A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information

Fumiaki OHATA Kouya HIROSE Masato FUJII
Katsuro INOUE

Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan

{*oohata, k-hirose, m-fujii, inoue*}*@ics.es.osaka-u.ac.jp*

Abstract

Program slicing has been used for efficient program debugging activities. Program slice is computed by analyzing dependence relations between program statements. We can divide dependence analyses into two categories, static and dynamic; the former requires little analysis costs, but the resulting slices are large. The latter has opposite characters.

In this paper, we propose a program slicing method for Object-Oriented programs and evaluate its effectiveness with some JAVA *programs. Since Object-Oriented languages have many dynamically determined elements, static analysis could not compute practical analysis results. Our method uses static and dynamic analyses appropriately and computes accurate slices with lightweight costs.*

1. Introduction

Program slicing is very promising approach for program debugging, testing, understanding, merging, and so on [3, 4, 7, 9, 17]. Given a source program p, *program slice* is a collection of statements possibly affecting the value of *slicing criterion* (a pair $<s, v>$, s is a statement in p and v is a variable defined or referred at s). Also, we call program slice simply *slice*. Slice computation is based on dependence analysis between program statements in a source program, and dependence analysis consists of two components, data dependence analysis and control dependence analysis. Many slice computation algorithms have been already proposed, and they are roughly divided into two categories, *static slicing*[17] and *dynamic slicing*[1]. The former analyzes all dependence relations *statically*, in other words, without program execution. The latter analyzes those relations *dynamically*. Since dynamic slicing focuses on the specific execution path and can grasp the values of all referred and defined variables on each execution point, its analysis precision is better than that of static slicing.

In existing software development environments, not only procedural languages like C and Pascal but also Object-Oriented languages like JAVA[8] and C++[15] become to be used. Since Object-Oriented languages have new concepts such as *class, inheritance, dynamic binding* and *polymorphism*[5], we cannot adopt existing slicing methods for procedural programs to Object-Oriented programs. Larsen $et\ al$ and Zhao proposed static and dynamic slicing methods for Object-Oriented programs, respectively[11, 18]; however, since Object-Oriented languages have many dynamically determined elements, static slicing cannot compute practical (or precise) analysis results. On the other hand, since dynamic slicing needs to record execution trace, it requires too much computation time and memory space.

In this paper, we will adopt an intermediate slicing method between static slicing and dynamic slicing named *Dependence-Cache (DC) slicing*[2] to Object-Oriented programs. DC slicing method uses dynamic data dependence analysis and static control dependence analysis, which computes more precise analysis results than static slicing and needs less analysis costs than dynamic slicing. We have implemented this method as a slicing system, whose target language is JAVA.

The structure of this paper is as follows:

In Section 2, we will briefly overview program slice and DC slice. In Section 3, we propose extended DC slice for Object-Oriented programs. In Section 4 and 5, we evaluate the proposed method using our implementation with some JAVA programs, and discuss experimental results, respectively. In Section 6, we conclude our discussion with a few remarks regarding plans for future work.

2. Dependence-Cache (DC) Slice

In this section, we will briefly explain the computation process of program slice, and introduce DC slice on which our proposed method is based.

2.1. Program Slice

[**Slice Computation Process**]

In general, slice computation process 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 *Program Dependence Graph (PDG)* using dependence relations extracted on Phase 2.

Phase 4: Slice Extraction
We compute the slice for the slicing criterion specified by the user. In order to compute 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 $<s, v>$.

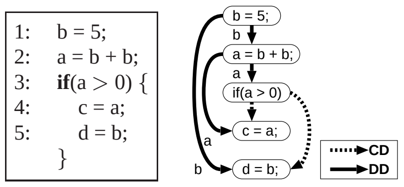
[**Dependence Relation**]

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:

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

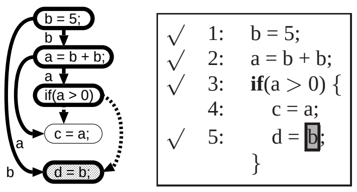
This relation is written by $CD(s_1, s_2)$ or $s_1 \dashrightarrow s_2$. When the following conditions are all satisfied, we say that a *data dependence (DD)*, from statement s_1 to statement s_2 by a variable v exists:

1. s_1 defines v, and
2. s_2 refers v, and
3. at least one execution path from s_1 to s_2 without redefining v exists (we call this condition *reachable*).



```
1:   b = 5;
2:   a = b + b;
3:   if(a > 0) {
4:      c = a;
5:      d = b;
     }
```

Figure 1. Sample C program and its PDG



```
√  1:   b = 5;
√  2:   a = b + b;
√  3:   if(a > 0) {
   4:      c = a;
√  5:      d = b;
        }
```

Figure 2. Slice for <5, b> on Figure 1

This relation is denoted by $DD(s_1, v, s_2)$ or $s_1 \xrightarrow{v} s_2$.

[**Program Dependence Graph (PDG)**]

A PDG is a directed graph whose nodes represent statements in a source program, and whose edges denote dependence relations (DD or CD) between statements. A DD edge is labeled with a variable name "a" if it denotes $DD(\cdots, a, \cdots)$. An edge drawn from node V_s to node V_t represents that "node V_t depends on node V_s".

[**Example**]

Figure 1 shows a sample C program and its PDG (**Phase 1 – 3**), and Figure 2 shows the slice ("$\surd$"-marked statements) for $<$5, b$>$ on Figure 1 (**Phase 4**).

2.2. Dependence-Cache (DC) Slice

When we statically analyze source programs that have array variables, too many DD relations might be extracted. This is because it is difficult for us to determine the values of array indices without program execution if they are not constant values but variables — *array indices problem*.

Also, in the case of analyzing source programs that have pointer variables, *aliases* (an expression refers to the memory location which is also referred to by another expression) resulting from pointer variables might generate implicit DD relations. In order to analyze such relations, pointer analysis should be needed. Many researchers have already proposed static pointer analysis methods[6, 14, 13]; however, it is difficult for static analyses to generate practical analysis results — *pointer alias problem*.

DC slicing uses dynamic DD analysis, so that it can resolve above *array indices problem* and *pointer alias problem*. Since dynamic DD analysis is based on program execution, we can extract the values of all variables on each execution point. On the other hand, since DC slicing uses static CD analysis, we need not record execution trace and its analysis cost is much less then that of dynamic slicing (dynamic slicing uses dynamic DD and CD analyses).

[**DC Slice Computation Process**]

Computation process for DC slice is as follows.

Phase 1: Defined and Referred Variables Extraction

Phase 2: Static Control Dependence Analysis and PDG Construction
We extract CD relations statically between statements, and construct PDG that has CD edges only.

Phase 3: Dynamic Data Dependence Analysis and PDG Edge Addition
We execute a source program. On program execution, we extract DD relations dynamically between statements using the following method, and add DD edges to PDG.

Phase 4: Slice Extraction

[**Dynamic Data Dependence Analysis**]

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

Step 1: We create 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 proceed the following methods on each execution point.
On executing statement s,

- when variable v is referred, we draw an 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.

[**Comparison with Static Slice and Dynamic Slice**]

Table 1 shows differences among static slice and dynamic slice and DC slice.

As an example, Figure 3 shows static, dynamic and DC slices for slicing criterion <23, `d`>; for dynamic and DC slices, we passed integer value "2" to **scanf**() statement on program execution. "$\surd$" represents a sliced statement, and "`S`", "`D`" and "`DC`" represents static, dynamic and DC slices, respectively. In this case, dynamic slicing and DC slicing compute the same slices.

Table 1. Comparison with static slice and dynamic slice

	Static Slice	Dynamic Slice	DC Slice
CD	static	dynamic	static
DD	static	dynamic	dynamic
PDG node	statement	execution point	statement

3. Object-Oriented Dependence-Cache (OODC) Slice

In this section, we will propose *Object-Oriented Dependence-Cache (OODC) Slice*, which is an extended DC slice for Object-Oriented programs.

[**Analysis Policy**]

Object-Oriented languages have the following characters.

Chr.1: Object is a collection of attributes and methods that operate attributes.

Chr.2: Dynamic binding feature exists.
Dynamic binding – based on the class type of an object, an appropriate override method is selected and invoked

For **Chr.1**, at the same time that a variable is created, the corresponding cache is also created. Using this rule, we can analyze each object independently even if they are instantiated from the same class.

For **Chr.2**, static analysis cannot always identify invoked methods without program execution (it is difficult for static pointer or alias analysis to identify the unique class type of each object that is referred to by pointer or reference variables; in general, two or more candidates exist). OODC slice also dynamically analyses CD relations about method invocation for more precise analysis results.

[**Algorithm**]

Figure 4 shows OODC slicing algorithm.

On **Step 1**, we construct PDG with no edge. On **Step 2**, we *statically*

- extract CD relations except those about method invocation, and add CD edges to PDG.

On **Step 3**, we *dynamically*

- extract DD relations and add DD edges to PDG, and
- extract CD relations about method invocation and add CD edge to PDG,

respectively; algorithm for dynamic DD analysis is shown in Figure 5.

S	D	DC		
√	√	√	1:	**#include** <stdio.h>
√	√	√	2:	**#define** SIZE 5
√	√	√	3:	**int** cube(**int** x) {
√	√	√	4:	**return**(x * x * x);
√	√	√	5:	}
√	√	√	6:	**void** main(**void**)
√	√	√	7:	{
√	√	√	8:	**int** a[SIZE];
√	√	√	9:	**int** b[SIZE];
√	√	√	10:	**int** c, d, i;
√			11:	a[0] = 0;
√			12:	a[1] = −1;
√	√	√	13:	a[2] = 2;
√			14:	a[3] = −3;
√			15:	a[4] = 4;
√	√	√	16:	**for**(i = 0; i < SIZE; i++) {
√	√	√	17:	b[i] = a[i];
√	√	√	18:	}
√	√	√	19:	**scanf**("%d", &c);
√	√	√	20:	d = cube(b[c]);
√	√	√	21:	**if**(d < 0)
√			22:	d = −1 * d;
√	√	√	23:	**printf**("%d", d);
√	√	√	24:	}

Figure 3. Static, Dynamic and DC slices for slicing criterion <23, d>

```
INPUTS
    $p$:  Program
    $\mathcal{I}$:  Inputs for $p$'s execution
    $<s_c, v_c>$:  Slicing criterion
OUTPUTS
    $PDG_{(p,\mathcal{I})}$:  PDG for $(p, \mathcal{I})$
    $\mathcal{S}$:  OODC slice for $<s_c, v_c>$
ALGORITHM
  Step 1: [ Create each $PDG_{(p,\mathcal{I})}$'s node $V(s)$ for statement $s$ in $p$ ]
  Step 2: foreach $s \in p$ do
            if $s$ is a conditional (or loop) statement then
              $s_1$ := $s$'s conditional expression
              $s_2$ := $s$'s branch statement (or loop body)
              [ Add "$V(s_1) \dashrightarrow V(s_2)$" to $PDG_{(p,\mathcal{I})}$ ]
            fi
          done
  Step 3: until $p$ with $\mathcal{I}$ terminates do
            $s$ := next execution statement
            [ Analyze dynamic DD relations for $s$ (see Figure 5) ]
            if $s$ is a method invocation statement then
              $C := V(s)$
            elif $s$ is a method declare statement then
              [ Add "$C \dashrightarrow V(s)$" to $PDG_{(p,\mathcal{I})}$ ]
            fi
            [ Execute $s$ ]
          done
  Step 4: $\mathcal{S} := \{V(s_c)\}, \mathcal{N} := \phi$
  Step 5: $\mathcal{N} := \{ n \mid n \xrightarrow{v} s_c \} \cup \{ m \mid m \dashrightarrow s_c \}$
  Step 6: while $\mathcal{N} \neq \phi$ do
            $\{n\} \cup \mathcal{N}' := \mathcal{N}$
            $\mathcal{S} := \mathcal{S} \cup \{ n \}$
            $\mathcal{N} = \mathcal{N}' \cup \{ m \mid m \notin \mathcal{S} \wedge$
                                  $(^{\exists}w(m \xrightarrow{w} n) \vee m \dashrightarrow n)\}$
          done
```

Figure 4. Algorithm for OODC slicing

On **Step 4 – 6**, we start PDG traversal in reverse order from the slicing criterion node, so that OODC slice would be extracted.

[**Example**]

Figure 6 shows a sample program and OODC slice ("√"-marked statements) for slicing criterion <16, c>. Since OODC slice needs program execution, we have executed this program as follows[1]:

```
% javac Main.java
% java Main -1
```

Table 2 shows a cache history for Figure 6. Cache $C(v)$ represents a statement on which variable v is defined using assignment expressions, parameter passing, and so on. "-" means "not defined yet". For example, at statement 16, variable c is referred and $C(c)$ is "14", so that we can extract DD(14, c, 16).

On program execution, we need not to record all cache history. In order to extract dynamic DD relations, we have only to focus on the values of caches at the executed statement. About implementation of caches, we will describe later.

[1]We have passed "−1" to the first parameter.

4. Implementation

We have implemented the proposed method as a slicing system for JAVA, which consists of two components, analysis libraries and *Graphical User Interface (GUI)*. They are also written in JAVA.

Figure 7(a) shows the design of our implementation. In Figure 7(a), Objects that have gray background represent the components we have developed.

[**Implementation of Analysis Libraries**]

We have adopted preprocessor style for implementation of analysis libraries. *Preprocessor style* means that before program execution, we add some JAVA codes to target

```
INPUTS
    $p$:  Program
    $\mathcal{I}$:  Inputs for $p$'s execution
    $s$:  Statement
    $\mathcal{C}$:  Cache set
    $PDG_{(p,\mathcal{I})}$:  PDG for ($p$, $\mathcal{I}$)
OUTPUTS
    $PDG_{(p,\mathcal{I})}$:  PDG for ($p$, $\mathcal{I}$)
    $\mathcal{C}$:  Cache set
ALGORITHM
    if variable $v$ is declared at $s$ then
        [ Create $C(v)$ ]
        $C(v) := V(s)$
        $\mathcal{C} := \mathcal{C} \cup \{ C(v) \}$
    elif variable $v$ is defined at $s$ then
        $C(v) := V(s)$
    elif variable $v$ is referred at $s$ then
        [ Add "$C(v) \xrightarrow{v} V(s)$" to $PDG_{(p,\mathcal{I})}$ ]
    fi
```

Figure 5. Algorithm for dynamic DD analysis

source program p, and on program execution, such codes dynamically analyze dependence relations between statements in p.

Interpreter style would also be a candidate for their implementation; however, we have to customize an existing JAVA *Virtual Machine (JavaVM)* or develop a JAVA interpreter, and too much execution time would be required.

Since it is easy to develop a preprocessing environment[10] and we can use existing *Just-in-Time (JIT)* compilers to optimize preprocessed programs, preprocessor style would be a more promising approach.

[**Implementation of Caches**]

We have used the following rules to implement caches:

- On each class, we consider cache $C(v)$ for instance variable v as an instance variable.
- On each class, we consider cache $C(v)$ for class variable v as a class variable.
- On each scope, we consider cache $C(v)$ for local variable v as a local variable.

Using above rules, when two objects a and b are instantiated from the same class, we can independently trace caches $C(a.v)$ and $C(b.v)$ for instance variables $a.v$ and $b.v$. When instance variable v is inherited, $C(v)$ is also inherited.

[**Implementation of Graphical User Interface (GUI)**]

GUI has the following features:

- Edit source programs

```
√  1: import java.util.*;
√  2: import java.io.*;
   3:
√  4: class Main {
√  5:     static Base b1;
√  6:     public static void main(String[] args)
               throws IOException {
√  7:         int c;
   8:         Base b2 = new Derived();
√  9:         int i = Integer.parseInt(args[0]);
√ 10:         if(i < 0)
√ 11:             b1 = new Base();
  12:         else
  13:             b1 = new Derived();
√ 14:         c = b1.m(i);
  15:         b2.set(c);
√ 16:         System.out.println(c);
  17:         System.out.println(b1.a);
√ 18:     }
√ 19: }
√ 20:
√ 21: class Base {
√ 22:     public int a = 10;
√ 23:     public int m(int i) {
√ 24:         return(a − i);
√ 25:     }
  26:     public void set(int i) {
  27:         a = i;
  28:     }
√ 29: }
  30:
  31: class Derived extends Base {
  32:     public int m(int i) {
  33:         return(a + i);
  34:     }
  35: }
```

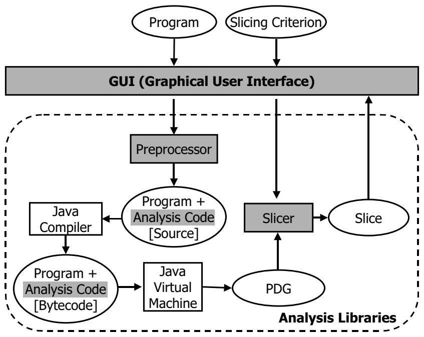
Figure 6. OODC slice for slicing criterion <16, `c`>

- Control analysis libraries (PDG construction, slice computation, program execution, and so on)
- Show slice results.

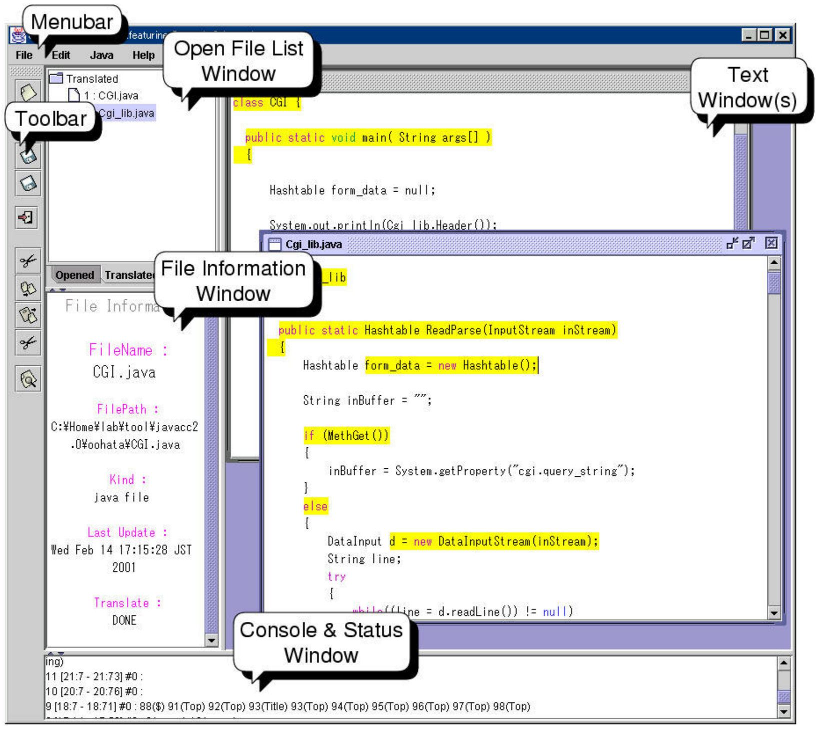
Figure 7(b) shows a screenshot of GUI part.

The following shows an example of slice computation using our GUI.

1. open JAVA source program p :
   `Open File List Window` shows p's name, `Text Window` shows the source text of p, and `File Information Window` shows p's status.
2. translate p to preprocessed JAVA source program p' :
   p' contains additional JAVA codes in order to analyze dependence relations dynamically between statements in p.
3. construct PDG_p for p:
   PDG_p has CD edges only (all DD edges and CD



(a) Design



```
class CGI {

  public static void main( String args[] )
  {

     Hashtable form_data = null;

     System.out.println(Cgi_lib.Header());
```

```
File Information

FileName :
CGI.java

FilePath :
C:¥Home¥lab¥tool¥javacc2
.0¥oohata¥CGI.java

Kind :
java file

Last Update :
Wed Feb 14 17:15:28 JST
2001

Translate :
DONE
```

```
_lib

  public static Hashtable ReadParse(InputStream inStream)
  {
     Hashtable form_data = new Hashtable();

     String inBuffer = "";

     if (MethGet())
     {
        inBuffer = System.getProperty("cgi.query_string");
     }
     else
     {
        DataInput d = new DataInputStream(inStream);
        String line;
        try
        {
           while((line = d.readLine()) != null)
```

```
ing)
11 [21:7 - 21:73] #0 :
10 [20:7 - 20:76] #0 :
9 [18:7 - 18:71] #0 : 88($) 91(Top) 92(Top) 93(Title) 93(Top) 94(Top) 95(Top) 96(Top) 97(Top) 98(Top)
```

(b) Screenshot of GUI

Figure 7. Slicing system for JAVA

edges about method invocation will be added on the next step).

4. execute p' :
   We collect dynamic DD relations and add the corresponding DD edges to PDG_p. Also, we collect CD relations about method invocation and add the corresponding CD edges to PDG_p. `Console & Status Window` shows p''s execution results and some debugging messages.

5. slice computation:
   The user specifies a slicing criterion, and we start PDG traversal in reverse order from the node corresponding to the slicing criterion. Statements in the resulting slice are highlighted with colored background on `Text Window(s)`. In the case of Figure 7(a), they are distributed on two source files.

5. Evaluation

5.1. Metrics

Using our slicing system, we have evaluated the proposed method. Table 3 shows the features of sample programs we have used; P_1 loads some data from text files and generates HTML files, P_2 is a paint application using a mouse.

Also, we have used the following metrics:

Slice Size: Comparison with static slice and dynamic slice [Table 4]
Since we have implemented OODC slicing method only, we compute static slice and dynamic slice by hand.

Execution Time: Comparison between before and after preprocessing (adding analysis codes) [Table 5]
Since P_2 is a dialogue application, we could not record its execution time.

Memory Use: Comparison between before and after preprocessing [Table 6]

Table 2. Cache history for Figure 6

St.	$C(\texttt{b1})$	$C(\texttt{b2})$	$C(\texttt{c})$	$C(\texttt{i})$	$C(\texttt{b1.a})$	$C(\texttt{b2.a})$	$C(\texttt{args})$	$C(\texttt{args[0]})$
4	-	-	-	-	-	-	-	-
5	5	-	-	-	-	-	-	-
6	5	-	-	-	-	-	6	6
7	5	-	7	-	-	-	6	6
8	5	8	7	-	-	-	6	6
31	5	8	7	-	-	-	6	6
21	5	8	7	-	-	-	6	6
22	5	8	7	-	-	22	6	6
9	5	8	7	9	-	22	6	6
10	5	8	7	9	-	22	6	6
11	11	8	7	9	-	22	6	6
21	11	8	7	9	-	22	6	6
22	11	8	7	9	22	22	6	6
14	11	8	14	9	22	22	6	6
23	11	8	14	9	22	22	6	6
24	11	8	14	9	22	22	6	6
15	11	8	14	9	22	22	6	6
26	11	8	14	9	22	22	6	6
27	11	8	14	9	22	27	6	6
16	11	8	14	9	22	27	6	6
17	11	8	14	9	22	27	6	6

Table 3. Target programs

Program	Classes	Override Methods	Lines
P_1	2	0	223
P_2	3	7	226

Table 4. Slice size [lines (slice/total)]

Slicing Criterion	Static	Dynamic	OODC
$P_1(1)$	26(11.7%)	15(6.7%)	15(6.7%)
$P_1(2)$	83(37.2%)	27(12.1%)	27(12.1%)
$P_1(3)$	37(16.6%)	24(10.8%)	24(10.8%)
$P_2(1)$	48(21.2%)	14(6.2%)	14(6.2%)
$P_2(2)$	45(19.9%)	12(5.3%)	12(5.3%)
$P_2(3)$	25(11.1%)	17(7.5%)	17(7.5%)

Table 5. Execution time [ms]

Program	Before(T_A)	After(T_B)	T_B / T_A
P_1	138	582	4.22

Celeron-500MHz(128MB) / Windows98SE / JDK 1.3.0_01(HotSpot)

Table 6. Memory use [KByte]

Program	Before(T_A)	After(T_B)	T_B / T_A
P_1	478	645	1.35
P_2	836	920	1.10

5.2. Discussions

The size of OODC slice is 20–70% as large as that of static slice, so that we can say that OODC slice is more precise than static slice [Table 4]. Since target programs are small, their precision difference are also small; however, we guess that precision difference would become wider for larger programs on which class inheritance and method overriding occur frequently.

On the other hand, additional costs for dynamic dependence analyses are small and practical [Table 5, Table 6].

Since we have not implemented dynamic slicing method and static slicing method for JAVA yet, we could not compare analysis costs among them; however, about analysis costs and precision, the following characteristics had been indicated by experimental results for programs written in procedural language Pascal[2]. These characteristics would be also satisfied on JAVA programs.

Analysis Time (Costs): Dynamic $\gg$ DC $>$ Static

Slice Size (Precision): Static $\geq$ DC $\geq$ Dynamic

5.3. Related Works

Larsen *et al* and Liang *et al* proposed static slicing methods for Object-Oriented programs[11, 12]. Since static analysis need not program execution, analysis costs might be small; however, Object-Oriented languages have many dynamically determined elements, such as polymorphism, dynamic binding, exceptions, and so on. Furthermore, we also have to analyze alias relations with pointer variables and reference variables for C++ and JAVA programs, respectively; alias relations should be resolved before DD analysis. Although Steensgaard and Tonella *et al* proposed static alias analysis methods[14, 16], it is difficult for us to compute practical analysis results statically.

Zhao proposed a dynamic slicing method for Object-Oriented programs[18]. Dynamic slicing methods would generate more practical results than static slicing methods; however, since it requires too much computation time and memory space to record execution trace, we cannot analyze large programs dynamically. Also, [18]'s method is proposed only and have not been implemented yet.

Asida *et al* proposed DC slice that was originally named *Dynamic Data Dependence (D^3) slice*, and they implemented a slice system for Pascal[2]. Our slicing method is based on their work; however, [2] focuses on ordinary procedural languages only, and their implementation is not for practical use, but a prototype only. Our proposed method takes Object-Oriented languages into account, and we have implemented a slicing system for JAVA that is used by many software developers.

6. Summary and Future Work

In this paper, we have proposed a slicing method for Object-Oriented programs, which is an intermediate method between static slicing and dynamic slicing. Since proposed method dynamically analyzes all DD relations and CD relations about method invocations, its analysis precision is better than that of static slicing. On the other hand, since it statically analyzes CD relations except method invocations, its analysis costs is less than that of dynamic slicing. Also, we have implemented our method as a slicing system for JAVA, and we have evaluated its effectiveness.

Since JAVA has other dynamically determined elements such as multi-thread and exception, we are planning to analyze CD relations about them dynamically. Also, we are going to evaluate our method for large programs.

Acknowledgments

This work is partly supported by Ministry of Education, Science, Sports, and Culture, grant-in-aid for priority areas "Principles for Constructing Evolutionary Software", #10139223.

References

[1] Agrawal, H., and Horgan, J. : "Dynamic Program Slicing", *SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, 1990.

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

[3] Beck, J. and D, Eichmann. : "Program and interface slicing for reverse engineering", *Proceedings of the 15th International Conference on Software Engineering*, pp. 509–518, Baltimore, Maryland, May 1993.

[4] Bates, S. and Horwitz, S. : "Incremental program testing using program dependence graphs", *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pp. 384–396, Charleston, South Carolina, January 1993.

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

[6] Enami, M., Ghiya, R., and Hendren, L. J. : "Context-sensitive interprocedural points-to analysis in the presence of function pointers", *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 242–256, Orlando, Florida, June 1994.

[7] Gallagher, K. B. and Lyle, J. R. : "Using program slicing in software maintenance", *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, 1991.

[8] Gosling, J., Joy, B. and Steele, G. : "The JAVA™ Language Specification", Addison-Weseley, 1996.

[9] Horwitz, S. and Reps, T. : "The Use of Program Dependence Graphs in Software Engineering", *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, Melbourne, Australia, May 1992.

[10] "JavaCC", `http://www.webgain.com/products/metamata/java_doc.html`

[11] Larsen L. D. and Harrold, M. J. : "Slicing Object-Oriented Software", *Proceedings of the 18th International Conference on Software Engineering*, pp. 495–505, Berlin, Germany, March 1996.

[12] Liang, D. and Harrold, M. J. : "Slicing Objects Using System Dependence Graphs", *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 358–367, Bethesda, Maryland, November 1998.

[13] Shapiro, M. and Horwitz, S. : "Fast and accurate flow-insensitive point-to analysis", *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 1–14, Paris, France, January 1997.

[14] Steensgaard, B. : "Points-to analysis in almost linear time", *Technical Report MSR-TR-95-08*, Microsoft Research, 1995

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

[16] Tonella, P., Antoniol, G., Fiutem, R., and Merlo, E. : "Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing", *Proceedings of the 19th International Conference on Software Engineering*, Boston, Massachusetts, pp. 433-443, May 1997.

[17] Weiser, M. : "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, San Diego, California, March 1981.

[18] Zhao, J. : "Dynamic Slicing of Object-Oriented Programs", *Technical Report SE-98-119*, *Information Processing Society of Japan*, pp. 17–23, May 1998.