Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique

Takashi Ishio, Shinji Kusumoto, Katsuro Inoue
Graduate School of Information Science and Technology,
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
+81 6 6850 6571
{t-isio, kusumoto, inoue}@ist.osaka-u.ac.jp

Abstract

One of the issues in software evolution is debugging. Debugging large and complex software systems evolved requires a lot of effort since it is very difficult to localize and identify faults. Therefore, reducing the effort of debugging process is an important step towards efficient software evolution. Program slicing, especially dynamic slicing, has been proposed to efficiently localize faults in a procedural program and an object-oriented program. Although several tools have been developed for Java programs, these are difficult to maintain because of the frequent revision of Java languages. Aspect-Oriented Programming (AOP) is a new technology for the separation of concerns in program development. Using AOP, modularizing crosscutting aspects of a system is possible. One useful application of AOP is for modularizing the collecting program's dynamic information for program analysis. Since the collection of dynamic information affects the over-all target program, this functionality is a typical crosscutting concern. In this paper, we apply AOP to develop a program debugging tool using program slicing. First, we examine the application of AOP for collecting dynamic information from program execution and for calculating program slices. Next, we develop a program slicing system using AspectJ. Finally, we describe the benefits, usability, and cost effectiveness of a module of dynamic analysis based on AOP.

1. Introduction

Software evolution generally means that software can change its structure and functions to tolerate changes of its specification and operating environment in which it is used[1]. Software is often modified to reflect new functionality with the changes of its specification. In the modification, several bugs are usually injected and so debugging is an important task in software evolution. However, debugging large and complex software systems evolved requires a lot of effort since it is very difficult to localize and identify faults. Therefore, reducing the effort of debugging process is an important step towards efficient software evolution.

Program slicing is a very promising approach to localize faults efficiently in a program[19]. 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 this paper, we label a program slice simply as a *slice*.

In recent software development, a programmer uses not only procedural languages like C and Pascal but also Object-Oriented languages like Java [9] and C++[10]. Since Object-Oriented languages include new concepts such as *class*, *inheritance*, *dynamic binding* and *polymorphism*[11], Object-Oriented programs have many dynamically-determined elements. In the slice calculation process, observing program execution, and using information about statements actually executed is effective. *Dependence-Cache (DC) slicing* has been proposed for use in a dynamic data dependence analysis and a static control dependence analysis to calculate accurate slices with lightweight costs [3, 16]. Ohata $et\ al.$, for example, extends the DC slicing method for Object-Oriented languages [14].

In the process of DC slice calculation of Java languages, how to analyze dynamic data dependence is an important issue. An analyzer implements a function that observes a target program to track and to collect information about dynamic data dependence. In past research, such a function has not been encapsulated in a single module. Instead, the function has been implemented as a pre-processor, which inserts analysis operations in the target program code [14], or as a customized Java Virtual Machine (JVM) [4]. In implementing and maintaining the rules of conversion, however, the former approach is difficult, and the latter approach is expensive because JVM must be re-customized when new versions are released.

On the other hand, Aspect-Oriented Programming (AOP) proposes a new module unit, or *aspect*, for encapsulating crosscutting concerns, such as logging and synchronization [2]. Since such concerns crosscut objects, program codes implementing such concern must be distributed among objects in Object-Oriented Programming. In AOP, one concern can be written in a single aspect.

AOP appears usable and useful, but actual examples that show the usefulness of applying AOP to program development are few. One useful application of AOP is to modularize the program's ability for collecting dynamic information for program analysis. Dynamic information, in short, is a series of program executions and is useful to analyse dynamic data dependence for DC slice calculation [3].

In this paper, we introduce an AOP for encapsulating dynamic program analysis into an aspect and for achieving a cost-effective DC slice calculation. We implement a DC slice calculation system using AspectJ [17], and conduct an experiment to evaluate the usefulness of our approach compared to a customized JVM approach. As a result, we confirm that the AOP approach can greatly reduce the cost needed for calculating DC slices, and we achieve a practical precision for the slice.

The structure of this paper is as follows: In Section 2, we describe the DC slicing. In Section 3, we present a briefly overview of Aspect-Oriented Programming and our approach to DC slice calculation using AOP. In Section 4, we describe the implementation of DC slicing tool. In Section 5, we evaluate the proposed method and compare our method to the customized JVM approach and discuss experimental results. In Section 6, we conclude our discussion with remarks regarding plans for future work.

2. Program Slicing

Program slicing is a promising approach for program debugging, testing, and understanding [19]. 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 [7].

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 "node V_t, which depends on node V_s". PDG also includes special nodes which represent method call and parameter passing [8].

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 $CD(s_1, s_2)$ or $s_1 \dashrightarrow s_2$.

***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 $DD(s_1, v, s_2)$ or $s_1 \xrightarrow{v} s_2$.

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 the slicing criterion specified by the user. In order to calculate the slice for a

slicing criterion $<s, v>$, PDG nodes are traversed in reverse order from V_s (node V_s denotes statement s.). The corresponding statements to the reachable nodes during this traversal form the slice for $<s, v>$.

We can obtain sufficient information about control dependence from static analysis (from only source code). 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 [3, 14, 16].

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 [3, 14, 16].

2.2. Dynamic Data Dependence Analysis in DC Slice Calculation

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 table, or *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 shows 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.

For example, Figure 1 is a program using an array. Table 1 shows the transition of cache $C(v)$ of each variable v at each statement when the program is executed with input $c = 0$.

Table 1. Cache transition of Figure 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

```
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 1. Example program using array

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 $statement1 \xrightarrow{a[0]} statement7$ is extracted because statement 7 refers to $a[0]$ and $C(a[0]) = 1$.

Figure 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.

3. Dynamic Analysis Using Aspect-Oriented Programming for Software Evolution

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.

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 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

```
class Count {

  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("java Main [sft|inc]");
      return;
    }

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

    int x = 0;
    for (int i=0; i<1000; ++i) {
      counter.proceed();                          (b)
      x = counter.value();
      if (x > 1000) break;
      System.out.println(x);
    }

    String result;
    if (isIncrementCounter) {
      result = "increment counter = ";                      (c)
      result = result + Integer.toString(x);
    } else {
      result = "shift counter = ";
      result = result + Integer.toString(x);
    }
    System.out.println(result);   (d)
  }
}

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

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

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

Figure 2. Source program and DC slice example (slice criteria = (d), input = "inc")

program are composed by *Aspect Weaver* to construct the program with a crosscutting structure.

AspectJ is an aspect weaver for Java. 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. Advice can be united by three types of forms, *before* (immediately before join points), *after* (immediately after), and *around* (before and behind).

AspectJ 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, AspectJ can generate codes accessing the information of source codes as the context of an

Table 2. Pointcut Designators of AspectJ

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

aspect, e.g. the join points' position in the source code, and in the signature of methods.

3.2. Example of an Aspect

Here, an aspect which records dynamic bindings is shown in Figure 3, as an example of the aspect. This code records how dynamic bindings are resolved. In actuality, 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. Also, extra codes do not have to be written in objects. We can reuse the aspect and the objects independently.

3.3. Dynamic Analysis of Program Execution

The analysis of dynamic information from program execution is a technology required for both program slice calculation and the measurement of dynamic software metrics.

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 [14].

(b) Using *Java Virtual Machine Profiler Interface* (JVMPI) to collect dynamic information [15].

(c) Using Java Debugger Interface (JDI) [12] to collect dynamic information.

(d) Using customized *Java Virtual Machine* for dynamic analysis [4].

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 because of complex language factors, such

```
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 3. The aspect which records dynamic bindings

as multi-threading and exception handling is difficult. 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.

Also, in (b), (c) and (d), the program has to be analyzed at the Java bytecode level. On one hand, therefore, a bytecode optimization by a *Just In Time* (JIT) compiler usually affects the analysis result.

On the other hand, in the AOP approach, a dynamic analysis aspect can be composed by join points, which is more abstract than syntactic tree conversion rules. 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.

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 using AspectJ can be described as follows .

- **Data Dependence Analysis**

  When 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 statement information 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,

```
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 4. A piece of the implementation of dynamic data dependence analysis

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.

A piece of code where the dynamic data dependence analysis is implemented is shown in Figure 4. A polymorphism resolution is a multi-threaded extension of the code, as shown in Figure 3.

The dynamic analysis aspect uses a wildcard of AspectJ to analyze all assignments and references of a 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 aspect using inheritance in AspectJ 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 4.4.

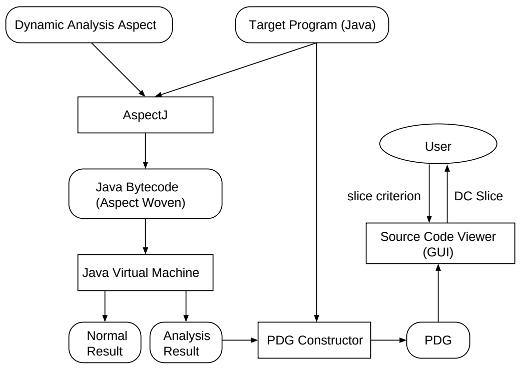

Figure 5. DC slicing system

4. Implementation

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 5 illustrates the system overview.

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

Step 1: The AspectJ compiler compiles 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.

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. Because such join points are fine grained, such join points require remarkable implementation cost, and such join points are rarely required.

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 5.2, we will discuss this issue based on the result of experimental evaluation.

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 [4].

However, even if we use the bytecode analysis, a dependence analysis to important objects, such as file I/O and basic data structures, cannot be done because of the limitations in the Java language described in Section 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. Next, 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 from a library, the aspect receives only information of the called method. Next, the aspect extracts a virtual control dependence relation between the last call to a library and the called method.

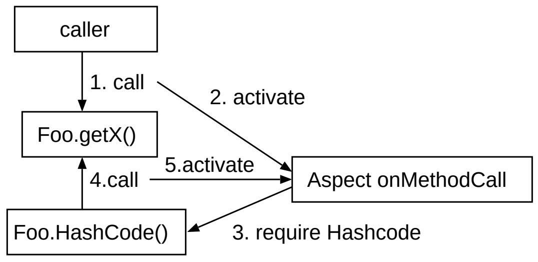

Figure 6. An example of a loop caused by an aspect

4.4. Loop Caused by Aspect

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

The example of such a loop is shown in Figure 6. In Figure 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 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 joining the aspect to their 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 3. Target programs

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

Table 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

5. Experimental Evaluation

5.1. Overview

In order to evaluate the proposed DC slice system, we have compared the DC slice system to the system developed using the customized JVM approach [4] from the viewpoint of cost and module size necessary for dynamic analysis. Since a customized JVM analyzes Java bytecodes, the JVM extracts data dependencies even in the libraries.

In the evaluation, we have used the programs shown in Table 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 5.2, we evaluate and discuss DC slice size. In Section 5.3 and 5.4 we also discuss time cost and module size necessary for DC slice calculation.

5.2. Resulting Slice Size

Here, we compare the two slicing tools from the viewpoint of resulting slice size.

Table 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. Redundant statements, however, were included. We may conclude that the difference of the slice size shows the difference of correctness.

With respect to the redundant statements, 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 non-dependent may be 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 in the conditional clause, which were not executed may be included in our approach, but not included in 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. Therefore, we believe that our approach is effective for the large scale programs.

Even if statements never executed are included in a slice using static information, no dynamic information, such as a method call and a field reference exist. Since their statements have no inter-method dependence relations, the difference of slice size is limited. In the future, we hope to treat the issue by asking how such a difference affects a task of developers.

Removing never executed statements from a slice using the information of a control flow is our future work.

5.3. Analysis Cost

Here, we evaluate the time necessary for calculating the DC slice.

Table 5 shows the time needed to execute the Java program with a normal JVM, with a customized JVM, and the

Table 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 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

program aspect which has been inserted 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 6.

In general, our approach shows good performance when compared with the customized JVM approach. We believe that the cost of a dynamic analysis of the local variables is 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 is unequal in runtime environment, this effect has experimentally been shown that JIT makes a crucial difference on system performance [13]. Improving performance is paramount because a program is executed repeatedly in the debugging process.

5.4. Effort to Implement the Slicing Tool

In this section, we examine the effort of implementing the slice tool.

A dynamic analysis module implemented as an aspect reached about 400 LOC. The total DC slice calculation tool reached about 16,000 LOC in Java.

In our approach, the aspect can be described at a highly abstracted level and has good readability compared with the pre-processor approach. Moreover, because the aspect is small and simple, the programmer (user) can easily switch to other implementation to adapt each runtime environment.

On the other hand, in the customized JVM approach, it was necessary to add about 16,000 lines of code to the JVM and Java compiler, which consists of about 500,000 LOC[5]. The additional codes consist of two parts, dynamic data analysis and source analysis. The dynamic data analysis handles local variables that the aspect does not handle. The source analysis extracts a map between source codes and byte codes. This map is needed for mapping a slice to source codes.

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.

6. Conclusion and Future Work

In this paper, we have examined an application of the aspect-oriented programming to collect dynamic information in program slicing calculation. Through the implementation of a dynamic program analysis module in AOP, we have developed a DC slice calculation system 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. Therefore, we improve maintainability and reusability of the module.

In our research, we have chosen AspectJ to implement the module. AspectJ has a restriction that does not allow us to analyze local variables and local control structures. However, compared with the customized JVM approach, we achieved cost reduction and maintainability improvement with a larger slice. Moreover, our aspect approach is not dependent on a Java-specific factor. Our method also allows possible implementation dynamic analysis using an appropriate aspect weaver for other languages.

In future work, we would like to evaluate our slicing system for large programs. We will also examine how dynamic information about local variables and local control structures affects a slice size and how such a difference affects a debugging task. Finally, we plan to examine the applicability of Aspect-Oriented Programming to other application in software development.

References

[1] T. Katayama: “A theoretical framework of software evolution”, Proceedings of International Workshop on

Principle of Software Evolution 1998, pp. 1-5 (1998).

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

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

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

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

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

[7] K. J. Ottenstein and L. M. Ottenstein: "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).

[8] R. Ueda, K. Inoue and H. Iida: "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).

[9] J. Gosling, B. Joy, and G. Steele: "The Java ^{TM} Language Specification", Addison-Weseley (1996).

[10] B. Stroustrup : "The C++ Programming Language (Third edition)", Addison-Wesley (1997).

[11] G. Booch: "Object-Oriented Design with Application", The Benjamin/Cummings Publishing Company, Inc (1991).

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

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

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

[15] S. Kusumoto, M. Imagawa, K. Inoue, S. Morimoto, K. Matsusita and M. Tsuda: "Function point measurement from Java programs", Proceedings of the 24th International Conference on Software Engineering, pp. 576-582 (2002).

[16] T. Takada, F. Ohata, K. Inoue: "Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information", Proceedings of the 10th International Workshop on Program Comprehension, pp.169-177, Paris, France, June (2002).

[17] AspectJ Team, "The AspectJ Programming Guide", http://aspectj.org/doc/dist/progguide/

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

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