Study on Aspect Extraction and Program Analysis for Effective Software Development

Takashi Ishio

January 2006

Dissertation submitted to
Graduate School of Information Science and Technology, Osaka University
for the degree of Doctor of Philosophy in Information Science and Technology

Abstract

Aspect-Oriented Programming is proposed to improve modularity for crosscutting concerns. In aspect-oriented programming, an aspect can capture crosscutting structure based on Join Point Model which abstracts an execution of program as a sequence of join points. A join point represents a well-defined event during program execution such as method call, method execution, field access and exception handling. A concern is called as a crosscutting concern when the related code fragments of the concern spread across (crosscuts) the modules. A crosscutting concern is hard to maintain since a developer needs to maintain all code fragments consistently and code fragments of multiple crosscutting concerns are often mixed at different places. Although such crosscutting concerns are found in many programs, rewriting all of them in aspect-oriented language is infeasible. Therefore, finding concerns for which aspect-oriented programming is suitable and effective is important. In this paper, we propose and discuss two aspects related to object-oriented software development.

1. Dynamic analysis recording program execution
2. Modularization of assertions crosscutting objects

Dynamic analysis is a process collecting execution history of a program. The collected information is used by debugger and several analysis tools since the information reflects the actual behavior of the program. Dynamic analysis needs to watch the behavior of objects in the program. A developer is required to implement a program transformation tool in order to achieve dynamic analysis. We have proposed to implement a dynamic analysis tool based on AOP to reduce efforts for the tool development. Comparing an approach using a customized compiler which generates code for dynamic analysis for program slicing, aspect-oriented analyzer improves modularity and maintainability of the analysis system.

Another development aspect is modularization of assertions crosscutting objects. Assertion checking is a powerful tool to detect software faults during debugging, testing and maintenance. Although assertion documents the behavior of

one component, it is hard to document relations and interactions among several objects since such assertion statements are spread across the modules. Therefore, we propose to modularize such assertion as an aspect in order to improve software maintainability. Taking *Observer pattern* as an example, we show a limitation of traditional assertion and effectiveness of assertion aspect through the case study, and discuss various situations to which assertion aspects are applicable.

Above two aspects become independent modules separated from the base-code, or the core functional part of a program, using aspect-oriented programming. Researchers in this area also proposed aspects for other purposes, e.g. implementing non-functional requirements in software system. These aspects show usefulness and effectiveness of aspect-oriented programming. However, aspect-oriented programming introduces two problems named "fragile base-code problem" and "inter-aspect problem". Fragile base-code problem is that an aspect may be broken when the base code is modified by a developer who is unaware of aspects since each aspect defined as a module collaborating with base-code has several assumptions on the base-code. Inter-aspect problem is caused when several aspects are working at the same time, and an aspect accidentally breaks assumptions of other aspects. Although the fragile base-code problem is partially supported with integrated software development environment and the aspect-aware refactoring support a developer to carefully modify base-code, the inter-aspect problem is not enough supported.

To tackle the inter-aspect problem, we have proposed and implemented an extension of program slicing for aspect-oriented programs including multiple aspects. Program slicing extracts a program dependence graph whose vertices represent program elements and edges represent relationship between vertices respectively. A developer can see inter-aspect relationship as a program slice. We have conducted an experiment debugging an aspect-oriented program including four aspects which are related to each other and causing inter-aspect problem. The result shows that program slicing effectively supports developers' debugging task.

Applications of Aspect-Oriented Programming show the effectiveness of aspect-oriented programming in various areas. Although Aspect-Oriented Programming brings new problems, developers can have the advantage of Aspect-Oriented Programming supported with program analysis techniques.

List of Publications

Major Publications

[1-1] Takashi Ishio, Shinji Kusumoto, Katsuro Inoue: Application of Aspect-Oriented Programming to Calculation of Program Slice. IPSJ Journal, Vol.44, No.7, pp.1709-1719, July 2003 (in Japanese).

[1-2] Takashi Ishio, Shinji Kusumoto, Katsuro Inoue: Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique. Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003), pp.3-12, Helsinki, Finland, September 2003.

[1-3] Takashi Ishio, Shinji Kusumoto, Katsuro Inoue: Debugging Support Environment for Aspect-Oriented Program Using Program Slicing and Call Graph. IPSJ Journal, Vol.45, No.6, pp.1522-1532, June 2004 (in Japanese).

[1-4] Takashi Ishio, Shinji Kusumoto, Katsuro Inoue: Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph. Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), pp.178-187, Chicago, Illinois, September 2004.

[1-5] Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: Aspect-Oriented Modularization of Assertion Crosscutting Objects. Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005), pp.744-751, Taipei, Taiwan, December 2005.

Related Publications

[2-1] Reishi Yokomori, Takashi Ishio, Tetsuo Yamamoto, Makoto Matsushita, Shinji Kusumoto, Katsuro Inoue: Java Program Analysis Projects in Osaka University: Aspect-Based Slicing System ADAS and Ranked-Component Search

System SPARS-J. Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pp.828-829, Portland, Oregon, USA, May 2003.

Acknowledgement

I am most indebted to my supervisor Professor Katsuro Inoue for his continuous support and supervision over the years. Without his help, experience and advice, this thesis would never have reached completion.

I am very grateful to Professor Shinji Kusumoto for valuable comments and helpful criticism on this thesis.

I would like to express my gratitude to Associate Professor Makoto Matsushita and Doctor Toshihiro Kamiya in National Institute of Advanced Industrial Science and Technology. Their comments, criticism and advice have helped guide and shape the development of this thesis.

I would also like to express my gratitude to Professor Kenichi Hagihara for his valuable comments on this thesis.

I would like to express my gratitude to all members of the Department of Computer Science for their guidance, especially Professor Toshimitsu Masuzawa.

Thanks are also due to many friends in the Department of Computer Science, especially students in Inoue Laboratory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) introduces a new module unit, or *aspect*, for encapsulating crosscutting concerns [34]. The goal of Aspect-Oriented Programming is to separate concerns in software. While the hierarchical modularity of object-oriented languages is extremely useful, they are inherently unable to modularize crosscutting concerns, such as logging and synchronization. These concerns usually affect various modules in the system.

AOP provides language mechanisms that explicitly capture the crosscutting structure. Encapsulating a crosscutting concern as an aspect improves modularity of the software system followed by good maintainability and reusability of the modules. Aspects separated from an object-oriented program are composed by *Aspect Weaver* to construct a program with a crosscutting structure.

AspectJ [80] is an extension of Java for aspect-oriented programming and it is one of the most famous and practical language in this area. 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 *pointcut* definition associated with the procedure. A procedure, or an advice body, is a code fragment written in Java. The advice body is executed when the condition specified by the associated pointcut is satisfied. A pointcut is defined as a subset of *join points*, which are well-defined events during program execution, such as method calls and field accesses. AspectJ and other AOP languages provide their own *Join Point Model* to define events which are available for a pointcut definition. For example, the join point model of AspectJ regards program execution as a sequence of events including method calls and field accesses. On the other hand, Ali et al. proposes a state-based join point model which regards program execution as a sequence of state transition of objects

```
after(): call(void Database.query(..)) {
  // calling a logging method here.
}
```

Figure 1.1: Logging example in AspectJ

caused by changing objects' fields [2].

Using join points, a developer can specify when the advice is executed independently of the implementation details of system functionality. One example is a rule such as "This logging advice is executed for each method call to `Database.query()`". In AOP, this rule removes the logging method calls from various callers since Aspect Weaver can automatically insert a logging method call for each query method call. The rule described in AspectJ is shown in Figure 1.1.

In traditional object-oriented programming, a developer has to consistently maintain the rule by hand. Whenever developers add or remove a database query, they have to update logging code. This is a hard task since the logging code spreads across the modules and the logging code may tangle with other similar code fragments which contribute other concerns. AOP allows developers to easily distinguish the aspects and manage rules since a rule is explicitly defined as an aspect and the code is automatically inserted based on the rules.

AOP improves maintainability and reusability of objects and aspects since AOP separates crosscutting structures which usually consists of method calls to infrastructural modules such as logging and synchronization from core functionality of objects. Separated crosscutting structure is modularized as aspects, and core functionalities in the system are represented as objects without crosscutting structure. In traditional programming, reusing a component needs some additional tasks which include removing application specific crosscutting structure from the component and adding crosscutting code for new application. In AOP, a developer easily reuse a component for another system since crosscutting structure is already separated.

Since AOP is widely applicable, the usage of aspect is categorized into three groups listed below [36].

Development Aspects support developers' task in the development process. This category includes logging aspect for debugging and profiling for performance improvement. These aspects are removed when the system is released.

Production Aspects contribute system functionality. An example is user authentication aspect which requires a password at an appropriate time.

Runtime Aspects improve performance of a system without changing the behavior of the system. Optimization and memory management are included in this category.

In this paper, we have tried to modularize development aspects for effective software development since software systems often include development aspects independently of the systems' domains. We discuss two aspects closely related to program analysis techniques to support various software development tasks.

1. Dynamic analysis aspect recording program execution
2. Aspect modularizing assertions crosscutting objects

One is dynamic analysis collecting execution history of a program. The collected information is used by debugger and several analysis tools since the information reflects the actual behavior of the program. Dynamic analysis is a variant of logging since dynamic analysis also records information of the program execution. In order to achieve dynamic analysis, a developer is required to implement analysis tool using meta-programming techniques. Proposal of this paper is to implement a dynamic analysis tool based on AOP to reduce efforts for tool development and maintenance.

Another is modularization of assertions crosscutting objects. Assertion checking is a powerful tool to detect software faults during debugging, testing and maintenance [62]. Assertion usually describes the behavior of one component and improves the robustness of the component. However, it is hard to document relations and interactions among objects since such assertion statements are spread across the modules. Such crosscutting assertion often damages encapsulation of objects followed by poor maintainability and reusability. Therefore, we propose to modularize such assertions as an aspect in order to improve modularity and usability of them.

Both two development aspects are closely related to program analysis techniques since dynamic analysis is widely used in program analysis, and assertion is often used as a description of a component's property in program analysis. In the following section, first we describe program analysis and relationship between program analysis and the development aspects. Effectiveness of these aspects is shown in Chapter 2 and Chapter 3.

Although the above aspects and other aspect proposed by researchers are effective, AOP brings two problems named "fragile base-code problem" and "inter-aspect problem". These problems are caused by aspects which are loosely coupled with a base program via join point model. Therefore, we propose a program analysis extended for AOP in order to support developers' debugging task. We explain

these problems in Section 1.4 and debugging support for these problems in Chapter 4 respectively.

1.2 Program Analysis

Developers need to understand the components for maintaining software system. Software maintenance activities span a system's productive life and can account for as much as 80 percent of the total effort expended on a software system [61]. Developing complex but high quality software efficiently is one of important goals of software engineering research [57]. *Program Analysis* is a research area to support program understanding by extracting some properties from a program [29]. Program analysis abstracts a program as:

- a graph whose vertices represent program entities and edges represent relationship between entities respectively, or
- a metric value representing some software properties including complexity, reusability and maintainability, or
- other models.

A developer can choose an analysis method to focus on an interesting perspective of a program. Extracting a graph representing a program is a typical abstraction tool in program analysis. Various graphs are used as follows.

- Abstract syntax tree represents source code.
- Control flow graph represents the execution order of statements in a procedure.
- Call graph represents caller-callee relationship among procedures.
- Program dependence graph represents control flow and data flow in a program.

Calculating metric values from a program are also effective approach for abstracting software properties. Examples are listed below.

- The number of classes, methods and lines of code are popular measures of software size.
- Similarity value extracted from a pair of code is a hint to find duplicated code.

Some graphs and metric values are derived from others. For example, a program dependence graph is extracted from an abstract syntax tree, cyclomatic complexity [46] of a procedure is calculated from the control flow graph of the procedure, respectively. A combination of properties often allows developers to evaluate properties of software and to compare software with each other.

There are two approaches to extract such information from software. One approach is *static analysis* which analyzes products including source code and related documents. Since static approach extracts software properties without execution of the software, the result contains all possible behavior of a program. Another is *dynamic analysis* which analyzes execution process of a program with a input. Dynamic analysis extracts detailed information about software execution process, including dynamic binding and exception handling, which are hardly extracted by static analysis. Various applications of program analysis exist as follows.

Graph-based applications

Code optimization: The automatic code optimization removing unreachable operations in the control flow graph of a procedure during compilation process is an application of static analysis. A dynamic analysis is profiling performance of a program by measuring time consumed for each procedure. The result of profiling shows a bottle-neck of a program.

Test case generation: Static analysis generates various input parameters for a procedure to cover as much lines of code as possible in the procedure based on the control flow graph and conditions which are satisfied at each assertion statement. Dynamic analysis reveals statements uncovered by given test cases. In this purpose, both approaches are combined to generate enough test cases for a procedure.

Debugging support: Program slicing is proposed to extract code related to variables specified by a developer [74]. This technique uses a program dependence graph reflecting both the control flow and the data flow in a program [27]. Related statements are identified by reachability in a program dependence graph. Although the original approach is based on static analysis, dynamic information is also used to analyze detailed information for one test case [1, 68].

Impact analysis: Impact analysis reveals code related to statements modified by a developer. This method is often regarded as a variant of program slicing since this method extracts statements affected by modified statements based on a program dependence graph. While this method

is usually implemented using static analysis to predict the effect of software modification [3], dynamic impact analysis is also proposed to indicate procedures affected by software modification [4, 43].

Invariant detection and propagation: A component has invariants, or special conditions always satisfied about its state during program execution. While a developer usually describes such invariants using assertion statement, finding additional invariants is practical and important problem [56]. Invariants are used to several purposes, e.g. predicting incompatibilities in component upgrades [47].

Program understanding: Abstracting program source code is useful to understanding the structure of the program. A call graph of a program and a document of a module interface including assertion are well-known examples. Visualization of execution history dynamically extracted from a program is also useful to understand the behavior of the program. These techniques are important for software maintenance and debugging.

Applications of numerical values

Evaluation of software quality: Software metrics measuring software quality are proposed to evaluate software quality including complexity, cohesion and coupling of a module [8, 46]. Static analysis is suitable for this purpose since most of metrics measures software products. Dynamic analysis is used to analyze runtime behavior, e.g. the complexity of communication among objects [75].

Similarity of source code: This is an application of static analysis to find similar code fragments in a system in order to support software maintenance. The similarity between two source files may be measured by the cosine measure when a source file is abstracted as a vector. Similar components are found by clustering components based on metrics including size and complexity [49].

Similarity of software behavior: This is an application of dynamic analysis for invasion detection and program understanding. One method to calculate similarity of the behavior of a program is the cosine measure of two vectors representing the call frequency of each method [60]. In an invasion detection system, if similarity between usual behavior and current behavior of program is under a threshold, the invasion detection system can stop the program execution since the program is executing unknown functions.

Program understanding: Software metrics supports developers to understand a program since developers can categorize modules based on the metrics. The metric values are useful to decorate a graph to indicate exceptional entities in the graph [12].

Program analysis research area includes various techniques and applications. Dynamic analysis implementation is an important crosscutting concern since dynamic analysis is one of two categories of program analysis. Assertion is also important concern since it provides useful information for program analysis which supports debugging, testing and program understanding.

1.3 Development Aspects

Dynamic analysis usually crosscuts a program to be analyzed. Assertion crosscutting objects is not always, but often observed too. This section explains traditional approaches to handle these concerns and the differences between traditional approach and aspect-oriented approach.

1.3.1 Dynamic Analysis

Dynamic analysis analyzes execution process of a program. The result of dynamic analysis depends on both the software and the input for the software. Dynamic analysis extracts detailed information about software execution process, including dynamic binding and exception handling, which are hardly extracted by static analysis.

A dynamic analysis tool requires extracting dynamic information from a target program during its execution. Most of dynamic analysis methods model an execution as a sequence of events including method calls and returns since an execution history of a program is usually too huge to analyze even if the granularity of the history is method-level [59].

How to collect the execution history is a design issue for developers implementing a dynamic analysis tool. Although inserting a logging method call into the beginning and the end of each method is a simple implementation, automatic insertion is required since inserting such code by hand is infeasible. The following list shows approaches implementing automatic insertion.

Customized Compiler approach modifies a compiler to generate a program which includes additional code recording the execution history. Although this approach is expensive to develop a new compiler and to maintain the compiler

according to the update of the language specification, it can be applied to fine-grained analysis.

Additional Preprocessor approach inserts analysis code to a target program during the preprocess before compilation. This approach is easier than customized compiler.

Object Code Translator approach adds analysis code to a target program during the translation process. Since the translator manipulates object code already compiled without analyzing source code, the translation rule may be simpler than the preprocessor approach. However, the result might be affected by code optimization.

Customized Debugger approach executes a program in a debugger, an emulator or an execution environment which extract information from the execution process. This approach enables an analysis without the modification of a target program. If the integration of the method with a debugger is important, this approach is suitable.

Java Virtual Machine Tool Interface [85] approach is similar to the customized debugger approach since the Java Virtual Machine is one execution environment. An advantage of this approach is very inexpensive to implement a tool since developers have to implement only a monitoring tool which records the data from the Java Virtual Machine. Although this approach depends on Java Virtual Machine specification, this interface is convenient.

The above approaches are usually a pair of the process which inserts code and the inserted code. The inserting process is required to be updated when the specification of a target language is updated. If a pair of the process and the inserted code is strongly coupled, updating such dynamic analysis tool is a hard task since the developer needs to update all rules included in the dynamic analysis correctly.

The proposal of this paper is modularizing dynamic analysis as an aspect using AOP. This approach stands on a point of view to divide the method and the analysis module in order to make the analysis module reusable and easy-to-maintain. Aspect-oriented approach is similar to using a generic translator for a target language and a translation program. However, such a generic translator approach uses a meta-programming language to describe syntactic translation for code insertion. Aspect-oriented approach uses the base level language to write an aspect. A semantic translation by an aspect weaver is based on a join point model which abstracts the dynamic behavior of a program [32].

1.3.2 Assertion

Assertion checking is powerful, practical, scalable and simple to use. Assertion is effective to detect software faults during debugging, testing and maintenance [62]. Assertion also supports developers in understanding the software because it documents the behaviors of a component and effectively prevents developers from depending on implementation details of the component [47].

Practical programming languages such as Java, C and C++ have `assert` as a language construct, a function of the standard library, or a macro of a preprocessor. Assertion describes a condition which should be true such as consistency of data structure and the range of a variable in a complex algorithm. The behavior of `assert(expr)` statement is shown as follows.

assert(true)	$\rightarrow$	do nothing
assert(false)	$\rightarrow$	throw a runtime exception

Design by Contract [50] is one of the most famous approaches to design a component using assertion checking effectively. Design by Contract improves robustness of software by specifying the behavior of a component based on *preconditions* and *postconditions* for each method of components. Preconditions specify the range of a parameter and the state of a component which must be satisfied before the method of the component is used. Postconditions specify the range of a result value and the state of a component satisfied after the method call. In short, preconditions protect the called component from illegal calls, and postconditions protect the caller against erroneous implementations, respectively [64]. Design by Contract also defines *class invariants* as common pre/post-conditions for all methods.

A simple way to implement Design by Contract is inserting appropriate assertion statements into the beginning and the end of the method in order to check preconditions and postconditions for the method. While Eiffel and other several languages provide language constructs to directly support Design by Contract, several behavioral specification languages and tools are proposed for existing language. For example, Larch [21] family includes a tool for C++. JML [87] and jContractor [30] are proposed for Java. They provide language constructs or class library to represent first order logic in order to improve expressiveness of assertions. The pre/post-conditions for each method are written in the comment of each method or in an external module.

We show an example of `pop` operation of a stack component in Figure 1.2. This method fails with `IndexOutOfBoundsException` at the operation of `implementation.remove(-1)` when a user calls this method for an empty stack. However, the user cannot judge the failure is caused by a defect of the

```
java.util.Vector implementation;

public Object pop(Object o) {
  // Return a last element of a vector.
  return implementation.remove(implementation.size()-1);
}
```

Figure 1.2: Pop operation of a stack component

```
requires !this.isEmpty();
ensures \result ==
          \old(this.implementation.lastElement());
ensures this.implementation.size() ==
          \old(this.implementation.size()-1);
```

Figure 1.3: Contract for pop operation of a stack component

stack or a misuse of the stack since there are no explicit information about the responsibilities of the stack.

Figure 1.3 is the contract for the pop method written in JML. This example fragment is a simplified version of the specification of java.util.Stack provided by JML Project [88]. This contract explicitly prohibits the user from calling the pop method for an empty stack. Advantages of Design by Contract are shown as follows.

- The contract throws an exception when the method is called for an empty stack instead of when the `remove` statement is executed. Stopping a program before its execution for erroneous input contributes to keep the integrity of important data structure.

- A user can easily judge the cause of a failure. The failure is caused by a misuse of a component when its precondition is not satisfied. The failure is caused by a defect of a component when an exception is thrown by the component even if the user satisfies the precondition of the method.

- Contract is a part of reliable interface document of a component. While the external documentation of a component tends to become obsolete during the software evolution process, a user can check whether the contract is valid or not at runtime of the program.

Assertions usually document the behavior of one component. Although assertion has great ability to represent constraints for a component, it is hard to handle and document properties held in interactions among objects since such assertion statements are spread across the modules.

An example which shows a limitation of traditional assertion is a constraint of "one subject-to-many observers" relationship for Observer pattern [16]. A developer can reuse the usual Observer pattern which models many-to-many relationship for an observer pattern instance with one-to-many constraint since many-to-many implementation covers one-to-many usage. In this case, the developer needs to write assertions in order to prevent an observer from being attached to several subjects. However, it is a hard task for the developer to describe such assertions for that purpose in a modularized manner since an observer has no variable which represents how many subjects the observer attaches to. So the developer needs to add a field containing an attached subject to Observer and modify Subject to check and update the field when an observer is attached. The scattered code damages modularity and maintainability of the components.

The proposal of this paper is modularizing crosscutting assertions in an aspect. Since assertion is useful information for developers and program analysis tools, improving modularity of assertion is important.

1.4 Complexity of Aspect-Oriented Programming

Although aspect-oriented modularization is extremely useful, AOP brings two problems named "fragile base-code problem" and "inter-aspect problem". This section explains these problems caused by aspects which are loosely coupled with a base program via join point model.

1.4.1 Fragile Base-Code Problem

Component developers unaware of extensions to the component developed by its users may produce a seemingly acceptable revision of a base class which may damage its extensions [52]. This problem is known as *fragile base-class problem* since an extension of a base class means a subclass in object-oriented programming. In aspect-oriented programming, this problem is called as fragile base-code problem since the code in a system is categorized into two parts, base-code, or classes representing core functional code, and aspects extending the base code.

A typical example of the problem is that renaming classes, methods or fields influences matching semantics of call, execution, get/set and other pointcuts when the names of classes, methods or fields are used in pointcut declaration [40]. A

developer is required to update aspects after the developer modified base-code.

To tackle this problem, Pointcut Delta Analysis [40] and Aspect-Aware Refactoring [22] are proposed. Pointcut Delta Analysis lists the difference of join points between base code and new revision of the base code for developers. Aspect-Aware Refactoring adds the update of relevant pointcuts to usual refactoring process in order to ensure that a developer updates pointcuts. Although these approaches partially solve the problem, further research is needed in this area.

1.4.2 Inter-Aspect Problem

Inter-aspect problem is caused by multiple aspects [55].

(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 a pair of Logging aspect and Parameter Validation aspect. When a method is called, Logging aspect outputs the method name, and Parameter Validation checks the input parameter and throws an exception if the parameter is invalid. 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 overwrite the data of the base code in order to implement an aspectual behavior. For example, an Encryption aspect which encrypts all strings in a database system damages keyword search function in the database since the keyword search compares keywords and strings in the database but the keywords are incomparable with the encrypted strings [7].

(c) 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.

Douence et al. takes a formal approach for problem (a) to check whether the aspects are independent on each other or not [13]. A program analysis for aspect-oriented program proposed and discussed in this paper focuses on problem (b) and (c).

1.5 Research Overview

In this research, we have discussed effectiveness of two development aspects modularizing dynamic analysis and assertion crosscutting objects. Although these aspects and other aspect proposed by researchers [58, 70] are effective to improve

software modularity and maintainability, fragile base-code problem and inter-aspect problem caused by aspects might reduce the maintainability of software. Therefore, we have proposed an extension of program slicing for aspect-oriented programming in order to support software maintenance, and have shown effectiveness of program slicing through an experiment of debugging an aspect-oriented program including multiple aspects which are causing inter-aspect problem. Although Aspect-Oriented Programming brings new problems, developers can have the advantage of Aspect-Oriented Programming supported with program analysis techniques.

1.5.1 Dynamic Analysis for Program Slicing Using Aspect-Oriented Technique

Chapter 2 describes effectiveness of modularizing dynamic analysis implementation. In a case study, a program slicing system which uses dynamic information including method calls and inter-procedural data flow is developed. The dynamic analysis aspect written in AspectJ achieved better maintainability and readability than the dynamic analysis tool developed by traditional approach.

1.5.2 Modularization of Assertions Crosscutting Objects

In Chapter 3, effectiveness of modularizing assertion is discussed. Although assertion is well-known practical tool to improve robustness and predictability of components, assertion spreads across components when a developer specifies inter-component properties. The assertion crosscutting objects damages the encapsulation of the components. Taking Observer pattern with an inter-component constraint as an example, we have compared an aspect-oriented assertion module using a simple aspect-oriented language with a traditional implementation. The result shows that the aspect-oriented approach improves modularity of the system and that the approach is promising for various situations.

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

Chapter 4 describes a development support based on program slicing technique extended for aspect-oriented program. Although aspect-oriented programming is useful, inter-aspect problem and fragile base-code problem are risky for developers when the developers introduce AOP into their product process.

There is not enough development support for AOP yet since several tools used in object-oriented programming is not directly applicable to AOP. For example,

Lopes et al. pointed out that unit-testing tool like JUnit [89] cannot be applied to AspectJ unit-testing [73].

Program slicing is a basic program analysis technique which is applicable to various tools including debugging support, impact analysis and program understanding. Therefore, we have extended program slicing for aspect-oriented programming by defining a rule to convert an aspectual behavior to an object behavior to handle aspects in traditional program slicing method. We have implemented the program slicing tool and conducted an debugging experiment. As a result, the program slicing for AOP is useful for developers to debug an AOP program including multiple aspects.

Chapter 2

Dynamic Analysis for Program Slicing Using Aspect-Oriented Technique

2.1 Introduction

Aspect-oriented programming aims to modularize crosscutting concerns in aspects. Since dynamic analysis recording program execution affects over all target program, dynamic analysis is a typical crosscutting concern. In this chapter, we apply aspect-oriented programming to develop a program slicing tool using dynamic analysis in order to show the effectiveness of aspect-oriented programming.

Program slicing is a very promising approach to localize faults efficiently in a program [74]. By definition, program slicing is a technique which extracts all statements that may possibly affect a certain set of variables in a program. The set of all extracted statements is called a program slice.

In recent software development, a programmer uses not only procedural languages like C and Pascal but also object-oriented languages like Java [18] and C++ [67]. Object-oriented programs have many dynamically-determined elements since object-oriented languages introduce new concepts including *class*, *inheritance*, *dynamic binding* and *polymorphism* [6]. In the slice calculation process, such dynamic elements affects the size of a program slice since a program slice should include all statements which are possibly executed. Dynamic analysis observing program execution is effective to remove statements which are not actually executed. *Dependence-Cache (DC) slicing* combines dynamic data dependence analysis and a static control dependence analysis in order to calculate accurate slices with lightweight costs [5, 68]. Ohata et al. extend DC slicing method for

object-oriented languages [53].

To implement DC slice calculation tool for Java, how to analyze dynamic data dependence is an important issue. A dynamic analyzer observes a target program to collect information about dynamic data dependence. Such dynamic analysis cannot be encapsulated in a single module in traditional programming language. Instead, inserting dynamic analysis code into a target program has been implemented as a pre-processor [53], or as a customized Java Virtual Machine (JVM) [38]. However, maintaining rules inserting code is difficult in the former approach since the rules are implemented in complex meta-programming, the latter approach is expensive to maintain the system since JVM must be re-customized when new versions are released.

On the other hand, aspect-oriented programming proposes a new module unit, or *aspect*, for encapsulating crosscutting concerns such as logging and synchronization [34]. Modularizing dynamic analysis for DC slicing in an aspect seems a natural usage of AOP since the dynamic analysis is a crosscutting concern. However, effectiveness of AOP is not clear. In this chapter, we introduce AOP for encapsulating dynamic analysis into an aspect to examine its effectiveness. We have implemented a DC slice calculation system using AspectJ [80], and conducted an experiment to evaluate the usefulness of our approach compared with a customized JVM approach. As a result, we confirmed that the AOP approach can greatly reduce the effort required for implementing dynamic analysis.

The structure of this paper is as follows: Section 2.2 describes the DC slicing. Section 2.3 presents a brief overview of Aspect-Oriented Programming and our approach to DC slice calculation using AOP. In Section 2.4, the implementation of DC slicing tool is presented. Section 2.5 evaluates the proposed method, compares our method with the customized JVM approach and discusses about experimental results. Section 2.6 summarizes the results and discussions.

2.2 Program Slicing

Program slicing is a promising approach for program debugging, testing, and understanding [74]. 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). Although many slice calculation algorithms have already been proposed, we use a *program dependence graph* (PDG) in this research [54].

2.2.1 Program Dependence Graph

A PDG is a directed graph whose nodes represent statements in a source program, and whose edges denote dependence relations (data dependence or control dependence) between statements. An edge drawn from node V_s to node V_t represents that node V_t depends on node V_s. PDG also includes special nodes which represent method call and parameter passing [71].

Control dependence and data dependence are defined as follows.

Control Dependence **(CD)** 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 (CD)*, 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.

This relation is written by $CD(s_1, s_2)$.

Data Dependence **(DD)** When all of the following conditions are satisfied, we say that a *data dependence (DD)*, from statement s_1 to statement s_2 by a variable v, exists if:

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 re-defining v exists (we call this condition *reachable*).

This relation is denoted by $DD(s_1, v, s_2)$.

The program slicing calculation consists of the following four phases:

Phase 1: Defined and Referred Variables Extraction
We identify defined variables and referred ones for each statement in a source program.

Phase 2: Data Dependence Analysis and Control Dependence Analysis
We extract data dependence relations and control dependence relations between program statements.

Phase 3: Program Dependence Graph Construction
We construct a PDG using dependence relations extracted in Phase 2.

Phase 4: Slice Extraction

We calculate the slice for slicing criterion specified by a user. In order to calculate the slice for a slicing criterion $<s, v>$, PDG nodes are traversed in reverse order from V_s corresponding to statement s. The program slice is the statements corresponding to the nodes which are reachable to the criterion node via control/data dependence path in PDG.

We can obtain sufficient information about control dependence from static analysis. However, in static analysis, information about data dependence contains a redundant part because we analyze all execution paths, including paths which may be never executed. If we use program slicing for debugging and program understanding, analyzing detailed information about one program execution path with a specific input is effective. Dependence Cache (DC) slicing has been proposed to realize such a requirement [5, 53, 68].

In DC slice calculation, the data dependence analysis is performed during program execution, and the information of dynamically determined elements is collected. Control dependence is analyzed statically from the source code since a high cost is needed to analyze control dependence during program execution. DC slicing requires a reasonable cost for the calculation of practical programs.

2.2.2 Dynamic Data Dependence Analysis for DC Slicing

When a value is assigned to variable v at statement t and the value of variable v is referred to at statement s, dynamic data dependence (DD) relation about v from t to s can be extracted if we can resolve v's defined statement t. We create a *Cache Table* that contains all variables in a source program and the most-recently defined statement information for each variable. When variable v is referred to, we extract a dynamic DD relation about v using the Cache Table. The following steps show the extraction algorithm for dynamic DD relations.

Step 1: We create a cache $C(v)$ for each variable v in a source program.
$C(v)$ represents the statement which most recently defined v.

Step 2: We execute a source program and conduct the following processes on each execution point.
In executing statement s,

- when variable v is referred to, we draw a DD edge from the node corresponding to $C(v)$ to the node corresponding to s about v, or
- when variable v is defined, we update $C(v)$ to s.

```
1: a[0] = 0;
2: a[1] = 1;
3: a[2] = 2;
4: a[3] = 2;
5: a[4] = 2;
6: read(c);
7: b = a[c] + 5;
```

Figure 2.1: Example program using array

Table 2.1: Cache transition of Figure 2.1

Statement number executed	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	b	c
1	1	-	-	-	-	-	-
2	1	2	-	-	-	-	-
3	1	2	3	-	-	-	-
4	1	2	3	4	-	-	-
5	1	2	3	4	5	-	-
6	1	2	3	4	5	-	6
7	1	2	3	4	5	7	6

For example, Figure 2.1 is a program using an array. Table 2.1 shows the transition of cache $C(v)$ of each variable v at each statement when the program is executed with input $c = 0$.

The table becomes $C(a[0]) = 1$, $C(a[1]) = 2$, $C(a[2]) = 3$, $C(a[3]) = 4$, $C(a[4]) = 5$ and $C(c) = 6$ when statement 6 is executed. When variable $a[0]$ is referred to at statement 7, data dependence $DD(statement1, a[0], statement7)$ is extracted because statement 7 refers to $a[0]$ and $C(a[0]) = 1$.

Figure 2.2 shows an example of the DC slice. This DC slice with input = "inc" and slice criteria $= (d)$ is the part contained in rectangles $(a)..(f)$ of Figure 2.2.

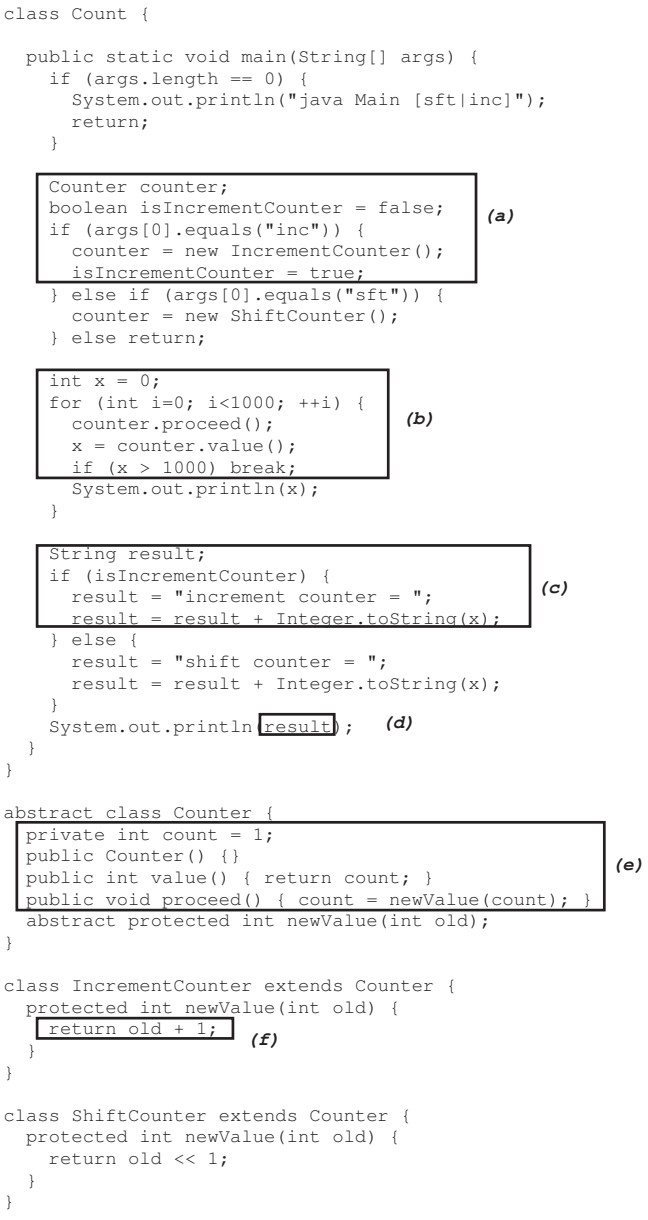

Figure 2.2: Source program and DC slice example (slice criteria = (d), input = "inc")

Table 2.2: Pointcut Designators of AspectJ

type of join point	representations
call	A method or a constructor is called.
execute	An individual method or a constructor is invoked.
get	A field of object is read.
set	A field of object is set.
handler	An exception handler is invoked.

2.3 Dynamic Analysis Using Aspect-Oriented Programming

The DC slice calculation requires dynamic program information. Although various ways exist in implementing the dynamic analysis, each way requires a high cost in implementation or in runtime.

2.3.1 Aspect-Oriented Programming

The goal of Aspect-Oriented Programming (AOP) is to separate concerns in software. While the hierarchical modularity of object-oriented languages is 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.

AspectJ is an aspect-oriented extension for Java [80]. AspectJ provides language constructs to write aspects. *Join points* are well-defined points in the execution of the program. The programmer chooses collections of join points as *pointcuts*, and defines a method-like construct named *advice*, which is an additional behavior at the join points. Examples of join points which programmer can use are shown in Table 2.2. Advice can be united by three types of forms, *before* (immediately before join points), *after* (immediately after), and *around* (before and behind).

The AspectJ compiler is an Aspect Weaver, which composes objects and aspects at source code level. AspectJ generates normal Java code, which includes aspects. Since AspectJ knows where an aspect is built in, The AspectJ compiler

```
aspect LoggingAspect {

  pointcut AllMethodCalls():
    !within(LoggingAspect) &&
    call(* *.*(..));

  pointcut MethodExecs():
    !within(LoggingAspect) &&
    execution(* somepackage.*.*(..));

  static Stack callStack = new Stack();
  static JoinPoint lastCall = null;

  Object around(): AllMethodCalls() {
    callStack.push(thisJoinPoint);
    lastCall = thisJoinPoint;
    proceed(); // execute original call
    lastCall = callStack.pop();
  }

  before(): MethodExecs() {
    if (lastCall != null) {
      Logger.logs("executed",
        lastCall.getSignature(),
        lastCall.getSourceLocation(),
        thisJoinPoint.getSignature(),
        thisJoinPoint.getSourceLocation());
    }
  }
}
```

Figure 2.3: The aspect which records dynamic bindings

can generate codes accessing the contextual information such as the signature of methods and the position in the source code for each join point.

2.3.2 Example of an Aspect

Here, an aspect which records dynamic bindings is shown in Figure 2.3, as an example of the aspect. This code records how dynamic bindings are resolved. Whenever a method is called, it records a signature of the method to be invoked and actually executed.

On one hand, if the aspect is not available, we have to insert code which records method invocation to the overall the program. On the other hand, since we can use a character "*" for pattern matching with a class name or a method name in AspectJ, an aspect becomes a small and simple module.

2.3.3 Dynamic Analysis of Program Execution

Dynamic analysis is used for various program purposes [1, 42, 53, 60, 75]. In the past, the following methods of dynamic analysis have been used for Java programs:

(a) Using a preprocessor to insert analysis operations into the target program [53].

(b) Using *Java Virtual Machine Profiler Interface* (JVMPI) to collect dynamic information [42].

(c) Using Java Debugger Interface (JDI) [84] to collect dynamic information.

(d) Using customized *Java Virtual Machine* for dynamic analysis [38].

In method (a), the preprocessor and conversion rules on an abstract syntax tree are made to insert operations for analysis in the target program. However, making generic conversion rules is difficult because of complex language factors. For example, a code fragments for logging should be generated for each method call appeared in a program. However, it is a complex meta-programming since multiple method calls can be appeared in one expression. On the other hand, problems of maintainability and reusability of a preprocessor exist, as well as conflict with other preprocessors. Therefore, implementing and maintaining the preprocessor is costly.

In (b), JVMPI is used to observe program execution. JVMPI is an interface of JVM used for profiling the CPU and for memory usage. JVM makes it possible to collect detailed events on program execution (e.g. method call, thread control, memory allocation and garbage collection). However, an overhead that JVM generates the events is expensive. Also, an analyzer using JVMPI must process events which are asynchronously generated. When an analyzer causes an error, both the analyzer and the JVM are aborted. Therefore, debugging the analyzer itself is difficult.

In (c), JDI is used to observe program execution. JDI is an interface with libraries used to implement a debugger. A program using JDI communicates with the Java Virtual Machine Debugger Interface (JVMDI) of JVM, which executes a program being debugged. JVMDI is a similar interface to JVMPI. A debugger program can set breakpoints, receive events such as field accesses and method calls, and receive stack frame information at each breakpoint. However, a debugger communicates with a target JVM by a socket, and frequently blocks the execution of a program to get information from the JVM. Consequently, JDI requires a high runtime cost. Although using JVMDI directly is possible, similar problems to the JVMPI approach arise.

(d) is a method that customizes JVM to observe and analyze program execution. An advantage of this method is that JVM can access all information in a Java runtime environment. However, JVM customization depends on its implementation. Whenever a new version of JVM is released, it must be re-customized.

In the AOP approach, a dynamic analysis aspect can be composed based on a join point model, which is more abstract than conversion rules for syntax tree. The aspect approach achieves good modularity, maintainability and reusability. The approach also achieves complex handling of control elements, such as multi-threading and exception in a well-organized way. Moreover, AspectJ composes the source codes of objects and aspects, and does not depend on implementation of a specific JVM. Since a program linked to the aspect becomes a standard Java program, debugging the aspect using a small test program and a debugger for Java is easy.

2.3.4 Dynamic Analysis Using AspectJ

In AspectJ, an aspect can access contextual information, e.g., a position of a join point, the signature of a method being called or the field being accessed. The dynamic analysis aspect can be written using this feature of AspectJ.

An algorithm of the data dependence analysis and polymorphism resolution based on AspectJ join point model is described as follows.

Data Dependence Analysis

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

When a field is referred to: The aspect receives the statement information of the last assignment to the 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.

```
public aspect DataDependsAnalysisAspect {

  pointcut target():
    !within(slice.aspect.*);

  pointcut exclude():
    within(somepackage.*);

  pointcut field_set():
    target() && !exclude() &&
    (set(* *) || set(static * *));

  pointcut field_get():
    target() && !exclude() &&
    (get(* *) || get(static * *));

  FieldDef def = new FieldDef();

  before(): field_set() {
    def.put(
      thisJoinPoint.getTarget(),
      thisJoinPoint.getSignature(),
      thisJoinPoint.getSourceLocation());
  }
  before(): field_get() {
    SourceLocation setpos =
      def.get(thisJoinPoint.getTarget(),
              thisJoinPoint.getSignature());
    Logger.logDataDepends(
      thisJoinPoint.getTarget(),
      thisJoinPoint.getSignature(),
      setpos,
      thisJoinPoint.getSourceLocation());
  }
}
```

Figure 2.4: A piece of the implementation of dynamic data dependence analysis

A piece of code where the dynamic data dependence analysis is implemented is shown in Figure 2.4. A polymorphism resolution is a multi-threaded extension of the code shown in Figure 2.3.

The dynamic analysis aspect uses a wildcard of AspectJ to analyze all assignments and references for each field. In this implementation, we can add the aspect into the target program without any changes of the aspect. If we do not want to analyze certain classes in the target program, writing a new sub-aspect inherited from the dynamic analysis aspect is possible.

The aspect keeps the original behaviors of the program. When the aspect is linked into the program, the control flow and the data flow are modified. However, since the aspect only reads data of the program without modifying such data, the data flow is not affected. Also, the aspect handles objects using *weak reference* so as not to affect the lifetime of objects. On one hand, weak reference is an available mechanism in Java, which does not prevent the weak-referenced object from being collected as garbage. On the other hand, since the control flow that is simply modified by the aspect may cause an infinite loop, an effort which prevents causing a loop is required. We will discuss this issue in Section 2.4.4.

2.4 Implementation

2.4.1 DC Slicing Tool

We have implemented a dynamic analysis module using AspectJ, and have then developed a DC slice calculation system for Java. Figure 2.5 illustrates the system overview.

Using this system, a user can calculate a DC slice through the following steps:

Step 1: The AspectJ compiler weaves the target Java program and the dynamic analysis aspect.

Step 2: The program is executed as usual in a Java program. The dynamic analysis aspect in the program generates a file containing dynamic information of the program execution.

Step 3: The DC slice calculation tool is executed with the source code of the target program and a dynamic information file which is generated by Step 2. The tool extracts static information from the source code, constructs PDG, and then opens a window of a source code viewer.

Step 4: The slice criterion is specified and the DC slice is viewed via a graphical user interface.

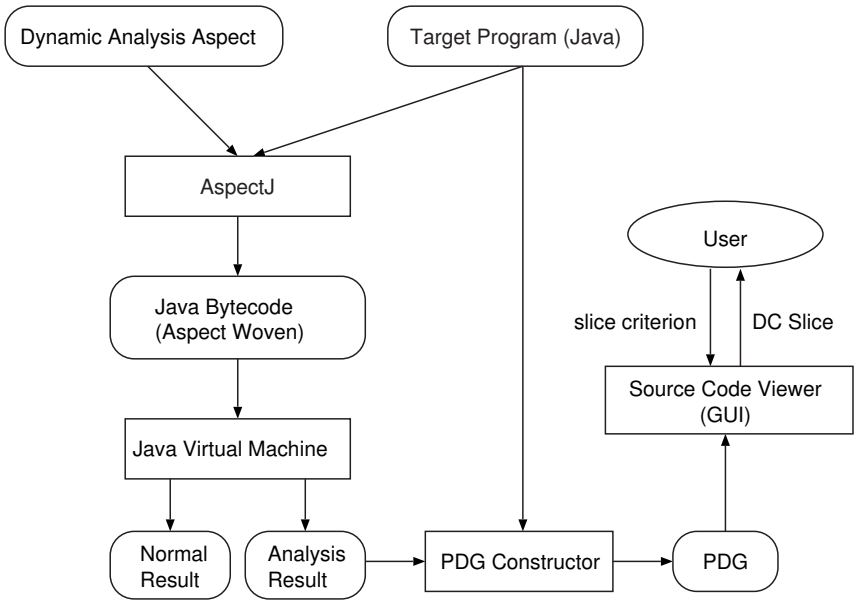

Figure 2.5: DC slicing system

2.4.2 Static Analysis Supplement

In AOP, an aspect may be limited by usable join points and by the applicable operation to the join points. The join points of AspectJ do not include local control structures (e.g. *if, while, for* statements), nor does AspectJ allow access to local variables. Such fine-grained join points are rarely required to improve separation of concerns.

Although the usual dynamic analysis requires the observation of the behavior of all variables and control structures, we cannot implement the proper dynamic analysis in AspectJ. Instead, we statically collect information about local variables and control structures for the compensation. This approach seems sufficient because the data dependence of local variables and the execution paths of local control structures are limited, and they are only affected slightly from dynamically determined elements in OOP. In section 2.5.2, we will discuss this issue based on the result of experimental evaluation.

2.4.3 Analysis of Libraries

Since AspectJ links the aspects to a target source code, AspectJ cannot link them into library classes. In this case, the library classes indicate reusable components which are not included as source codes.

In this research, libraries are excluded from analysis for the following reasons:

Library classes are reliable. Since library classes are repeatedly reused, it can be assumed that defects in the libraries are already removed. Therefore, we do not need to conduct a detailed analysis into the library classes.

Amount of code of library is numerous. The cost of the dynamic analysis of libraries is generally higher than for the main program.

When a program uses callback from the library, a hidden dependence via the library might be caused. This dependence can be extracted by the dependence analysis at bytecode level [38].

However, even if we use the bytecode analysis, a dependence analysis to important objects, such as file I/O and basic data structures, is unavailable because of the limitations in the Java language described in Section 2.4.4. Therefore, we cope with the problem by using static analysis.

When a program calls a method in a library, the aspect receives only information from the caller method since the aspect is not attached to the library. Then, the aspect extracts a virtual data dependence relation between a call statement and a return value. We assume that a return value of the called method is usually affected by the parameters of a call. Also, if another method is called back from a library, the aspect receives only information of the called method. Then the aspect extracts a virtual control dependence relation between the last call to a library and the called method.

2.4.4 Loop Caused by Aspect

Although AspectJ has an advantage that allows programmers to easily write aspects in Java, AspectJ causes dependence relations from the dynamic analysis aspect to classes used to record dynamic information. Therefore, if the aspect is built into such classes, the aspect and classes might cause a loop.

Figure 2.6 shows an example of such a loop. In Figure 2.6, the aspect operates by corresponding to a method call *Foo.getX*. The aspect calls *Foo.hashCode* to get the hash code of the object, and calling *Foo.getX* occurs in *Foo.hashCode*. Solving the problem that the aspect and classes cause a loop is not possible in Java

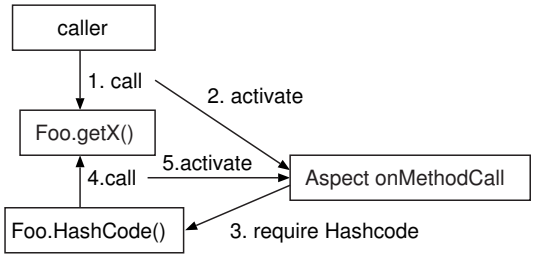

Figure 2.6: An example of a loop caused by an aspect

Table 2.3: Target programs

	Program	# of classes	Size (LOC)
P1	Simple database	4	262
P2	Sorting	5	228
P3	DC slice calculation	125	16207

language. Only the approach such as the customized JVM approach can solve this problem.

Since we have implemented the data analysis module using a Java standard library, a loop might be caused if the target program has the methods called from a standard library. Since we use only a hash table and an output stream in a standard library, two methods are called from the library. One method is *Object.toString*, which is a method that converts an object into a character string to make data readable. Another method is *Object.hashCode*, which is a method that calculates the hash code for fast access to data structures. Avoiding the loop is possible by not weaving the aspect into these methods. This implementation causes a decrease in the completeness of the information, but we consider that this incompleteness does not influence practical use because these methods are only used to store objects to a certain data structures, such as the hash table, and these methods are usually independent on the other part of the program.

Table 2.4: Slice size [LOC]

Slice criterion	Customized JVM	Aspect	Aspect/JVM
S1 (P1)	29	36	1.24
S2 (P2)	28	50	1.79
S3 (P3)	708	839	1.19

2.5 Experimental Evaluation

2.5.1 Overview

We have compared the DC slice system with the system developed using the customized JVM approach [38] in order to evaluate the proposed DC slice system. The evaluation stands on the viewpoint of implementation cost, modularity of the dynamic analysis and quality of resulted program slice since a customized JVM approach analyzes Java bytecode and extracts data dependencies even in the libraries.

In the evaluation, we have used the programs shown in Table 2.3 as the input of the systems. P1 is a simple database program which contains few elements of the object-oriented language. P2 is a program which uses polymorphism to switch sorting algorithms. P3 is the DC calculation system presented in this paper. The calculation system includes many features of Java, such as polymorphism, classes and package hierarchies, exception handling, and interactive user interfaces.

We have executed each program once with certain input data, and calculated the DC slice for arbitrary slice criterion.

In Section 2.5.2, we evaluate and discuss DC slice size. In Section 2.5.3 and 2.5.4 we also discuss time cost and module size necessary for DC slice calculation.

2.5.2 Slice Size

Here we compare the two slicing tools from the viewpoint of resulting slice size. Table 2.4 shows the size of DC slice for slice criterion S1 in P1, S2 in P2, and S3 in P3. Since each program outputs a set of data to file or GUI, the slice criterion is chosen from the variables referred at an output statement.

The DC slices calculated by both systems included the correct DC slice that is obtained manually, but they also included redundant statements. The difference of the slice size shows the difference of correctness.

In our approach, we have to statically analyze the target program to collect information about local variables and local control structures. Therefore, statements which are possibly dependent but are actually independent of the slice criterion

included in the slice result. For example, assume that there are some conditional clauses in the program and one of them is not executed because the corresponding conditional predicate is not satisfied. Then, the statements not executed in the conditional clause are included in the result of our approach, but excluded from the result of customized JVM approach.

On one hand, for the program P1 with a slicing criterion S1, the DC slice sizes of the customized JVM approach was 29 lines of code (LOC) and the size of our approach was 36 LOC, respectively. No substantial difference exists because the program size of P1 is small and does not include the characteristics of an object-oriented program.

On the other hand, for the program P2 with a slicing criterion S2, the size of our approach became about twice the size of the customized JVM approach. Program P2 is small but contains several methods which use many local variables and nested control structures.

The difference is not huge for a program P3 with a slicing criterion S3, although the size of P3 is much larger than the other programs, P1 and P2. Program P3 is skillfully decomposed into modules with proper sizes, and each method has a few local variables and simple control structures.

As we expected, the result shows that the size of the DC slice of our approach is larger than the slice of the customized JVM approach for programs that include many local variables and local control structures. However, for the size of the target program (especially P3), the difference of the resulting slice size between the two approach is insignificant.

This is because our approach uses dynamic information such as a method call and a field reference. Even if statements never executed are included in a slice using static information, a method called from such statements is excluded from the slice since no dynamic information for the statements is available.

Removing never executed statements from a slice using the information of a control flow and lack of dynamic information for the statements is a future work.

2.5.3 Analysis Cost

Here we evaluate the time necessary for calculating the DC slice. Table 2.5 shows the time needed to execute the Java program with a normal JVM, with a customized JVM, and the program to which dynamic analysis aspect has been attached with a normal JVM (our approach) for the same input. These values are measured in a JIT disabled environment. The execution time with enabled JIT is shown in Table 2.6.

In general, our approach shows better performance than the customized JVM approach. We believe that the cost of a dynamic analysis of the local variables is

Table 2.5: Execution time (JIT disabled) [sec.]

Target program	Normal	Customized JVM	Aspect
P1	0.18	1.8	0.26
P2	0.19	2.8	0.39
P3	1.2	81.0	10.3

Table 2.6: Execute time (JIT enabled) [sec.]

target program	Normal	Aspect
P1	0.24	0.34
P2	0.24	0.41
P3	1.1	9.9

very expensive, because of infrequent use of the library in P1 and P2. Moreover, in P3, analyzing internal processing in the library required further cost. As program size becomes larger, analysis cost must increase further because more libraries are used.

Our aspect approach has the advantage that we can use a JIT compiler to improve performance. In small programs such as P1 and P2, performance of the program without optimization by JIT compiler is better, because the optimization is not effective in this case. However, in a practically-large scale program like P3, the JIT compiler is very effective to improve performance. Although the effect of the JIT compiler varies with runtime environment, JIT makes a crucial difference on system performance [86]. Improving performance is paramount because a program is executed repeatedly in the debugging process.

2.5.4 Effort to Implement the Slicing Tool

In this section, we examine the effort of implementing the slicing tool. The size of the dynamic analysis module implemented as an aspect is about 400 lines of code (LOC). The total size of DC slice calculation tool reached about 16,000 LOC in Java.

In our approach, the aspect is described in the viewpoint of the join point model without meta-programming facilities. It results good readability compared with the pre-processor approach. Moreover, because the aspect is small and simple, the programmer (user) can easily modify the implementation to adapt each runtime

environment.

On the other hand, the customized JVM approach needed to add about 16,000 LOC to the JVM and Java compiler, whose total size is about 500,000 LOC [39]. The additional code consists of two parts, dynamic analysis and source code analysis. The dynamic analysis handles local variables that the aspect does not handle. The source code analysis extracts a map between source code and byte code. This map is required to map a node in a program dependence graph to a line in source code.

Furthermore, the overall program must be re-customized when the original JVM is updated. Therefore, keeping the customized JVM consistent with the original JVM is unrealistic. Our aspect approach, which uses the aspect written once, is applicable to any platform where the aspect weaver is available. Since AspectJ is written in Java, the aspects achieve good reusability, much cheaper to implement than the customized JVM approach.

2.6 Summary

In this chapter, we have examined modularization of dynamic analysis which collects dynamic information for program slicing. We have developed a DC slice calculation system including a dynamic analysis aspect and evaluated its usefulness.

Since we make pointcuts of the aspect in a generic form, the dynamic data dependence analysis aspect can be woven into various object-oriented programs without changes. We have improved maintainability and readability of the dynamic analysis since the aspect is simply defined without meta-programming facilities.

Although we have chosen AspectJ to implement the module, the design of the dynamic analysis is reusable for other aspect weaver supporting a join point model which is compatible with the AspectJ model. AspectJ join point model has a limitation that does not allow developers to analyze local variables and local control structures. If a developer needs fully implemented DC slicing system, the developer needs an appropriate aspect weaver which supports fine-grained join points.

Chapter 3

Modularization of Assertions Crosscutting Objects

3.1 Introduction

Design by Contract [50] provides behavioral specifications including preconditions, postconditions and invariants to improve robustness of software. Preconditions protect the called component from illegal calls, and postconditions protect the caller against erroneous implementations, respectively [64]. However, they are hard to handle properties held in interactions among objects because traditional assertions specify behaviors of one object used by arbitrary clients.

We show one variant of Observer pattern [16] for an example. In the Observer pattern, observers and subjects are modeled as many-to-many relationship, in other words, a number of observers may observe one subject and an observer may observe several subjects. After a developer has implemented the observer pattern, the developer may reuse the many-to-many relationship code for one subject-to-many observers relationship because many-to-many implementation covers one-to-many usage. Although the developer may add assertions to observers and subjects in order to prohibit attaching an observer to several subjects, the assertions in the observers and the subjects strongly depend on each other.

This kind of assertions crosscutting objects is caused when a developer assumes some interaction patterns among the objects. Assertion specifying interactions is a promising tool for software maintenance since the behavior out of the interaction patterns expected by the developer may indicate a defect [11]. To use assertions effectively, we need a method to write assertions in a well-modularized manner since assertions crosscutting objects are harmful to the maintainability of software.

We propose to modularize assertions crosscutting objects as an aspect using an

aspect-oriented language. In order to show the effect of modularization, we have defined a simple language whose pointcuts are a subset of AspectJ and compared two versions of the observer pattern with the one-to-many constraint in Java and in our language. As a result, modularized assertion simplifies objects and improves the maintainability of the objects.

This chapter consists of six sections. In the next section, we present the background of the research. Section 3.3 describes our proposal modularizing crosscutting assertions to aspects. In Section 3.4, we show the difference between our approach and traditional assertion. In Section 3.5, we discuss software quality affected by our approach, situations to which our approach is applicable and related work. We draw conclusions in Section 3.6.

3.2 Motivation

Design by Contract [50] improves robustness of software by specifying the behavior of a component based on preconditions and postconditions for each method of the component. Preconditions protect the called component from illegal calls, and postconditions protect the caller against erroneous implementations, respectively [64].

Practical programming languages such as Java and C++ have `assert` as a language construct, a function of the standard library, or a macro of a preprocessor. The behavior of `assert(expr)` statement is shown as follows.

assert(true)	$\rightarrow$	do nothing
assert(false)	$\rightarrow$	throw a runtime exception

Preconditions and postconditions of a method are regarded as `assert` statements inserted into the beginning of the method and the end of the method, respectively.

Assertion checking is powerful, practical, scalable and simple to use. Assertion is effective to detect software faults during debugging, testing and maintenance [62]. Assertion supports developers in understanding the software because it documents the behaviors of a component and effectively prevents developers from depending on implementation details of the component [47].

Several behavioral specification languages and tools including JML [87], jContractor [30], Larch [21] and Contract4J [82] are proposed to use assertion effectively. They provide several convenient functions and predicate to improve expressiveness of assertions. A developer describes properties for each method of a component using these languages. Gibbs et al. have proposed *Temporal Invariants*, or an extension of assertion for temporal properties held in a series of method calls

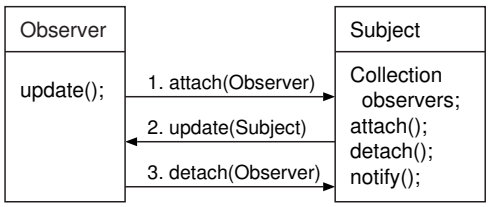

Figure 3.1: Observer Pattern

for one component [17]. On the other hand, Yamada et al. have proposed *Moxa*, or an aspect-oriented extension of JML [76]. Moxa provides language constructs to write common properties to several methods and classes. These approaches extend assertions to describe a property related to several methods. However, they are hard to handle properties held in interaction among objects because traditional assertions specify behaviors of only one object used by arbitrary callers.

We show a variant of Observer pattern [16] for an example. Observer pattern is an interaction pattern between Observers and Subjects. Figure 3.1 shows the structure of the pattern. `Subject` represents an object which has some data, and `Observer` represents an object watching subjects. An observer first attaches itself to a subject by calling the `attach` method. When the state of a subject is updated, the subject notifies the attached observers by calling `update` method of them. The notified observers call some methods of the subject to get updated information. An observer calls the `detach` method of a subject when the observer no longer need notification message from the subject. Observers and subjects are modeled as many-to-many relationship in the pattern. In other words, a number of observers may observe one subject and an observer may observe several subjects.

The assertion is hard to handle inter-object properties since the traditional assertion is described for each class. We discuss a variant of the observer pattern, a model of one subject-to-many observers relationship in order to show a limitation of the traditional assertion. After a developer has implemented an instance of the usual observer pattern, the developer can reuse the many-to-many relationship code for one-to-many relationship since many-to-many implementation covers one-to-many usage. The developer may want to add assertions in order to prevent an observer from being attached to several subjects. However, it is a hard task for the developer to describe the assertion for that purpose in a modularized manner since an observer has no variable which represents how many subjects the observer attaches to. So a developer need to add a field containing an attached subject to Observer and modify Subject to check and update the field when an observer is attached. The scattered code damages modularity and maintainability of the com-

ponents. We show detailed code in Section 3.4.

We propose to write such crosscutting assertions in an aspect. In Aspect-Oriented Programming [34], an aspect is a module unit for a crosscutting structure such as above example. The features of our approach are following:

- Our approach is based on aspect-oriented programming. It is important to separate crosscutting assertions from objects since assertion is often regarded as a part of the interface of an object [50]. The crosscutting assertions of objects affect modularity and maintainability of the objects.

- An aspect-oriented approach also enables developers to separate assertions into aspects for each purpose. Developers could not group assertions for each purpose in traditional approaches since traditional assertions are written for each method of a class. While Moxa also supports developers to group common properties for several methods, our approach allows developers to group several properties of classes for each purpose.

- Crosscutting assertions are caused in various situations. For example, developers write a method along with assumptions for the usage of the method. The developers usually write such assumptions in a comment such as "This method `foo` is to be called from the method `bar`". We should assert such assumptions since other developers may accidentally break assumptions when they reuse the method, and violated assumptions often cause a defect. We discuss the applicability of our approach in Section 3.5.4.

- Developers can deploy an aspect including application specific assertions to legacy components. Heineman pointed out that a service should be provided to enforce local properties specified by components as well as global properties specified by the application [25].

In the next section, we present the details of our approach.

3.3 Assertion as an Aspect

We propose to modularize crosscutting assertions in aspects. First we discuss language constructs which are useful to describe assertion. After the short discussion, we introduce a new simple aspect-oriented language whose pointcut designators are a small subset of AspectJ [80] to show basic language constructs to write assertions.

Pre-/post-conditions of a method are checked before/after the method is called respectively. Therefore, we regard them as `before`/`after` advices with a `call`

pointcut in AspectJ. When assertions are separated from the objects, the separated assertions need a way to access context information including a method caller object, a callee object, their own fields, method parameters and a return value through context exposure provided by AspectJ.

Comparing with an aspect implementing some functionality (or non-functional requirements), an aspect for assertion has following features.

- An aspect observes method call events and often accesses contextual information, a caller object and a callee object. So a developer often uses pointcut designators including `call`, `this`, `target` and `args`.
- An aspect has utility methods and fields to collect information through several method calls.
- An aspect sometimes needs to access private members of the object.

We have developed a simple aspect-oriented language specialized to write an aspect for assertion based on the above features. Our language provides several pointcut designators to easily access context information.

We allow a module of our language to include a block written in AspectJ to declare methods for utility functions and advices handling context information. We have defined our language to be converted to AspectJ since our purpose is not to develop a new practical language but to use a simple subset of AspectJ to write assertions.

3.3.1 Assertion Module

In this language, a developer declares assertion aspect using the keyword `assertion`. The declaration of a module consists of the name of the module, a module level pointcut and a set of advices.

```
assertion name ( params ) : pointcut
  << advice definition >>
end
```

The name of a module is provided just for management, it has no meaning in programming semantics. A module may have a module level pointcut describing a common pointcut among advices included in the module. The pointcut is optional, a developer may omit ": pointcut" fragment. Our language provides following primitive pointcuts to specify context in simple expression:

- `p calls q` represents a method call from `p` to `q`. `p` and `q` usually specify objects declared as parameters of the module. `p` and `q` may be type name or a wild card "`*`" when the developers have no interest in caller/callee objects, respectively. This statement is translated into following pointcut designators: `call(* *.*(..)) && this(p) && target(q)`.

- `p calls q.method(params)`, which is another form of the pointcut, is also allowed to specify the signature of the methods. This form is translated into the following pointcut designators: `call(* *.method(..)) && args(params) && this(p) && target(q)`.

- `if(expr)` represents a condition of the context. The assertions in the context module are enabled when the `expr` is true. This pointcut is exactly same as `if` pointcut of AspectJ.

- `method(signature)` represents a method signature constraint. This pointcut is simply converted to `call(signature)` pointcut. A developer uses this pointcut to specify methods when the developer is not interested in objects.

Although above pointcuts are sufficient to write usual assertions, other pointcut designators are also useful to write assertions. For example, `cflowbelow` can specify pre/post-conditions for recursive method calls. Examining how powerful pointcuts such as `cflow` and `dflow` [45] affect the expressiveness of assertions is future work.

Parameters in the module declaration are module level variables, so all advices in the module can access the parameters. It allows developers to declare common parameters for each assertion.

3.3.2 Assertion Advice

An assertion module includes a number of advices. An advice is defined in the following format:

```
def name ( params ) : pointcut
  pre
    <<expression or code block>>
  post
    <<expression or code block>>
end
```

The name, the parameters and the pointcut of an advice are same as the module level declaration. `pre` and `post` specify preconditions and postconditions,

respectively. A developer writes a list of Boolean expressions and code blocks in AspectJ. A list of expressions separated by "`;`" specifies conditions that must be satisfied. A code block is a procedure executed at the beginning or the end of the method. We assume that the code block updates variables for assertion checking. Here we show an example code as follows.

```
assertion OneSubjectManyObserver
 def attach(Observer o, Subject s):
  * calls s.attach(o)
   pre  o.subject == null;
   post o.subject == s;
        s.getObservers().contains(o);
        {
         Logger.log(o, "connects", s);
        }
 end
end
```

The above code is same as following AspectJ code:

```
aspect OneSubjectManyObserver {
 // condition attach
 before(Observer o, Subject s):
  call(* Subject.attach(Observer)) &&
  target(s) && args(o) {
    assert o.subject == null;
 }
 after(Observer o, Subject s):
  call(* Subject.attach(Observer)) &&
  target(s) && args(o) {
   assert o.subject == s;
   assert s.getObservers().contains(o);
   Logger.log(o, "connects", s);
 }
}
```

And an assertion module may include utility methods, member variables (fields), inter-type declarations, internal classes and arbitrary advices in AspectJ. The format is simple block form as follows.

```
{
  <<AspectJ Code Block>>
}
```

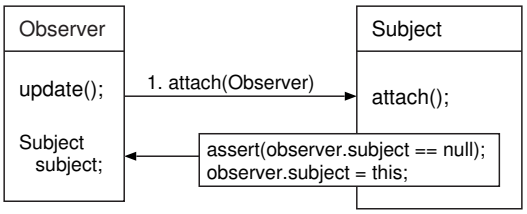

Figure 3.2: Crosscutting assertion in Java

3.3.3 Implementation

We have implemented a translator from our language into AspectJ using `Racc`, or a parser generator for Ruby [90]. Our translator converts pointcut declarations to AspectJ style, and just copies code blocks to the output. Therefore, developers can use several pointcut designators which are not directly supported by our translator, e.g. `cflow`. Since our language is translated into AspectJ, the code optimization of AspectJ provides executable code with less-overhead [26].

3.4 Case Study

We have implemented an Observer pattern with the one subject-to-many observers relationship constraint as a case study. One-to-many relationship means that an observer can register to only one subject, and one subject can be observed by multiple observers. Such relationship is found when a subject represents a data model and several views for the model are provided as observers. Developers can simply reuse normal Observer pattern to achieve the purpose, but some constraints are needed to prevent an observer to watch several subjects. Here, we are comparing two implementations, in Java and our language, in the view point of the modularity.

Figure 3.2 shows the overview of the implementation in Java, and Figure 3.3 shows code fragments added to the usual Observer pattern implementation. This implementation is problematic because it damages the encapsulation of Observer. An observer has a reference to a subject and the method `Subject.attach` checks and updates the field. When an observer calls `attach`, the subject checks that the observer is not connected to any subjects using the `subject` field of the observer. After the subject accepts the observer, the subject updates the observer's field. The observer must not modify the field by itself nevertheless it is the field of the observer. It is a bad manner to prevent a component to modify its field and to allow another component to modify the field.

```
// Extended interface for Observer
interface Observer2 extends Observer {
  public Subject getSubject();
  public void setSubject(Subject subject);
}

// Extend an Observer
public class NewObserver extends AnObserver
  implements Observer2 {
  Subject subject;

  public void setSubject(Subject subject) {
    this.subject = subject;
  }

  public Subject getSubject() {
    return subject;
  }
}

// New subject
public class ASubject implements Subject {
      :
      :
  public void attach(Observer o) {
    assert (o instanceof Observer2) &&
     (((Observer2)o).getSubject() == null);
    assert !observers.contains(o);
    this.observers.add(o);
    assert observers.contains(o);
    ((Observer2)o).setSubject(this);
  }

  public void detach(Observer o) {
    assert (o instanceof Observer2) &&
     (((Observer2)o).getSubject() == this);
    assert observers.contains(o);
    this.observers.remove(o);
    assert !observers.contains(o);
    ((Observer2)o).setSubject(null);
  }
}
```

Figure 3.3: One-to-many relationship in Java

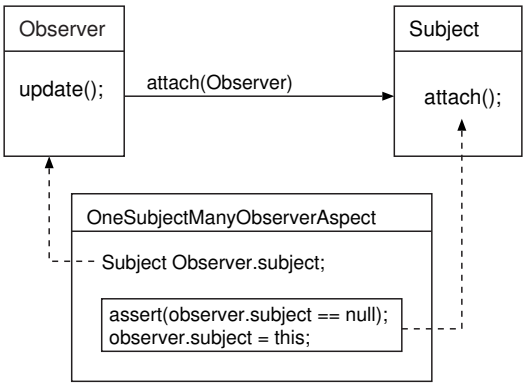

Figure 3.4: Assertion modularized in an aspect

The broken encapsulation also affects maintainability of the code. Developers need to maintain two versions of the subject and the observer for one-to-many and many-to-many relationships because the code fragments included in the observer and the subject depend on each other to implement the one-to-many constraint. Developers cannot mix one-to-many observers and many-to-many observers for one subject.

Figure 3.4 shows the overview of a solution in our approach. The aspect has separated crosscutting assertions from subjects and observers. The source code is shown in Figure 3.5. The field of `Observer.subject` is also moved to the aspect. The aspect introduces the field `subject` into Observer. Its value must be null before the method `attach`, and a reference to subject is set to the field after `attach` is called. The value is cleared after the method `detach` is called.

The aspect modularizes all related assertions, fields and methods. The modularization prevents a developer from accidentally mixing these assertions with assertions for other purposes and from misusing methods and fields defined only for the assertions.

An advantage of our approach is that developers can deploy one-to-many relationship for generic observers and subjects. Another advantage is that developers can mix one-to-many observers and many-to-many observers for one subject since the aspect affects only a pair of `AnObserver` and `ASubject`.

Our language enables a developer to easily write assertions. When a developer uses AspectJ to implement one-to-many observer pattern aspect, the implementation is similarly modularized as our language. The difference from our language is that a developer writes a pair of before and after advices for a method in AspectJ instead of a set of preconditions and postconditions in one advice.

```
assertion OneSubjectManyObserver

 { // AspectJ inter-type declaration
   public Subject AnObserver.subject = null;
 }

 def attach(ASubject s, AnObserver o):
   * calls s.attach(o)
 pre   !s.getObservers().contains(o);
       o.subject == null;
 post  s.getObservers().contains(o);
       { // code executed after s.attach
         o.subject = s;
       }
 end

 def detach(ASubject s, AnObserver o):
   * calls s.detach(o)
 pre   s.getObservers().contains(o);
       o.subject == s;
 post  !s.getObservers().contains(o);
       {
         o.subject = null;
       }
 end
end
```

Figure 3.5: One-to-many relationship aspect

3.5 Discussion

In this section, we discuss about behavioral subtyping, modular reasoning, situations to which our approach is applicable and related work.

3.5.1 Behavioral Subtyping

Behavioral subtyping guarantees that all objects of a subtype preserve all of the original type's invariants [15]. Our approach enables developers to add assertions to a component using an aspect, or an external module. When a developer creates a new subclass of a component, assertion should automatically affect the new subclass for the consistency of the assertion. For example, when an aspect adds an assertion to a class P, a class Q which is a subclass of P is also affected by the assertion.

Our approach allows developers to add preconditions to a component. This feature may break behavioral subtyping since a subtype may have weak preconditions and strong postconditions of the supertype, but cannot have strong preconditions and weak postconditions [44]. However, we decided to allow strong preconditions since a developer sometimes can assume the stronger preconditions based on application constraints [25].

On the other hand, we prohibit removing assertions from objects for safety. Although the behavioral subtyping allows weak preconditions, we prevent a developer from being confused by mixing a code fragment explicitly violating a precondition with an aspect removing the precondition.

3.5.2 Modular Reasoning

Assertion documents the abstract behavior of a method. Well-modularized assertions provide useful information for a developer to understand interactions among objects.

Assertions separated from a component might reduce the comprehensibility of the component since assertions for a component crosscut a class and several assertion modules. Therefore, Aspect Visualizer [9] and tools finding aspects are important for managing aspects. If a developer finds aspects using Aspect Visualizer, the developer can inspect all related assertions. This is easier than finding all crosscutting assertions in other classes with `grep` or other tools [35]. A program just crashes even if a developer who does not know assertion added by an aspect writes a code fragment which violates the assertion. A developer can get assertion information from the stack trace of a crashed program, so it is not too serious problem.

3.5.3 Avoiding Side Effects

Assertion aspect should have no side effects for the state of objects. If the side-effect free methods are available, we should enforce aspects to call only such methods. Java with Access Control proposed by Kniesel et al. [37] is promising approach for this purpose since it provides `readonly` context prohibiting side effects and a method statically checking the context.

We allow an aspect to have its own state in order to calculate some statistical value, record a log or other similar purposes. Assertion aspect does not reduce maintainability since a developer does not need impact analysis for an assertion aspect when the aspect has no side effect on other objects.

3.5.4 Applicability

We have shown the example of the Observer pattern. Here we discuss the applicability of our approach. Our approach is also applicable to following situations:

- *Client specific assertions for a reusable component.* A simple example is a list (e.g. `java.util.List`) containing strings which match a certain regular expression. If several developers want to share such a list in their application, they may develop a new list component. However, many lists are hard to maintain when each component needs a customized list respectively. This problem is well-known since a developer often needs to handle a set of data with containers, and C++ template mechanism and Java Generics support containers with type constraints. Our assertion approach supports other constraints that cannot be handled in type checking mechanisms.

- *Assertions for experimental/untrustworthy code.* A developer can add strict assertions to only new code but not to other well-tested code when the developer creates a new client accessing a long-lived component. After the new client is tested, the developer may remove assertions easily by just removing the assertion aspect. A developer can also write assertions for untrustworthy input from an external component added by a user after the system is released, e.g. a plug-in extending the software.

- *Assertions describing developers' expectation.* Developers sometimes have implicit assumptions such as "When this method `foo` calls the method `bar`, the object holds a particular condition." Although some developers declare this kind of expectation as a comment, other developers may accidentally break such assumptions when they introduce subclasses or aspects [48]. Our assertion may protect the method from the illegal usage by specifying the

caller's state. Specifying strict assertion for the usage of a component, developers might assure the behavior and the quality of the component.

- *Assertions for component behavior affecting other components.* A component usually accesses other components to achieve its task. Our assertion can specify a condition about method calls to other components. Since a wrong series of method calls from a component indicates a defect of the component [11], an assertion checking the behavior of an object is useful for developers. Developers can write assertions and utility advices checking the behavior of the object independent from the object code. Temporal invariants are also useful for this purpose [17].

- *Collaborating with unit testing.* JUnit [89] is a well-known unit testing tool for Java. JUnit provides class libraries which support to write assertions and a tool which executes a test suite and collects the result. Our approach supports to test interactions among objects in addition to unit testing since assertion methods of JUnit can specify only the state of an object but cannot specify the expected behavior of the object, e.g. methods should be called in a test case.

3.5.5 Related Work

Zhao et al. proposed Pipa, or an extended language of JML which enables programmers to write assertions for advices [78]. Hanneman et al. have shown the usefulness of aspects implementing interactions among objects [23], and published their code [24]. If a developer introduces Pipa into an Observer pattern implemented in AspectJ, the result may be similar to our approach because Pipa code fragments are assertions for each advice and our code fragments are translated into assertion statements in advices. However, the purpose of Pipa is different from ours. Pipa aims to introduce assertion checking for advices; our approach aims to modularize crosscutting assertions. The concept of crosscutting assertion in our approach includes assertion for a set of the components which may be coded only in classes as main functionality (base code).

Yamada et al. proposed Moxa, another extension of JML [76]. Moxa provides an assertion module which enables programmers to write assertions shared by several methods and classes. Our approach focuses on modularizing crosscutting assertions into an aspect, their approach focuses on modularizing common properties into an aspect. The approaches may collaborate with each other.

Gibbs et al. have proposed Temporal Invariants, or an extension of assertion for temporal properties for one component [17]. Temporal invariants can describe

temporal properties held in a series of method calls for one component. This approach can replace a process collecting context information (some flags checking control flow) with an expression of the temporal logic. When a developer can write temporal invariants for several components, it would be more useful tool.

Our current implementation statically weaves assertions into other modules. Dynamic deployment of aspect [51] enhances flexibility and usability of assertions since it allows developers to temporarily deploy assertions for a certain context to components.

3.6 Summary

Assertion documents the behavior of a component. Assertion checking is a powerful tool to detect software faults during debugging, testing and maintenance. Since traditional assertions are described for each method, the assertions crosscut several modules in order to specify inter-object properties. Crosscutting assertions are harmful to the modularity and maintainability of the components. Therefore we have proposed to modularize such assertion as an aspect using an aspect-oriented language. We have introduced a simple aspect-oriented language to show basic language constructs to write assertions, and developed a translator for the language into AspectJ. We have implemented two version of the Observer pattern in java and our language and have shown that crosscutting assertions in Java are modularized in our language. Aspect-oriented assertion is promising to improve software maintainability, and is applicable to various situations. In the future work, we are planning to research design by contract for inter-aspect properties and examine how powerful pointcuts such as `cflow` and `dflow` affect the expressiveness of assertions.

Acknowledgement

This work was supported by MEXT.Grant-in-Aid for JSPS Fellows (No.17-9539).

Chapter 4

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

4.1 Introduction

Aspect-Oriented Programming (AOP) proposes a new module unit, or *aspect*, for encapsulating crosscutting concerns such as logging and synchronization [34]. 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 developer can separate crosscutting concerns from objects. Various applications of AOP have been reported [23, 65].

Although AOP is useful, it introduces a new complexity into a program. Since an aspect modifies the behavior of objects, a developer must inspect objects and related aspects to understand system behavior; otherwise, the developer 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 [55]. Early detection of inter-aspect problems is crucial to support the debugging tasks.

In this chapter, 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 [74]. 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* [53], which is a program-slicing method based on static and dynamic dependence relations in a program.

We implement a call graph construction and program slice calculation tool as an Eclipse plug-in [83]. When a developer 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 chapter is as follows: in Section 4.2, we present a brief overview of Aspect-Oriented Programming. In Section 4.3, we describe infinite loop detection using a call graph. In Section 4.4, we present an extension of program slicing for an aspect-oriented program. In Section 4.5, we evaluate the proposed method and discuss experimental results. In Section 4.6, we conclude our discussion with remarks regarding plans for future work.

4.2 Aspect-Oriented Programming

4.2.1 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, developers must modify all related objects.
- Developers cannot reuse an object independently of other objects since objects are connected to each other with a crosscutting concern.
- Developers cannot reuse implementation of a concern independently of objects. If another set of objects interacts in the same way, developers must re-implement the concern.

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 developer 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.

```
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 4.1: Aspect examples: Logging and parameter checking

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 4.1. *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, developers 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, developers 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 [16]. Since an interaction of objects is a kind of crosscutting concern, developers can write a pattern as an aspect. Aspects implementing design patterns are reusable components [23]. On the other hand, it is also useful for applications to support debugging and to write crosscutting concerns in a distributed software environment [65].

4.2.2 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 se-

quence of advices may affect the result of calculation. An example is the program in Figure 4.1. 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) In the software evolution process, a pointcut definition may become obsolete according to the changes of objects.

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

Problems (a) and (b) are a part of inter-aspect problem [55], a research regarding how to solve such issues exists [13]. Problem (c) is known as fragile base-code problem. Aspect-aware refactoring is proposed for the problem since aspects may conflict with refactoring techniques in OOP [22]. Problem (d) debugging pointcut definition is partially supported by the Integrated Development Environment in AspectJ [79].

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 [28]. We propose a debugging support based on a call graph and program slicing. Debugging support is effective for problem (a), (b) and (d). Although program slicing is also applicable to debug a defect of problem (c) caused by a pointcut definition which becomes obsolete, program slicing cannot prevent a pointcut from being obsolete.

4.3 Loop Detection using Call Graph

Carelessly defined, incorrect pointcuts cause accidental advice executions. Accidental advice executions are hard to detect since a developer 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 [66]. A typical result of an incorrect pointcut is an infinite loop [81]. Infinite loops should be statically detected in a compilation process instead of runtime.

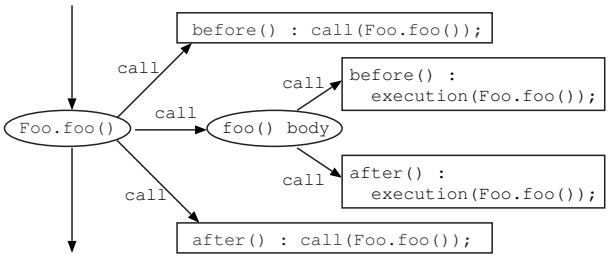

Figure 4.2: Before advice call and after advice call handled as method calls

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 [19]. 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 [26]. 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 [66].

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 pro-

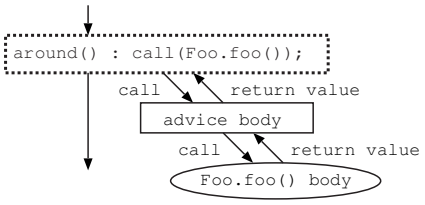

Figure 4.3: Around advice call handled as a method call

gram execution. We regard *join point shadows* [26] which may trigger an advice execution as an advice call. `before` and `after` advices are simply replaced to method calls as shown in Figure 4.2. An `around` advice is handled in another way since an `around` advice replaces join points. Figure 4.3 shows that an `around` advice replaces 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 [13] and a framework to allow developers to manually control advice executions exist [55]. 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 [69]. Figure 4.4 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 4.4, a cyclic 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.

Developers can confirm that inter-aspect dependence relations are their intentional result. Developers 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

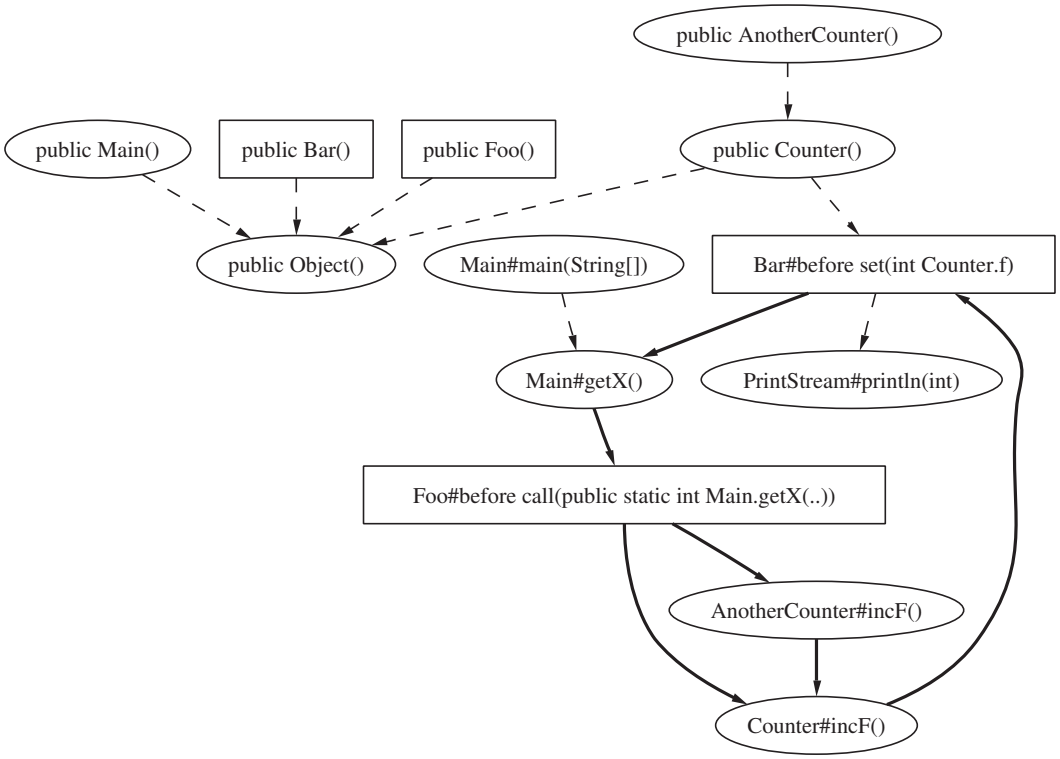

Figure 4.4: A call graph

extracting cycles and dependence relations between aspects is important. The implementation details are described in Section 4.5.1

4.4 Program Slicing

Program slicing is a promising approach for program debugging, testing, and understanding [74]. 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.

4.4.1 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 re-defining 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 [74] 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 [53]. Therefore, our approach is based on DC slicing.

An example of a DC slice for Java is shown in Figure 4.5. 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 4.5. When a program results in an invalid output, developers choose the variable which contains the output as a slicing criterion, and calculate a slice to localize a fault.

4.4.2 Extension 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 developer 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 developer first runs a test case to collect dynamic information. Next, a developer activates a tool to construct a program dependence graph. Finally, a program slice is calculated by slice criteria specified by the developer. A developer can localize a fault using the program slice.

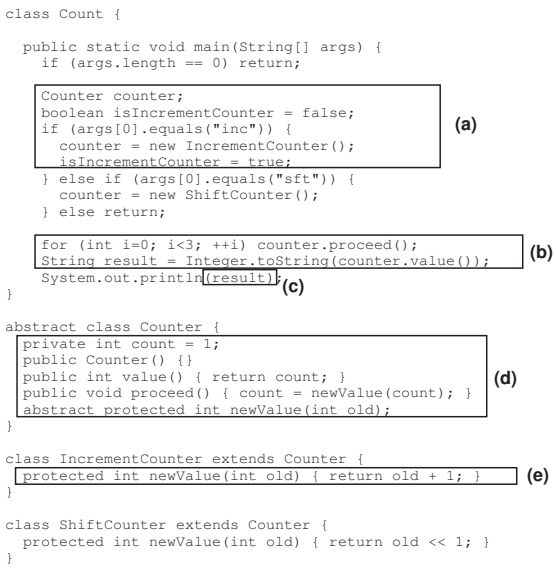

Figure 4.5: DC slice example

Program slicing for aspect-oriented languages has already been proposed, but has not been implemented and evaluated yet [77]. 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 [63]; 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.

4.4.3 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, which has been developed in the case study described in Chapter 2. Developers 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 data dependence relations need to be extracted dynamically too. Though dynamic information is effective to distinguish objects and to extract inter-method dependence relations, dynamic intra-method dependence relations are less effective. 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.

4.5 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 [24], 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 4.5.1, we describe the implementation overview of our tool. We present the former experiment in Section 4.5.2 and the latter experiment in Section 4.5.3.

4.5.1 Implementation Overview

Developers 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 developers 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 [83] for the platform of our tool. Eclipse is an open source IDE, and developers can write a plug-in in Java to add new functionalities

Table 4.1: Target codes

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

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 developers 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 [79]. On the other hand, we can apply a DC slicing tool for Java byte-code [72] since the current version of AspectJ compiler generates Java byte-code [80]. 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.

4.5.2 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 [24]. We have used five patterns in Table 4.1 since Developers can effectively implement these patterns using AspectJ [23]. We describe the result of the slicing in Section 4.5.2, and discuss analysis costs in Section 4.5.2.

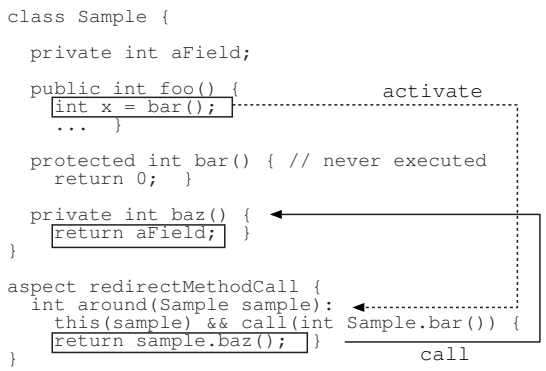

Figure 4.6: A slice including an aspect replacing a method call

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 execute 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, developers must track definitions of methods and advices manually, and this task requires much time since developers 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. Developers 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

Table 4.2: 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

relations modified by aspects. Figure 4.6 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 [79] provides an extended editor, which shows the locations where the advice is executed. However, since AJDT cannot visualize statements replaced by advices, developers must carefully consider *around* advices replacing original join points.

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 4.2. 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.

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 [80]. The size of the compiler is about 60,000

Table 4.3: 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 (*))

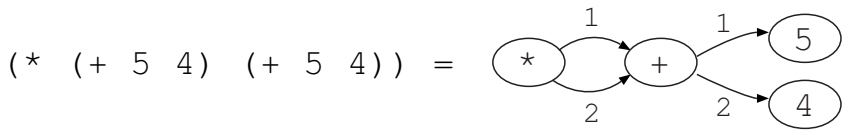

Figure 4.7: A graph representing an expression

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 developer 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 [14].

4.5.3 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 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.

Table 4.4: 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

An example of a graph representing $(* \ (+ \ 5 \ 4) \ (+ \ 5 \ 4))$ is shown in Figure 4.7. 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 4.3. 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 4.4 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.

4.6 Summary

In this chapter, 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. Developers 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 developers change code without knowledge about classes and related aspects, developers 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 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 are planning to extend our research on debugging support using impact analysis. We will also apply a reflection analysis based on dynamic analysis [20].

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.

Chapter 5

Conclusions

5.1 Summary of Major Results

In this paper, we have proposed to modularize two development aspects using aspect-oriented programming and implemented a development support based on program slicing technique.

First, we have proposed and implemented dynamic analysis for program slicing as an aspectual module. This approach improves modularity and maintainability of the dynamic analysis since the behavior of aspect is written in base code, not a meta-programming language used in traditional approaches.

Second, we have proposed to write assertions crosscutting objects as an aspect. Assertion is a part of the interface of an object, crosscutting assertion damages maintainability of the modules. Aspectual assertion is promising approach to reduce inter-dependence between module interfaces since it is applicable to various situations.

Using these aspects more effectively, we have extended program slicing technique for aspect-oriented programs and developed a program slicing tool for AspectJ. An experimental evaluation shows that the program slicing technique supports debugging a program including multiple aspects which causes inter-aspect problem.

5.2 Directions of Future Research

Although aspect-oriented programming is effective to improve modularity followed by better maintenability and reusability, managing concerns in a large scale system is difficult for developers. A program slice for a entire system is too large for developers to see [41]. In order to support to manage concerns, visualizing crosscutting

structure is important. Existing research for visualization of aspects focuses on visualizing relationship among an aspect and classes affected by the aspect [10]. Visualizing inter-aspect relationship extracted by program slicing technique is a research direction which supports aspect-oriented software development.

Another direction is modular analysis for dependence relationship among program entities. Since many class libraries and frameworks are used for software development in recent years, program analysis tools can access only a part of the source code in the system. In such a situation, a method effectively combining modular analysis for each component and available inter-component relationship information is important. Although this is not a special problem for aspect-oriented programming, the problem becomes more important since AOP environments are often implemented as a framework or a middleware.

Bibliography

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

[2] Ali, N. M. and Rashid, A.: A State-based Join Point Model for AOP. Proceedings of the Workshop on Views, Aspects and Roles 2005 (VAR 2005), http://swt.cs.tu-berlin.de/~stephan/VAR05, Glasgow, UK, July 2005.

[3] Arnold, R. S. and Bohner S. A. : Impact Analysis - Towards a Framework for Comparison Proceedings of the 19th International Conference on Software Maintenance (ICSM 1993), pp.292-301, Amsterdam, The Netherlands, September 1993.

[4] Apiwattanapong, T., Orso, A. and Harrold, M. J.: Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences. Proceedings of the 27th International Conference of Software Engineering (ICSE 2005), pp.432-441, St. Louis, Missouri, USA, May 2005.

[5] Ashida, Y., Ohata, F. and Inoue, K.: Slicing Methods Using Static and Dynamic Information. Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC 1999), pp.344-350, Takamatsu, Japan, December 1999.

[6] Booch, G.: Object-Oriented Design with Application. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, USA, 1991.

[7] Boström, G.: A Case Study on Estimating the Software Engineering Properties of Implementing Database Encryption as an Aspect. Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT 2004), http://www.daimi.au.dk/~eernst/splat04/, Lancaster, UK, March 2004.

[8] Chidamber, S. R. and Kemerer, C. F.: A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, Vol.20, No.6, pp.476-493, June 1994.

[9] Clement, A., Colyer, A. and Kersten, M.: Aspect-Oriented Programming with AJDT. Proceedings of Workshop on Analysis of Aspect-Oriented Software 2003 (AAOS 2003), Darmstadt, Germany, July 2003.

[10] Coelho, W. and Murphy, G. C.: ActiveAspect: Presenting Crosscutting Structure. Proceedings of the 1st International Workshop on the Modeling and Analysis of Concerns in Software. St. Louis, USA. May 2005.

[11] Dallmeier, V., Lindig, C. and Zeller, A.: Lightweight Defect Localization for Java. Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005), Glasgow, UK, July 2005.

[12] Demeyer, S., Ducasse, S. and Lanza, M.: A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. Proceedings of the 6th Working Conference on Reverse Engineering (WCRE 1999), Atlanta, Georgia, USA, October 1999.

[13] Douence, R., Motelet, O. and Südholt, M.: A Framework for the Detection and Resolution of Aspect Interactions. Proceedings of the 1st Conference on Generative Programming and Component Engineering (GPCE2002), pp.173-188, Pittsburgh, Pennsylvania, USA, October 2002.

[14] Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code, IEEE Transactions on Software Engineering, Vol.29, No.3, pp.210-224, March 2003.

[15] Findler, R. B., Latendresse, M. and Felleisen, M.: Behavioral Contracts and Behavioral Subtyping. Proceedings of the 9th Symposium on Foundations of Software Engineering (FSE 2001), pp.229-236, Vienna, Austria, September 2001.

[16] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Pub Co., Boston, Massachusetts, USA, 1995.

[17] Gibbs, T. H. and Malloy, B. A.: Weaving Aspects into C++ Applications for Validation of Temporal Invariants. Proceedings of Conference on Software Maintenance and Reverse Engineering (CSMR 2003), pp.249-258, Benevento, Italy, March 2003.

[18] Gosling, J., Joy, B. and Steele, G.: The Java ™ Language Specification. Addison-Weseley Pub Co., Boston, Massachusetts, USA, 1996.

[19] Groove, D., Furrow, G., Dean, J. and Chambers, C.: Call Graph Construction in Object-Oriented Languages. Proceedings of the 12th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1997), pp.108-124, Atlanta, Georgia, October 1997.

[20] Gschwind, T., Oberleitner, J. and Pinzger M.: Using Run-Time Data for Program Comprehension. Proceedings of the 11th International Workshop for Program Comprehension (IWPC 2003), pp.245-250, Portland, Oregon, USA, May 2003.

[21] Guttag, J. V., Horning, J. J. and Wing, J. M.: The Larch Family of Specification Languages. IEEE Software, Vol.2, No.5, pp.24-36, September 1985.

[22] Hanenberg, S., Oberschulte, C. and Unland, R.: Refactoring of Aspect-Oriented Software. Proceedings of Net.Object Days 2003 (NODe 2003), pp.19-35, Erfurt, Germany, September 2003.

[23] Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ. Proceedings of the 17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), pp.161-173, Seattle, Washington, USA, November 2002.

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

[25] Heineman, G. T.: Integrating Interface Assertion Checkers into Component Models. Proceedings of the 6th International Workshop on Component-Based Software Engineering (CBSE 2003), pp.37-42, Portland, Oregon, USA, May 2003.

[26] Hilsdale, E. and Hugunin, J.: Advice Weaving in AspectJ. Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), pp.26-35, Lancaster, UK, March 2004.

[27] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Languages and Systems, Vol.12, No.1, pp.26-60, January 1990.

[28] Ichisugi, Y., Tanaka, A. and Watanabe, T.: Extension Rules: Description Rules for Safely Composable Aspects. Technical Report of the National In-

stitute of Advanced Industrial Science and Technology (AIST), AIST01-J00002-4, February 2003.

[29] Jackson, D. and Rinard, M.: Software Analysis: A Roadmap. The Future of Software Engineering, pp.135-145, ACM Publishing, 2000.

[30] Karaorman, M., Holze, U. and Bruno, J.: jContractor: A Reflective Java Library to Support Design By Contract. Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection 1999), pp.175-196, Saint-Malo, France, July 1999.

[31] Katayama, T.: A Theoretical Framework of Software Evolution. Proceedings of the 1st International Workshop on Principle of Software Evolution (IWPSE 1998), pp. 1-5, Kyoto, Japan, April 1998.

[32] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold W. G.: An Overview of AspectJ. Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), pp.327-353, Budapest, Hugary, June 2001.

[33] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: Getting started with AspectJ. Communications of the ACM, Vol.44, No.10, pp.59-65, October 2001.

[34] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming. Proceedings of the 11th European Conference for Object-Oriented Programming (ECOOP 1997), Vol.1241 of LNCS, pp.220-242, June 1997.

[35] Kiczales, G., Mezini, M.: Aspect-Oriented Programming and Modular Reasoning. Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp.49-58, St. Louis, Missouri, USA, May 2005.

[36] Kiselev, I.: Aspect-Oriented Programming with AspectJ. Sams Publishing, Indiana, USA, 2002.

[37] Kniesel, G. and Theisen, D.: JAC - Access Right Based Encapsulation for Java. Software Practice and Experience, Vol.31, No.6, pp.551-576, May 2001.

[38] Konda, K., Ohata, F. and Inoue, K.: Extraction Method for Dynamic Dependence Relations between Bytecodes Using Java Virtual Machine. JSSST Computer Software, Vol.18, No.3, pp.40-44, in Japanese, March 2001.

[39] Konda, K.: An Extraction Method for Dynamic Dependence Relations between Bytecodes Using Java Virtual Machine. Master's Thesis, Osaka University, in Japanese, 2002.

[40] Koppen, C. and Stoerzer, M.: PCDiff: Attacking the Fragile Pointcut Problem. Proceedings of European Interactive Workshop on Aspects in Software 2004 (EIWAS 2004), http://www.topprax.de/EIWAS04/, Berlin, Germany, September 2004.

[41] Krinke, J.: Visualization of Program Dependence and Slices. Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), pp.168-177, Chicago, Illinois, USA, September 2004.

[42] Kusumoto, S., Imagawa, M., Inoue. K., Morimoto, S., Matsusita, K. and Tsuda, M.: Function Point Measurement From Java Programs. Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp.576-582, Orland, Florida, USA, May 2002.

[43] Law, J. and Rothermel, G.: Whole Program Path-Based Dynamic Impact Analysis. Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pp.308-318, Portland, Oregon, USA, May 2003.

[44] Liskov, B. H. and Wing, J. M.: A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, Vol.16, No.6, pp.1811-1841, November 1994.

[45] Masuhara, H. and Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming. Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS 2003), pp.105-121, Beijing, China, November 2003.

[46] McCabe, T. J. and Watson, A. H.: Software Complexity. Crosstalk, Journal of Defense Software Engineering, Vol.7, No.12, pp.5-9, December 1994.

[47] McCamant, S. and Ernst, M. D.: Predicting Problems Caused by Component Upgrades, Proceedings of the 11th Symposium on Foundations of Software Engineering (FSE 2003), pp.287-296, Helsinki, Finland, September 2003.

[48] McEachen, N. and Alexander, R. T.: Distributing Classes with Woven Concerns - An Exploration of Potential Fault Scenarios. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), pp.192-200, Chicago, Illinois, USA, March 2005.

[49] Merlo, E., Antoniol, G., Penta, M. D. and Rollo, V. F.: Linear Complexity Object-Oriented Similarity for Clone Detection and Software Evolution Analyses. Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), pp.412-416, Chicago, Illinois, USA, Septembr 2004.

[50] Meyer, B.: Object Oriented Software Construction. Prentice Hall, New York, USA, 1988.

[51] Mezini, M. and Ostermann, K.: Conquering Aspects with Caesar. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp.90-100, Boston, Massachusetts, USA, March 2003.

[52] Mikhajlov, L. and Sekerinski, E.: A Study of The Fragile Base Class Problems. Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP 1998), Vol.1445 of LNCS, pp.355-382, Brussels, Belgium, July 1998.

[53] Ohata, F., Hirose, K., Fujii, M. and Inoue, K.: A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information. Proceedings of the 8th Asia Pacific Software Engineering Conference (APSEC 2001), pp.273-280, Macau SAR, China, December 2001.

[54] Ottenstein, K. J. and Ottenstein, L. M.: The Program Dependence Graph in a Software Development Environment. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp.177–184, Pittsburgh, Pennsylvania, April 1984.

[55] Pawlak, R., Seinturier, L., Duchien, L. and Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java. Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION 2001), pp.1-24, Kyoto, Japan, September 2001.

[56] Perkins, J. H. and Ernst, M. D.: Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. Proceedings of the 12th Symposium on the Foundations of Software Engineering (FSE 2004), pp.23-32, Newport Beach, Callifornia, USA, November 2004.

[57] Pressman, R.S: Software Engineering A Practitioner's Approach (Fourth Edition). McGraw-Hill Companies, 1997.

[58] Rashid, A. and Chitchyan, R.: Persistence as an Aspect. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp.120-129, Boston, Massachusetts, USA, March 2003.

[59] Reiss, S. P. and Renieris, M.: Encoding Program Executions. Proceedings of the 23th International Conference on Software Engineering (ICSE 2001), pp.221-230, Toronto, Ontario, Canada, May 2001.

[60] Reiss, S. P.: Dynamic Detection and Visualization of Software Phases. Proceedings of Workshop on Dynamic Analysis (WODA 2005), St. Louis, MO, USA, May 2005.

[61] Robson, D. J., Bennet K. H., Cornelius, B. J. and Munro, M.: Approaches to Program Comprehension. Jounral of Systems and Software, Vol.14, No.2, pp.79-84, February 1991.

[62] Rosenblum, D. S.: A Practical Approach to Programming with Assertions. IEEE Transactions on Software Engineering, Vol.21, No.1, pp.19-31, January 1995.

[63] Sereni, D. and de Moor, O.: Static Analysis of Aspects. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD2003), pp.30-39, Boston, Massachusetts, USA, March 2003.

[64] Siedersleben, J.: Errors and Exceptions - Rights and Responsibilities. Proceedings of Workshop on Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms, pp.2-9, Darmstadt, Germany, July 2003.

[65] Soares, S., Laureano, E. and Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. Proceedings of the 17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA2002), pp.174-190, Seattle, Washington, USA, November 2002.

[66] Störzer, M. and Krinke, J.: Interference Analysis for ApsectJ. Proceedings of Workshop on Foundations of Aspect-Oriented Languages 2003 (FAOL 2003), pp.35-44, Massachusetts, Boston, USA, March 2003.

[67] Stroustrup, B.: The C++ Programming Language (Third Edition), Addison-Wesley Pub Co., Boston, Massachusetts, USA, 1997.

[68] Takada, T., Ohata, F. and Inoue, K.: Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information. Proceedings of

the 10th International Workshop on Program Comprehension (IWPC 2002), pp.169-177, Paris, France, June 2002.

[69] Tarjan, R. E.: Depth First Search and Linear Graph Algorithms. SIAM Journal on Computing, Vol.1, No.2, pp.146-160, June 1972.

[70] Tonella, P. and Ceccato, M.: Migrating Interface Implementations to Aspects. Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), pp.220-229, Chicago, Illinois, USA, September 2004.

[71] Ueda, R., Inoue, K. and Iida, H.: A Practical Slice Algorithm for Recursive Programs. Proceedings of the International Symposium on Software Engineering for the Next Generation, pp.96-106, Nagoya, Japan, February 1996.

[72] Umemori, F., Konda, K., Yokomori, R. and Inoue, K.: Design and Implementation of Bytecode-based Java Slicing System. Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation (SCAM 2003), pp.108-117, Amsterdam, The Netherlands, September 2003.

[73] Lopes, C. V. and Ngo, T. C.: Unit-Testing Aspectual Behavior. Proceedings of Workshop on Testing Aspect-Oriented Programs 2005 (WTAOP 2005), Chicago, Illinois, March 2005.

[74] Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering, Vol.10, No.4, pp.352-357, July 1984.

[75] Yacoub, S., Ammar, H. and Robinson, T.: Dynamic Metrics for Object-Oriented Designs, Proceedings of the 6th International Symposium on Software Metrics (METRICS 1999), pp.50-61, Boca Raton, Florida, USA, November 1999.

[76] Yamada, K. and Watanabe, T.: Moxa: An Aspect-Oriented Approach to Modular Behavioral Specifications. Proceedings of Workshop on Software-Engineering Properties of Languages and Aspect Technologies 2005 (SPLAT 2005), http://www.daimi.au.dk/~eernst/splat05/, Chicago, Illinois, March 2005.

[77] Zhao, J.: Slicing Aspect-Oriented Software. Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC2002), pp.251-260, Paris, France, June 2002.

[78] Zhao, J. and Rinard, M.: Pipa: A Behavioral Interface Specification Language for AspectJ. Proceedings of the 6th International Conference on Funda-

mental Approaches to Software Engineering (FASE 2003), pp.150-165, Warsaw, Poland, April 2003.

[79] AJDT: AspectJ Development Environment. http://www.eclipse.org/ajdt/

[80] AspectJ Project. http://eclipse.org/aspectj/

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

[82] Contract4J. http://www.contract4j.org/

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

[84] Java Platform Debugger Architecture. http://java.sun.com/j2se/1.4/docs/guide/jpda/architecture.html

[85] Java Virtual Machine Tool Interface. http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jvmti/

[86] Performance Comparison of JIT. http://www.shudo.net/jit/perf/index.html

[87] JML Reference. http://www.dc.fi.udc.es/ai/tp/jml/JML/docs/jmlrefman/jmlrefman/

[88] Examples of JML Specifications. http://www.cs.iastate.edu/~leavens/JML/examples.shtml

[89] JUnit. http://junit.org/

[90] Racc: LALR(1) Parser Generator for Ruby. http://www.loveruby.net/en/prog/racc.html