method calls**

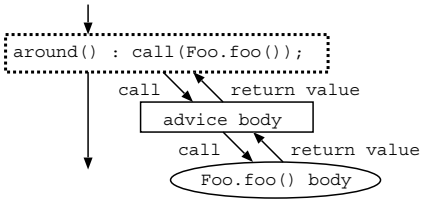


**Figure 3. Around advice call handled as a method call**

an `around` advice replace a method call. When a keyword `proceed` exists in an `around` advice, the keyword represents an original join point replaced by the advice. We regard the `proceed` keyword as a method call relation from the proceed to the original join point when the join point is a method call.

A call graph approach is easier to implement than other approaches such as formal techniques. Both a framework to detect inter-aspect dependence relations [5] and a framework to allow programmers to manually control advice executions exist [19]. However, these approaches cannot detect accidental aspect dependence relations which are not inter-aspect relations. A call graph visualizes all aspect dependence relations in a program.

We can detect candidates of infinite loops from a call graph based on depth first search [23]. Figure 4 shows an example of a call graph. An ellipse vertex represents a method, and a rectangle vertex represents an advice. All edges represent a method call relation. The program represented by the call graph consists of three classes: *Main*, *Counter*, *AnotherCounter*, and two aspects, *Foo* and *Bar*. The aspect *Foo* counts a method call of *Main.getX()* using a *Counter* object. The aspect *Bar* logs the result of *Main.getX()*. Any vertices are not explicitly connected to vertices representing constructors since these objects and aspects are created by static initializers when the Java Virtual Machine loads classes. In the graph of Figure 4, a cyclic

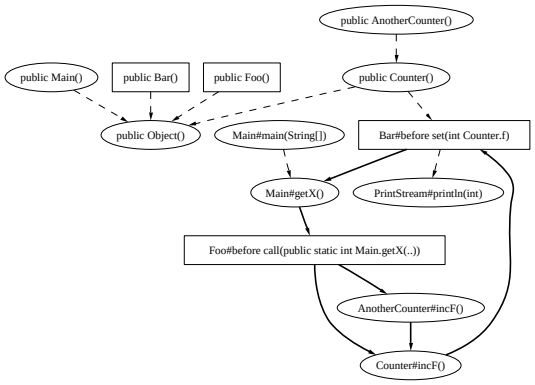


**Figure 4. A call graph**

path including the vertex corresponding to the advice *before(): call(Main.getX)* is represented by bold line edges. The cyclic path is an infinite loop.

Programmers can confirm that inter-aspect dependence relations are their intentional result. Programmers detect accidental dependence relations and remove errors of a control flow. Our tool is implemented for call graph construction and cycle detection. Since a call graph grows proportionally to program size, automatically extracting cycles and dependence relations between aspects is important. The implementation details are described in Section 5.1

## 4. Program Slicing

*Program slicing* is a promising approach for program debugging, testing, and understanding [25]. Given a source program $p$, a *program slice* is a collection of statements possibly affecting the value of *slicing criterion* (in the pair $<s$, $v>$, $s$ is a statement in $p$, and $v$ is a variable defined or referred to at $s$). We extend program slicing to an aspect-oriented program to aid in a debugging task.

### 4.1. A Slice Calculation Algorithm

A program slice is calculated through the following three phases:

**(a)** Extract dependence relations in a target program,

**(b)** Construct a program dependence graph, and

**(c)** Traverse a graph.

Phase (a) is an extraction of dependence relations. Program slicing is based on data and control dependence relations. A *data dependence relation* is a relation between an assignment and a reference of a variable. When all of

the following conditions are satisfied, we say that a data dependence relation from statement $s_1$ to statement $s_2$ by a variable $v$ exists:

1. $s_1$ assigns a value to $v$, and
2. $s_2$ refers to $v$, and
3. At least one execution path from $s_1$ to $s_2$ without redefining $v$ exists. We call this condition *reachable*.

The above definition is a *static* data dependence relation. A *dynamic* data dependence relation is extracted when the value assigned at a statement $s_1$ has reached to a reference statement $s_2$ during program execution.

On the other hand, a *control dependence relation* is a relation between a conditional statement and a controlled block. Consider statements $s_1$ and $s_2$ in a source program $p$. When all of the following conditions are satisfied, we say that a control dependence relation, from statement $s_1$ to statement $s_2$ exists if:

1. $s_1$ is a conditional predicate, and
2. The result of $s_1$ determines whether $s_2$ is executed or not.

A dynamic control dependence relation is extracted when a statement $s_2$ is executed after a conditional predicate $s_1$ is evaluated during program execution.

Phase (b) is a construction of a *program dependence graph*. The nodes of a graph represent statements of a program, and directed edges represent data and control dependence relations. In Phase (c), a program slice is calculated by backward traversal of the program dependence graph from a slicing criterion.

We choose DC slicing from three slicing methods: static slicing, dynamic slicing and DC slicing. These slicing methods are classified by a method how to extract dependence relations. Static slicing is used for program understanding and verification [25] since static slicing analyzes source codes of a program to extract the possible behaviors of the program. Dynamic slicing analyzes a program execution with a certain input data. Since a dynamic slice includes statements actually executed, dynamic slicing is used to support a debugging task [1]. In DC slice calculation, the dynamic data dependence analysis is performed during program execution, and the information of dynamically determined elements is collected. Control dependence relations are statically extracted from the source code since a high cost is required to analyze control dependence relations during program execution. DC slicing requires a reasonable cost for the calculation of practical programs [18]. Therefore, our approach is based on DC slicing.

```
class Count {

  public static void main(String[] args) {
    if (args.length == 0) return;

    Counter counter;
    boolean isIncrementCounter = false;
    if (args[0].equals("inc")) {                    (a)
      counter = new IncrementCounter();
      isIncrementCounter = true;
    } else if (args[0].equals("sft")) {
      counter = new ShiftCounter();
    } else return;

    for (int i=0; i<3; ++i) counter.proceed();      (b)
    String result = Integer.toString(counter.value());
    System.out.println(result);                     (c)
}

abstract class Counter {
  private int count = 1;
  public Counter() {}
  public int value() { return count; }              (d)
  public void proceed() { count = newValue(count); }
  abstract protected int newValue(int old);
}

class IncrementCounter extends Counter {
  protected int newValue(int old) { return old + 1; }   (e)
}

class ShiftCounter extends Counter {
  protected int newValue(int old) { return old << 1; }
}
```

**Figure 5. DC slice example**

An example of a DC slice for Java is shown in Figure 5. In this program, an instance of the class *IncrementCounter*, and an instance of the class *ShiftCounter*, output their value. The input parameter of this program determines which counter is used. A slice with input "inc" and slicing criterion $(c)$ is indicated by rectangles $(a), \cdots, (e)$ in Figure 5. When a program results in an invalid output, programmers choose the variable which contains the output as a slicing criterion, and calculate a slice to localize a fault.

## 4.2. Extension of Program Slicing for Aspect-Oriented Program

We extend program slicing to an aspect-oriented program to aid in a debugging task. We assume that a target program has no infinite loops since a programmer has already removed infinite loops using a call graph. Therefore, our approach focuses on a debugging task to remove a defect detected by a test case. In the debugging process, a programmer first runs a test case to collect dynamic information. Next, a programmer activates a tool to construct a program dependence graph. Finally, a program slice is calculated by user-specific slice criteria. A programmer can localize a fault using the program slice.

Program slicing for aspect-oriented languages has already been proposed, but has not been implemented and evaluated yet [26]. In this paper, we choose AspectJ as a target language and extend program slicing from Java to AspectJ. In this basic idea, which is the same as a call graph extension, we regard an advice execution as a method call.

Data dependence relations and control dependence relations in advices are the same as program slicing for OOP. Features of program slicing introduced for AOP are following:

**Join point information:** An advice can access runtime context information such as a caller object and parameters of a method call. We regard such information as parameters passed to the advice from the join point. In order to access context information, AspectJ provides `thisJoinPoint` object. The method `thisJoinPoint.getArgs(int)` is prepared for accessing parameters. Since the parameter of the method call to `getArgs` is determined in runtime, the caller of `getArgs` is handled as references to all parameters of the method of the join point. The other context properties such as the method signature and `this` object are regarded as a reference to a parameter passed to the advice from the join point.

**A pointcut reference:** An advice depends on a pointcut definition. A program slice includes a pointcut definition when a corresponding advice is included in the slice. Since a pointcut determines an advice execution, we connect a control dependence edge from a pointcut to an advice.

**A dynamic pointcut:** Dynamic context sometimes determines whether or not an advice is executed. Since a program slice should include all statements which may affect a slicing criterion, the slice always includes statements which may have been affected by advices which use a dynamic context. Static analysis can reduce a slice based on a call graph [20]; however, this is a subject for future work.

**An advice call relation:** The idea of handling an advice call is same as the case of the call graph construction. A vertex corresponding to a join point shadow is regarded as a caller vertex of the advice.

### 4.3. Dynamic Analysis

In the DC slice calculation process, dynamic information of a target program is required. Dynamic information consists of dynamic data dependence relations and dynamic binding information. Dynamic analysis, a process collecting such information, is also a crosscutting concern. Therefore, we implement a dynamic analysis aspect in AspectJ. This aspect is based on the dynamic analysis aspect for Java [16]. Programmers link the aspect to the target program to extract dynamic information.

A dynamic analysis aspect collects dynamic information as follows:

- **Data Dependence Relation**

  **When a new value is set to a field:** The aspect logs a signature of the field and the position of the assignment statement.

  **When a field is referred to:** The aspect receives the position of the last assignment to a field, and logs a data dependence relation from the assignment to the reference.

- **Polymorphism Resolution**

  **When a method is called (before call):** The aspect pushes the method signature and the position of calling into a call stack prepared for each thread of control.

  **When a method is invoked (before execution):** The aspect checks the top of the call stack, and generates a call edge from the caller to the actually-invoked method.

  **After a method call:** The aspect removes the top of the call stack.

  **When an exception is thrown:** The aspect removes the top of the call stack.

Since aspects cannot access local variables in AspectJ, we analyze intra-method dependence relations statically and inter-method dependence relations dynamically. As a result, our slice becomes larger than a complete DC slice. To calculate a complete DC slice, intra-method dependence relations need to be extracted dynamically. Though dynamic information is effective to distinguish objects and to extract inter-method dependence relations, dynamic intra-method dependence relations are less effective[16]. When a dynamic analysis aspect conflicts with other aspects in the target program, conflicts are solved by precedence declaration in AspectJ and by static analysis using a call graph.

## 5. Implementation and Evaluation

We have implemented a call graph calculation and a program slicing tool. In order to evaluate our tool, we have conducted two experiments. In one experiment, we have applied our tool to the AspectJ source code of design patterns [13], and evaluated how program slicing works for the aspect-oriented programs. In the other experiment, we have evaluated how program slicing affects the debugging task. We have measured the working time of a debugging task using a program slice.

In Section 5.1, we describe the implementation overview of our tool. We present the former experiment in Section 5.2 and the latter experiment in Section 5.3.

## 5.1. Implementation Overview

Programmers repeatedly modify source code and run test cases in their debugging process. Integrated Development Environments provide tools which support debugging tasks. Our tool should be used with other tools such as a debugger, an incremental compiler, and a customized editor in the IDE. For example, when programmers find a location where the value of a variable is incorrect, they can then calculate a slice based on the variable.

We have chosen Eclipse [6] for the platform of our tool. Eclipse is an open source IDE, and programmers can write a plug-in in Java to add new functionalities to the IDE. We have implemented the tool as an Eclipse plug-in based on Java and AspectJ development plug-ins. The size of our plug-in is about 5,000 lines of code.

Eclipse plug-ins handle important events in the IDE, for example, saving files and completion of compilations. We have used such event handlers to implement the plug-in. When compilation succeeds, our slice plug-in extracts static information from a source code and constructs a call graph. If the call graph contains a cyclic path, notification is shown by a dialog. On the other hand, we have integrated our tool to the editor provided by AspectJ plug-in. Our tool allows programmers to specify a slice criterion on the source code editor and shows a program slice by underlining on the editor.

In order to analyze AspectJ source code, our tool collects the information from AspectJ Development Tools plug-in [4]. On the other hand, we can apply a DC slicing tool for Java byte-code [24] since the current version of AspectJ compiler generates Java byte-code [2]. However, when we use the tool for Java byte-code, we need to preserve a mapping from AspectJ source to Java byte-code. Preserving such a mapping is difficult since pointcut information and join point shadows are not expressed in Java byte-code. We have chosen an approach to analyze AspectJ source code instead of Java byte-code. Since we have implemented the tool extracting information from the compiler, our tool does not handle run-time weaving.

## 5.2. Experiment 1: Evaluation of Program Slicing

We have conducted an experiment to evaluate how program slicing works for aspect-oriented programs. We have applied our tool to the AspectJ source code of several design patterns [13]. We have used five patterns in Table 1 since programmers can effectively implement these patterns using AspectJ [12]. We describe the result of the slicing in Section 5.2.1, and discuss analysis costs in Section 5.2.2.

**Table 1. Target codes**

| Name | Size (LOC) |
|---|---|
| ChainOfResponsibility | 517 |
| Observer | 667 |
| Singleton | 375 |
| Mediator | 401 |
| Strategy | 465 |

### 5.2.1 Evaluation of Slicing Result

First, we construct a call graph based on the source code of five design patterns. As a result, the call graph consists of 179 vertices and 240 edges, and a subgraph including a loop is extracted from the graph. The only one loop included by the subgraph is a recursive call of the method *recieveRequest* defined by *ChainOfResponsibilityProtocol* aspect. We can easily see that the loop is just a recursive call since the loop consists of the method vertex and a recursive call edge.

Next, we executes five programs of design patterns and calculates program slices based on the variable which contains the output of the program. Program slicing is effective for tracking inter-module dependence relations. In this experiment, a program slice is calculated based on a certain variable in an aspect for each design pattern. In order to get the same information as the slice, programmers must track definitions of methods and advices manually, and this task requires much time since programmers must track several files which affect the variable. One design pattern is usually defined as a set of aspects, an abstract defining the structure of the pattern, and as concrete aspects declaring the actors of the pattern. For example, an Observer pattern is defined as one abstract aspect named *ObserverProtocol*, and two concrete aspects: *ColorObserver* and *CoordinationObserver*. *ObserverProtocol* contains code on how Observer objects and Observed objects interact. *ColorObserver* and *CoordinationObserver* declare that *Screen* objects act as observers, and that the color and the coordination of *Point* objects are observed. Programmers must inspect two classes and three aspects to track the location where the value of the variable originates.

An advantage of program slicing is the visualization of codes and dependence relations modified by aspects. Figure 6 shows a slice including an aspect replacing a method call. Such a method replacement aspect is used in unit testing and temporary implementation. Recognizing a change of dependence relations by such aspects is difficult because aspect definitions are usually separated from class definitions. AspectJ Development Tools (AJDT) plug-in [4] provides an extended editor, which shows the locations where the advice is executed. However, since AJDT cannot visual-

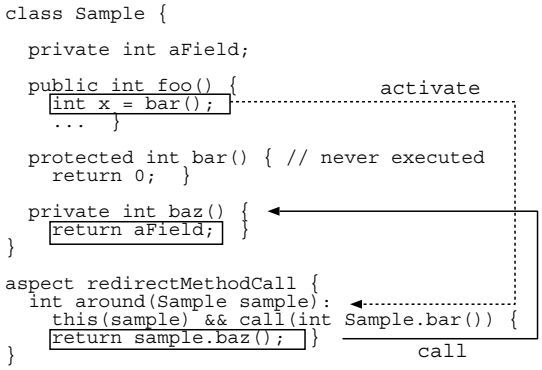


**Figure 6. A slice including an aspect replacing a method call**

**Table 2. Time cost of dynamic analysis (seconds)**

| Target | Normal Execution | Execution with Dynamic Analysis |
|---|---|---|
| ChainOfResponsibility | 3.76 | 3.93 |
| Observer | 0.32 | 0.37 |
| Mediator | 3.21 | 5.69 |
| Singleton | 0.14 | 0.32 |
| Strategy | 0.18 | 0.22 |

ize statements replaced by advices, programmers must carefully consider *around* advices replacing original join points.

#### 5.2.2 Evaluation of Time and Space Requirements

Analysis cost for the program slicing consists of the following costs: the cost of static dependency extraction in compile time, the cost of dynamic dependency extraction in run time and the cost of slice calculation for traversal a program dependence graph.

Static analysis is implemented by a traversal of an abstract syntax tree (AST), constructed by an AspectJ compiler. Although only a rough estimation, the traversal process is proportional to the size of the AST, and AST size is proportional to the size of the target program. The time cost required to analyze 10,000 lines of code-implementing design patterns is 14.7 seconds. The cost accounts for 17 percent of 85.5 seconds, the total compilation time.

The time cost of dynamic analysis is shown in Table 2. The overhead of our tool for dynamic analysis is acceptable. The time cost of slice calculation is proportional to the size of a program dependence graph.

**Table 3. Test cases of the program**

| Input Expression | Output Value | Output String |
|---|---|---|
| (* (+ 5 3) (+ 5 5)) | 80 | (* (+ 5 3) (+ 5 5)) |
| (* (+ 5 4) (+ 5 4)) | 81 | (* (+ 5 4) ) |
| (+ (+ 4 2) 5 (* 4 2)) | 19 | (+ (+ 4 2) 5 (* )) |

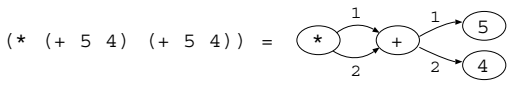


**Figure 7. A graph representing an expression**

Memory cost usually depends on the size of a target program. Aspects also affect memory cost since aspects complicate program dependence relations. For example, the memory cost required to analyze the design patterns is about 20MB.

In order to test scalability, we have constructed a program dependence graph of AspectJ compiler version 1.1.1 [2]. The size of the compiler is about 60,000 lines of code (without unit tests). Since our tool requires about 200MB memory to compile and to analyze, the scalability of the tool is achievable by decomposing a system to the subsystems. On the other hand, when a program slice becomes larger and crosscutting many classes and aspects, a programmer cannot track a slice using a normal text editor. Examining how a large program slice should be shown to a user is a future work. We are going to investigate the decomposition of a program slice into small pieces using concept analysis [7].

### 5.3. Experiment 2: The Debugging Task

In order to evaluate how program slicing influences a debugging task, we compare the working time of debugging between students using a program slice, and students working without a program slice. Twelve graduate students of computer science attended the experiment. They have had experienced with Java but not with AspectJ. Therefore, we prepared the preliminary tasks in order to allow students to get used to Eclipse and AspectJ. We conducted the experiment using the following steps: First, we explained to the information science students about Java and Eclipse, and gave them the task of debugging a small Java program (Task 1). Next, we explained about AspectJ and Aspect-Oriented Programming, and gave the student the task of writing an aspect (Task 2). Finally, we gave them the task of debugging an AspectJ program (Task 3).

We prepared a small AspectJ program for Task 3. The

**Table 4. Time required for debugging task (minutes)**

| Group | Task 1 | Task 2 | Task 3 |
|---|---|---|---|
| 1. works without the slice | 150 | 186 | 200 |
| 2. works with the slice | 200 | 210 | 190 |

program processes an expression consisting of literals and two kinds of operators: adders, and multipliers. An expression is defined by a graph whose nodes are terms of the expression. An example of a graph representing $(* \ (+ \ 5 \ 4) \ (+ \ 5 \ 4))$ is shown in Figure 7. The program contains the following aspects:

**Print Aspect** constructs a string representation of a graph and outputs the string when the program is finished.

**Loop Detection Aspect** observes a traversal of a graph to detect whether or not the graph has a cyclic path. When the aspect detects a loop, the aspect throws an exception.

**Caching Aspect** caches a value of the nodes and prevents re-evaluation of shared nodes.

**Graph Destruction Aspect** destructs a graph. When a node is evaluated, the node is removed from the graph.

Task 3 included debugging a program which contained a bug. We have prepared a bug caused by an aspect preventing the behavior of another aspect. When the Caching Aspect omits a re-evaluation of the nodes, the aspect omits a process of the Print Aspect too. Sample inputs and outputs for the program are shown in Table 3. Since inputs are internally represented as a graph, shared nodes are omitted in the output.

We gave the students the correct output, and a short explanation for each aspect. We randomly chose half of the students, and gave them a program slice. The program slice was calculated based on an output variable in the Print Aspect. The program slice indicates that the output variable is affected by the Caching Aspect and the Print Aspect, but is not affected by the other two aspects.

We collected information on how the students modified the program and on how long it took them to fix the bug. We asked the students whether or not a program slice was useful for debugging using a program slice.

Table 4 shows the average time the students took for the tasks. Although students who used a program slice showed a better performance time than the students who did not use the slice, no statistically intentional difference exists.

Students who used a program slice reported that a program slice was a good guideline for reading a program and was useful for excluding source code not related to a bug since a program slice visualizes all dependence relations including indirect impact by aspects. However, students also reported that they needed to inspect the entire program to confirm whether or not a modification of an aspect impacts other modules. Therefore, we conclude that combining program slicing with impact analysis or other techniques to support a bug-fixing task is important.

In summary, we cannot say that program slicing is quantitatively effective to bug-fixing of AOP. However, according to the students' opinions, it is very useful to localize a fault in AOP. It is important to combine program slicing with other techniques to support bug-fixing tasks, especially for AOP, in the reduction of debugging time.

The conclusion of the experiment is limited since only one group works with program slicing and another group works without the technique. The result is affected by the distinction of the programming ability of each group.

## 6. Conclusion

In this paper, we have proposed an application of program slicing to support a debugging task for aspect-oriented programs.

A key feature of Aspect-Oriented Programming is the separation of crosscutting concerns. Programmers encapsulate an interaction between multiple objects into an aspect. Since crosscutting code is localized to a module, AOP improves maintainability and reusability.

An aspect modifies objects' behavior without modification of their code. If programmers change code without knowledge about classes and related aspects, programmers may inject a fault such as an accidental advice execution. Such bugs are difficult to detect; therefore, we propose a support using a call graph and program slicing. The call graph and program slicing are already available for procedural programs and object-oriented programs. We have extended the call graph by regarding an advice execution as a kind of method call. We have also extended DC slicing based on the same idea.

We have implemented a call graph construction and a slice calculation tool as an Eclipse plug-in. We have conducted two experiments to evaluate the tool. In one experiment, we applied the tool to certain programs and showed that program slicing visualizes changes of dependence relations caused by aspects. In the other experiment, we had students debug a program of AspectJ using a program slice. As a result, program slicing effectively showed aspect dependence relations to a developer. Program slicing was also effective in localizing faults.

For future work, we extend our research on debugging support using impact analysis. We will also apply a reflection analysis based on dynamic analysis [10]. Finally,

we will also examine how a large program slice should be shown to a user.

## Acknowledgement

This work is partly supported by the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology.

## References

[1] Agrawal, H. and Horgan, J.: Dynamic Program Slicing. *SIGPLAN Notices*, Vol.25, No.6, pp.246-256, May (1990).

[2] AspectJ Team: AspectJ Project. http://eclipse.org/aspectj/

[3] AspectJ Team: *The AspectJ Programming Guide*. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/

[4] AspectJ Team: *AspectJ Development Environment*. http://www.eclipse.org/ajdt/

[5] Douence, R., Motelet, O., and Südholt, M.: A Framework for the Detection and Resolution of Aspect Interactions. *In Proc. of GPCE 2002*, pp.173-188, October (2002).

[6] Eclipse Project. http://www.eclipse.org/

[7] Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code, *IEEE Transactions on Software Engineering*, SE-29(3), pp.210-224, (2003).

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

[9] Groove, D., Furrow, G., Dean, J. and Chambers, C.: Call Graph Construction in Object-Oriented Languages. *In Proc. of OOPSLA 1997*, pp.108-124, October (1997).

[10] Gschwind, T., Oberleitner, J. and Pinzger M.: Using Run-Time Data for Program Comprehension. *In Proc. of IWPC 2003*, pp.245-250, May (2003).

[11] Hanenberg, S., Oberschulte, C. and Unland, R.: Refactoring of Aspect-Oriented Software. *In Proc. of NODe 2003*, pp.19-35, September (2003).

[12] Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ. *In Proc. of OOPSLA 2002*, pp.161-173, November (2002).

[13] Hannemann, J.: *Aspect-Oriented Design Pattern Implementations*. http://www.cs.ubc.ca/~jan/AODPs/

[14] Hilsdale, E. and Hugunin, J.: Advice Weaving in AspectJ. *In Proc. of AOSD 2004*, pp.26-35, March (2004).

[15] Ichisugi, Y., Tanaka, A. and Watanabe, T.: Extension Rules: Description Rules for Safely Composable Aspects. *Technical Report of the National Institute of Advanced Industrial Science and Technology (AIST)*, AIST01-J00002-4, February (2003).

[16] Ishio, T., Kusumoto, S. and Inoue, K.: Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique. *In Proc. of IWPSE 2003*, pp.3-12, September (2003).

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming. *In Proc. of ECOOP 1997*, vol.1241 of LNCS, pp.220-242, June (1997).

[18] Ohata, F., Hirose, K., Fujii, M. and Inoue, K.: A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information. *In Proc. of APSEC 2001*, pp.273-280, December (2001).

[19] Pawlak, R., Seinturier, L., Duchien, L. and Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java. *In Proc. of REFLECTION 2001*, pp.1-24, September (2001).

[20] Sereni, D. and de Moor, O.: Static Analysis of Aspects. *In Proc. of AOSD 2003*, pp.30-39, March (2003)

[21] Soares, S., Laureano, E. and Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. *In Proc. of OOPSLA 2002*, pp.174-190, November (2002).

[22] Störzer, M. and Krinke, J.: Interference Analysis for ApsectJ. *In Proc. of FOAL 2003*, pp.35-44, March (2003).

[23] Tarjan, R. E.: Depth first search and linear graph algorithms. *SIAM Journal on Computing*, Vol.1, No.2: pp.146-160, June (1972).

[24] Umemori, F., Konda, K., Yokomori, R. and Inoue, K.: Design and Implementation of Bytecode-based Java Slicing System. *In Proc. of SCAM 2003*, pp.108-117, September (2003).

[25] Weiser, M.: Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4), pp.352-357, (1984).

[26] Zhao, J.: Slicing Aspect-Oriented Software. *In Proc. of IWPC 2002*, pp.251-260, June (2002).