Design and Implementation of Bytecode-based Java Slicing System

Fumiaki Umemori[†], Kenji Konda[††], Reishi Yokomori[††], Katsuro Inoue[†]
[†]Graduate School of Information Science and Technology, Osaka University
[††]Graduate School of Engineering Science, Osaka University
{umemori,inoue}@ist.osaka-u.ac.jp,
{konda,yokomori}@ics.es.osaka-u.ac.jp

Abstract

Program slice is a set of statements that affect the value of variable v in a statement s. In order to calculate a program slice, we must know the dependence relations between statements in the program. Program slicing techniques are roughly divided into two categories, static slicing and dynamic slicing, and we have proposed DC slicing technique which uses both static and dynamic information.

In this paper, we propose a method of constructing a DC slicing system for Java programs. Java programs have many elements which are dynamically determined at the time of execution, so the DC slicing technique is effective in the analysis of Java programs. To construct the system, we have extended Java Virtual Machine for extraction of dynamic information. We have applied the system to several sample program to evaluate our approach.

1 Introduction

Program slice is a set of statements that affect the value of variable v in a statement s. Program slicing is a very promising approach for program debugging, testing, understanding, merging, and so on[5, 6, 9, 11, 20]. We have empirically investigated effectiveness of program slicing for program debugging and program maintenance processes, and its significance was validated by several experiments[12].

In order to calculate a program slice, we must know the dependence relations between statements in the program. Program slicing techniques are roughly divided into two categories, static slicing[13, 20] and dynamic slicing[1, 21]. The former is based on static analysis of source program without input data. The dependence of program statements is investigated for all possible input data. The latter is based on dynamic analysis with a specific input data, and the dependence of the program statements is explored for the program execution with the input data. The size of the static slice is larger than that of the dynamic one in general, since the static slice considers all possible input data. The size of the dynamic slice is smaller in general, but the dynamic one requires a large amount of CPU time and memory space to obtain it.

We thought that using both static and dynamic information would be better than using only dynamic information, and have proposed DC slicing technique which uses both static and dynamic information, and have shown that we can obtain suitable compromises of slice precision and slicing performance.

In software development environment in recent years, object-oriented languages, such as Java, are used in many cases. Although we would like to adapt the program slicing techniques to Java programs, Java programs have many features which are dynamically determined at the time of execution. Therefore, applying the static slicing technique to the object-oriented languages will cause a problem in slice precision. Also the dynamic slicing has a problem in analysis cost. We consider that the DC slicing technique is effective in the analysis of Java programs.

In this paper, we propose a method of constructing a DC slicing system for Java programs. To construct the system, we extended Java Virtual Machine for extraction of dynamic information. Since the execution is on bytecode, we define the slice calculation method on bytecode.

This DC slicing system consists of 4 subsystems, an extended **Java Compiler** that can generate a cross reference table between the source code and the bytecode, an extended **Java Virtual Machine(JVM)** that can perform the dynamic data dependence analysis for the bytecode, a **static control dependence analysis tool** for the bytecode, and a **slicer**. A slice in the bytecode calculated by the slicer is mapped onto a slice in the source code by using the cross reference table .

In section 2, we will briefly overview program slicing. In section 3, we will present a method of constructing a DC slicing system, and discuss an implementation of the sysytem. In section 4, we will evaluate the proposal method

by comparison with traditional slicing methods. In section 5, we will conclude our discussions with a few remarks.

2 Program Slice

In this section, we briefly show static slicing, dynamic slicing and dependence-cache(DC) slicing for further discussions.

2.1 Static Slice

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)* relation, from statement s_1 to statement s_2, exists:

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

This relation is denoted by $s_1 \dashrightarrow s_2$.

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

- s_1 defines v, and
- s_2 refers to v (we say s uses v), and
- at least one execution path from s_1 to s_2 without redefining v exists.

This relation is denoted by $s_1 \xrightarrow{v} s_2$.

A *Program Dependence Graph (PDG)*[16] is a directed graph whose nodes represent statements such as conditional predicates or assignment statements in a source program, and whose edges denote dependence relations (CD or DD) between statements. A DD edge is labeled with a variable name "a" if it denotes $s_x \xrightarrow{a} s_y$. An edge drawn from node V_s to node V_t represents that "node V_t depends on node V_s". For the Java source program shown in Figure 1 (which computes an absolute value of the squared or cubed value selected by an input), we have a PDG presented in Figure 2. To handle constructor/method invocations, we employed additional nodes for the input and output parameters.

A *Static Slice* with respect to a variable v on a statement s (this pair (v, s) is called a *slicing criterion*) is a collection of statements corresponding to the nodes in the PDG, which possibly reach s using CD and DD edges transitively.

The static slice for variable d at line 24 as the slicing criterion for the program is a collection of all statements except for the message output statements (lines 12, 14, 16) shown in Figure 3.

```
 1  class sample {
 2    static int Square(int x) {
 3      return x*x;
 4    }
 5    static int Cube(int x) {
 6      return x*x*x;
 7    }
 8    public static void main(String args[]) {
 9      int a, b, c, d;
10      BufferedReader br = new BufferedReader
          (new InputStreamReader(System.in));
11      try {
12        System.out.println("Squared Value ?");
13        a = Integer.valueOf(br.readLine()).
              intValue();
14        System.out.println("Cubed Value ?");
15        b = Integer.valueOf(br.readLine()).
              intValue();
16        System.out.println
            ("Select Feature! Square:0 Cube:1");
17        c = Integer.valueOf(br.readLine()).
              intValue();
18        if (c == 0)
19          d = Square(a);
20        else
21          d = Cube(b);
22        if (d < 0)
23          d = -1 * d;
24        System.out.println(d);
25      }
26      catch (IOException e) {
27        System.out.println("IO Error occuerd!");
28      }
29    }
30  }
```

Figure 1. A Sample Source Program

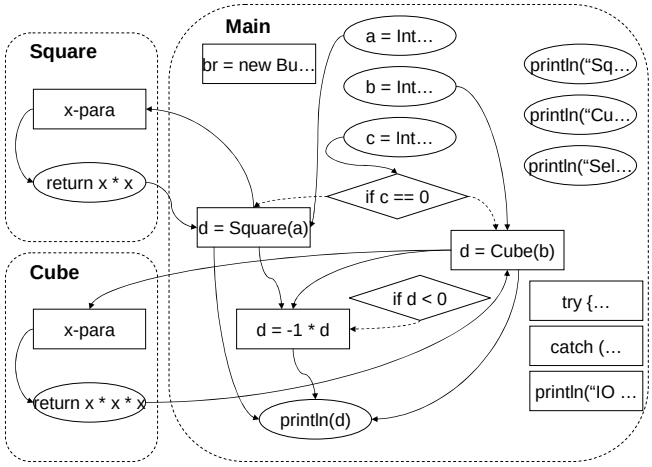

Figure 2. Program Dependence Graph (PDG) of the Program shown in Figure 1.

```
 1  class sample {
 2    static int Square(int x) {
 3      return x*x;
 4    }
 5    static int Cube(int x) {
 6      return x*x*x;
 7    }
 8    public static void main(String args[]) {
 9      int a, b, c, d;
10      BufferedReader br = new BufferedReader
          (new InputStreamReader(System.in));
11      try {
12
13        a = Integer.valueOf(br.readLine()).
              intValue();
14
15        b = Integer.valueOf(br.readLine()).
              intValue();
16
17        c = Integer.valueOf(br.readLine()).
              intValue();
18        if (c == 0)
19          d = Square(a);
20        else
21          d = Cube(b);
22        if (d < 0)
23          d = -1 * d;
24        System.out.println(d);
25      }
26      catch (IOException e) {
27        System.out.println("IO Error occuerd!");
28      }
29    }
30  }
```

Figure 3. Static Slicing Result by d at Line 24

2.2 Dynamic Slice

For dynamic slicing, the analysis target is an *execution trace* in contrast to a source program for static slicing. An execution trace is a sequence of statements that are actually executed for an input data. The rth-executed statement in an execution trace is called execution point r.

When all of the following conditions are satisfied, we say that a *dynamic control dependence (DCD)* relation, from execution point r_1 to execution point r_2, exists:

- r_1 is a conditional predicate, and
- the result of r_1 determines whether r_2 is executed or not.

When the following conditions are all satisfied, we say that a *dynamic data dependence (DDD)* relation, from execution point r_1 to execution point r_2 by a variable v, exists:

- r_1 defines v, and
- r_2 uses v, and
- the execution path after r_1 dose not before r_2 re-define v.

A *dynamic dependence graph (DDG)* is created using the dynamic dependence relations: DCD and DDD.

Consider an execution point r and variable v in an execution trace e whose input data set is $\mathcal{X}$. A triple ($\mathcal{X}$, r, v) is called the *dynamic slicing criterion*.

A *dynamic slice* for the dynamic slicing criterion ($\mathcal{X}$, r, v) is computed by tracing DDG's edges backward from the node for r, and by mapping all reachable nodes into the source program.

Figure 4 shows a dynamic slice of the program shown in Figure 1. The dynamic slicing criterion is an input data set $(a = 2, b = 3, c = 0)$, an execution point corresponding to line 24, and variable d.

Dynamic slicing is based on a single execution path, and it can give narrower slices than static slices. This is preferable in debugging situation, since it is easier for debugger to focus his/her attention on the smaller slices.

```
 1  class sample {
 2    static int Square(int x) {
 3      return x*x;
 4    }
 5
 6
 7
 8    public static void main(String args[]) {
 9      int a, b, c, d;
10      BufferedReader br = new BufferedReader
          (new InputStreamReader(System.in));
11
12
13        a = Integer.valueOf(br.readLine()).
              intValue();
14
15
16
17        c = Integer.valueOf(br.readLine()).
              intValue();
18        if (c == 0) {
19          d = Square(a);
20
21
22
23
24        System.out.println(d);
25
26
27
28
29    }
30  }
```

Figure 4. Dynamic Slicing Result by d at Line 24 with input $(a = 2, b = 3, c = 0)$

2.3 DC(Dependence-Cache) Slicing

When we statically analyze a source program that has 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. Also, in the case of analyzing a source program that has 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[8, 17, 18]; however, it is difficult for static analysis to generate practical analysis results of the pointer alias problem.

On the other hand, dynamic slicing is based on a single execution path, and it can give narrower slices than static slices. However, the dynamic slicing requires a large amount of execution time, and the execution trace grows to an enormous size.

We have proposed a technique called DC slicing which combined both the static and dynamic slicing [4, 15, 19].

Consider a variable v on a statement s and input variables set at execution is $\mathcal{X}$. The triple ($\mathcal{X}$, s, v) is called a *DC slicing criterion*.

[**DC Slice Computation Process**]

Computation process of a DC slice is as follows.

Phase 1: Static Control Dependence Analysis

We extract CD relations between statements statically. After the extraction, we prepare nodes for each statement or predicate statement, and then draw control dependence edges between nodes, as we do when constructing a PDG for the static slicing. No data dependence edges are added to the graph at this step.

Phase 2: Dynamic Data Dependence Analysis

The target program is executed with an input data set $\mathcal{X}$. Along the execution, dynamic data dependence relations are collected using the *data dependence collection algorithm* shown below, and data dependence edges are added to the graph. When the program execution terminates, the PDG has been completed.

Phase 3: Slice Extraction

We extract the node which can reach the DC slicing criterion by following transitive CD and DDD edges.

[**Dynamic Data Dependence Analysis for DC slicing**]

In DC slicing, a DDD relation is extracted at the time of execution of a target program like a dynamic slicing. DDD relation of variable v from statement s to statement t can be extracted when v defined at s is used at 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 to, we extract DDD relation of v using the cache table. The following shows the extraction algorithm for DDD relations for DC slicing.

Step 1: We create cache $C(v)$ for each variable v in the 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.
For the execution of s

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

Since DC slicing compute the DDD relation along with the program execution, array indices and indirect references by the pointers can be resolved easily. DDD relations obtained by DC slicing include surplus ones compared with the dynamic slicing, since DC slicing cannot distinguish instances of the execution of each statement. However, this drawback is not so significant, as shown in the next section.

2.4 Comparison with Static, Dynamic and DC Slices

Table 1 shows the difference among static slice, dynamic slice and DC slice.

Table 1. Comparison of analysis method among static, dynamic and DC slicing

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

Figure 5 depicts static, dynamic and DC slices for slicing criterion <28, d>. For dynamic and DC slicing, we give integer "2" to **readLine**() statement at line 20. "$\surd$" represents a statement in the slices, and "S", "D" and "DC" represents static, dynamic and DC slices, respectively. In this case, dynamic slicing and DC slicing compute the same slices.

Nature of each slicing method is summarized as follows.

analysis accuracy (slice size)

Static Slice $\geq$ DC Slice $\geq$ Dynamic Slice

S	D	DC		
√	√	√	1:	class sample2 {
√	√	√	2:	static int SIZE = 5;
√	√	√	3:	static int Cube(int x) {
√	√	√	4:	return x*x*x;
√	√	√	5:	}
√	√	√	6:	public static void main(String args[]) {
√	√	√	7:	int a[] = new int[SIZE];
√	√	√	8:	int b[] = new int[SIZE];
√	√	√	9:	int c = 0, d, i;
√	√	√	10:	BufferedReader br = new BufferedReader
√	√	√		(new InputStreamReader(System.in));
√			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:	try {
√	√	√	20:	c = Integer.valueOf(br.readLine()).intValue();
√	√	√	21:	}
√			22:	catch (IOException e) {
√			23:	System.out.println(”IO Error occuerd!”);
√	√	√	24:	}
√	√	√	25:	d = Cube(b[c]);
√	√	√	26:	if (d < 0)
√			27:	d = −1 * d;
√	√	√	28:	System.out.println(d);
√	√	√	29:	}
√	√	√	30:	}

Figure 5. Static, Dynamic and DC slices for slicing criterion <28, d> and Input c=2

analysis cost (dependence relations analysis time and space)

$$\text{Dynamic Slice} \gg \text{DC Slice} > \text{Static Slice}$$

3 Implementation of DC Slicing for Java

In order to show the usefulness of DC slicing method for object-oriented languages such as Java, we have studied an implementation methods. In [15], we proposed a method of implanting analysis codes to the target source code before compilation. However, it has some drawbacks; it does not work properly for complex statements with nested method calls. Therefore, its applicability is limited.

In this paper, we propose an implementation method of DC slicing system, by extending Java Virtual Machine which processes bytecodes, so that the virtual machine can extract dynamic data dependencies during execution. Since the analysis target is bytecode, we will define a slice calculation method on bytecode while the conventional slice calculation method is defined on source code. User cannot understand the slice on bytecode. Thus, a slice of the bytecode is mapped back onto a slice of the source code by referring to the cross reference table which is created by Java compiler.

DC slices are extracted as follows.

Phase 0: Cross Reference Table between source code and bytecode is created by the Java compiler

Phase 1: Static control dependence is analyzed and a PDG without DDD edges is constructed

Phase 2: Dynamic data dependence is analyzed during the program execution by JVM and the PDG is completed

Phase 3: Slices on bytecode is extracted and they are mapped back to the source code

3.1 Cross Reference Table between Source Code and Bytecode

In the proposal method, we create a cross reference table, in order to get a mapping between bytecode and source code.

We have extended the Java compiler so that the cross reference table can be obtained. When building a syntactic-analysis tree from the source code, the Java compiler holds the information of each token in the source code. When the Java compiler generates the bytecode from the syntactic-analysis, the correspondance relation as is extracted.

In general, the Java compiler optimizes the bytecode. However, the correspondence relation between the source code and the bytecode is lost by the optimization. Thus, we turn off the the optimaization here.

We show an example of the cross reference table in Figure 6. By referring to this cross reference table, we can point the slice criterion specified on the source code into the slice criterion on the bytecode, and calculate the slice on the bytecode.

3.2 Control Dependence Analysis

In the DC slicing method, control-dependence analysis is made statically. Here we define control dependence relation on the bytecode as follows[3]. This control dependence relations are computed by applying the algorithm on Figure 7 to each method in the bytecode.

Defintion Control Dependence

Consider two bytecode statements s and t. When s and t satisfy the following conditions, we say that a control dependence relation exists from s to t.

1. s is a branch command, and the last command of a *basic block*[2] X.

```
int i = 2;
if (i > 2) {
    i++;
} else {
    i--;
}
int j = i;
```

bytecode statements		a corresponding token set
	iconst_2	"2"
	istore_1	"i ="
	iload_1	"i"
	iconst_2	"2"
	if_icmple L13	">"
	iinc 1 1	"i++"
	goto L18	"if"
L13:	nop	""
	nop	"if"
	iinc 1 -1	"i−−"
L18:	nop	"if"
	iload_1	"i"
	istore_2	"j ="
	return	"main"

Figure 6. Cross Reference Tables

2. Assume that X branches to basic blocks U and V, and consider an execution path p from U to the exit and q from V to the exit. t satisfies the following.
   (a) Any p includes t
   (b) No q includes t.

3.3 Data Dependence Analysis for DC Slicing

In the DC slicing method, the target program in bytecode is executed on an extended JVM(Java Virtual Machine), and the data dependence relation is extracted at the execution time. We prepare a cache area for each data field to identify the bytecode statement which defines the latest value of the data field. Examples of the data field are member variables in each instance, stack elements on JVM, and local variables in each method.

When a data field d is referred to at execution of bytecode on JVM, we extract a DDD relation for d using the cache of d. A DDD relation is obtained from the statement specified by the cache of d to the statement which made this reference. When a value of a data field d is defined, the cache of d is updated by the statement which made the definition. The cache for a dynamically allocated data field is created at the same time when the data field is created.

Input Bytecode
Output Control dependence relations between bytecode statements
Process Compute static control dependence relations for bytecode

(1) Divide bytecode into Basic Block, and construct its control flow graph G
(2) Add an entry node R, an exit node E, and their associated edges to G, and add each edge from$R$ to first node in G, last node in G to E, from R to E
(3) Construct reverse control flow graph G' for G($\mathcal{N}$: set of nodes in G')
(4) Construct dominator tree[10, 14] for G'(the root is E)
(5) **foreach** basic block x **in** $\mathcal{N}$ **begin**
(6) find Dominance Frontier[7][a] $DF\_G'[x]$
(7) **foreach** y **in** $DF\_G'[x]$
(8) Compute and output pairs of last statement in y and each statement in x regarded as control dependence relations
(9) **end**

[a]The **Dominance Frontier** of a node s is the set of all nodes t such that s dominates a predecessor of t, but does not strictly dominate t.

Figure 7. Static Control Dependence Analysis Algorithm

Figure 8 shows the dynamic data dependence analysis algorithm. In this algorithm, each instance generated from the same class has independent cache, so that we can analyze the DDD relation of each instance independently. Table 2 shows a transition of caches and DDD relations during the execution of bytecode shown in Figure 9.

Input Bytecode
Output Data dependence relations of statements s
Process Compute dynamic data dependence relations for execution of each statement s

(1) **foreach** feild data n referred to at s
(2) output the pair of the statement specified by the cache for n and s
(3) **foreach** field data n defined at s **begin**
(4) **if** no cache for n exists **then**
(5) generate a cache for n
(6) update the content of cache for n to s
(7) **end**

Figure 8. Dynamic Data Dependence(DDD) Analysis Algorithm

A program dependence graph(PDG) is constructed by using the control dependences and the dynamic data depen-

Table 2. Transitions of the caches for figure9

statement	local variable[0]	stack[0]	stack[1]	dependence relations
1	-	1	-	
2	2	-	-	$1 \rightarrow 2$
3	2	3	-	
4	2	3	4	$2 \rightarrow 4$
5	2	-	-	$3{,}4 \rightarrow 5$
6	6	-	-	$2 \rightarrow 6$
7	6	-	-	
9	6	9	-	$6 \rightarrow 9$
10	6	-	-	$9 \rightarrow 10$

dences. Figure 9 shows an example of PDG. In this method, each node represents a bytecode statement, each edge represents a dependency relation. To compute DC slices, we need only dependence relations of statements. On the other hand, dynamic slices require huge execution trajectory[20]. Therefore the analysis cost of DC slices is much smaller than that of dynamic slices.

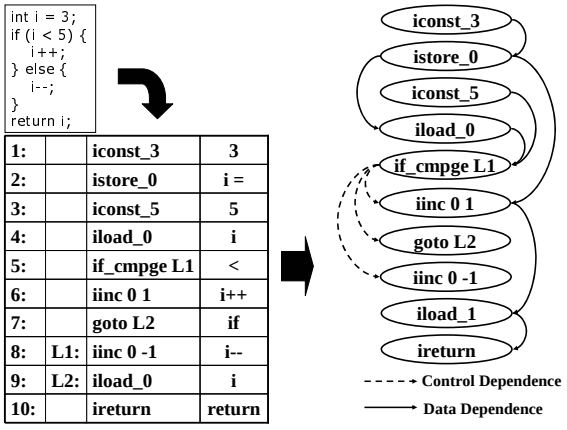

Figure 9. Example of Program Dependence Graph for Bytecode

3.4 Computation of a Slice

After constructing the PDG, we compose a slice from given a DC slicing criterion. The method is essentially the same as usual ones. We collect a set of reachable nodes through edges reversely from the node corresponding to the DC slicing criterion.

3.5 System Architecture

In this section, we show a DC slicing system for Java programs we have developed. Figure 10 shows its architecture. Figure 11 is a screenshot of the main window of the system.

Java compiler produces the cross reference table between the source code and bytecode. Control dependence relations is statically analyzed by CD analysis tool, and data dependence relations is dynamically analyzed by the extended JVM while execution. Based on the dependence relations extracted by these processes, a PDG of the bytecode is constructed. A DC slice criterion on the source program is specified by the user, and it is translated to the bytecode statement with the cross reference table. The reachable statements are collected by traversing the PDG. Finally, the slice result is mapped back on the source program by the cross reference table.

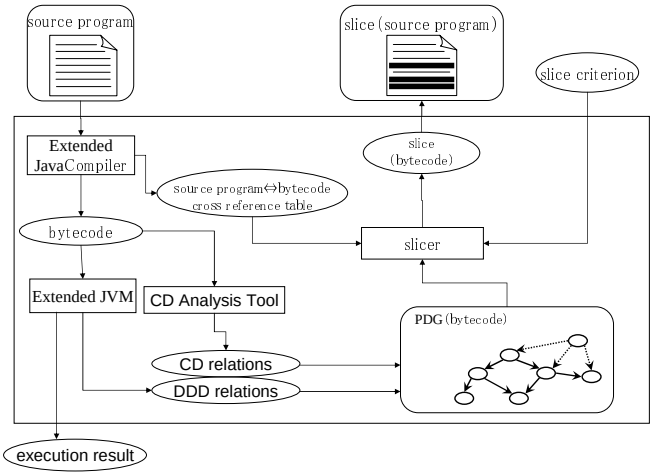

Figure 10. System Architecture

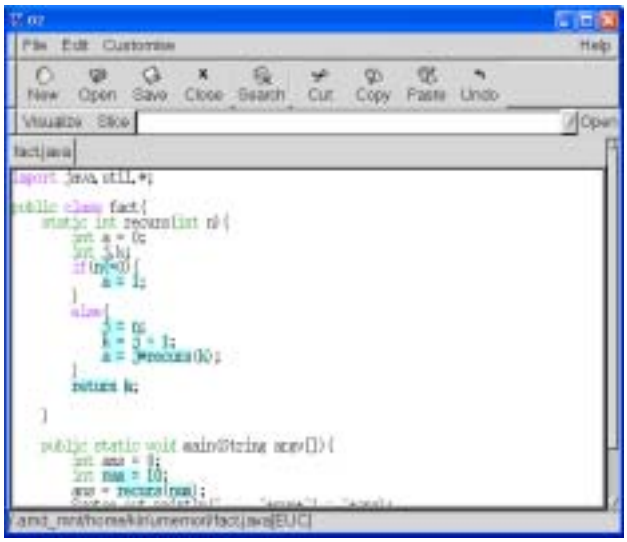

Figure 11. System Main Window

Table 5. Analysis Cost

program	JVM execution time[ms]		JVM memory usage[Kbytes]		PDG construction time and slice calculation time[ms]
	original	extended	original	extended	
P1	325	2,058	3,780	15,980	525
P2	341	3,089	4,178	26,091	450

Table 3. Slice target program

program	classes	total lines
P1 (database management)	4	262
P2 (sorting)	5	231

4 Evaluation

In this section, we evaluate the proposal approach by a comparison with traditional slicing methods. We have made an experiment, and we have evaluated the slice size and analysis cost. Table 3 lists the target programs for the evaluation. Program P1(which consists of 4 classes, 262 statements) is a database management program, and the program P2(which consists of 5 classes, 231 statements) is a sorting program.

We have applied our DC slicing system to P1 and P2, and measured the slice size, used memory, PDG construction time, slice calculation time, and the number of PDG nodes.

4.1 Slice Size

We have measured the slice sizes for the static slicing, dynamic slicing, and DC slicing. Table 4 shows the sizes of slices for two slice criteria. The static and dynamic slices were counted by hand.

From the viewpoint of fault localization, we prefer smaller slice sizes. DC slice sizes are smaller than static slice sizes. In this experiment, the DC slice sizes are almost equivalent to the dynamic slice sizes.

The DC slice sizes are about 50% to 93% of the static slice sizes, and DC slices provide a better focus to falut locations. Since the target programs used here are small-scale ones, the difference between static slices and DC slices is not so large. However, if class inheritances, overrides and overloads of methods are used in a large-scale program, we would guess that the difference becomes larger. This is because static slicing has to consider all possible cases, but our approach considers actually used inheritances, overrides, and overloads.

Table 4. Slice Size [Lines]

	Static Slice	Dynamic Slice	DC Slice
P1-slice criterion 1	60	24	30
P1-slice criterion 2	19	14	15
P2-slice criterion 1	79	51	51
P2-slice criterion 2	27	23	25

4.2 Analysis Cost

We have compared the extended JVM with the original JVM with respect to the execution time and the memory usage. The target programs are listed ones in Table 3. Table 5 shows the results.

As you can see from these tables, the extended JVM requires more execution time and space. The extended JVM is 6-9 times slower and 4-6 times space consuming than the original JVM. One reason for this is that the DDD analysis is performed not only for the traget program but for associated JDK libraries. Moreover, the extended JVM executes bytecodes without any optimization, but the origined one performs JIT(Just In Time) optimization.

Table 6 shows the number of nodes in PDG created by the DC slicing and dynamic slicing. Dynamic slicing required 30-50 times more nodes, which drastically increase the memory usage at the execution. Compared to the dynamic slicing, DC slicing is less costly and more practical approach to get reasonable slices.

We are planning to improve analysis speed with less memory, although, current system is considered practical enough.

Table 6. PDG nodes

program	DC slicing	dynamic slicing	ratio
P1	34,966	1,198,596	1 : 34.3
P2	34,956	1,808,051	1 : 51.7

5 Summary

In this paper, we have proposed an implementation method of the DC slicing for Java program.

The major characteristics of this method is that the analysis of control dependences and data dependences are per-

formed at the bytecode level, and the slice results are mapped back to the source program.

The proposed method has been actually implemented by extended JVM to collect the dynamic data dependences. To validate this approach, we have applied the developed system to sample programs. The result shows that our approach to implement DC slicing for Java is very practical and realistic one to get effective slices.

As a future work, we are planning to improve JVM further for more efficient dynamic data dependence analysis.

References

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

[2] A. V. Aho, R.Sethi and J. D. Ullman: "Compilers Principles, Techniques, and Tools", Addison-Wesley, (1986).

[3] A. W. Appel and M. Ginsburg: "Modern Compiler Implementation in C", Cambridge University Press, Cambridge (1998).

[4] Y. Ashida, F. Ohata and K. Inoue: "Slicing Methods Using Static and Dynamic Information", Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC'99), pp.344–350, Takamatsu, Japan, December (1999).

[5] D. Atkinson and W. Griswold: "The design of whole-program analysis tools", Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, pp.16-27, (1996).

[6] D. Binkley, S. Horwitz, and T. Reps: "Program integration for languages with procedure calls", ACM Transactions on Software Engineering and Methodology 4(1), pp.3-35, (1995).

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck: "Efficiently Computing Static Single Assignment Form and the Control Dependench Graph", ACM Transactions on Programming Languages and Systems, Vol.13, No.4, pp.461-486, October (1991).

[8] M. Enami, R. Ghiya and L. J. Hendren: "Contextsensitive interprocedural points-to analysis in the presence of function pointers", Proceedings of the ACM SIGPLAN94 Conference on Programming Language Design and Implementation, pp.242-256, Orlando, Florida, June (1994).

[9] K.B.Gallagher: "Using Program Slicing in Software maintenance", IEEE Transactions on Software Engineering, 17(8), pp.751-761 (1991).

[10] D. Harel: "A liner time algorithm for finding dominator in flow graphs and related problems", Proceedings of 17th ACM Symposium on Theory of computing, pp.185–194, May (1985).

[11] M. Harman and S. Danicic: "Using program slicing to simplify testing", Journal of Software Testing, Verification and Reliability, 5(3), pp.143-162 (1995).

[12] S. Kusumoto, A. Nishimatsu, K. Nishie, K. Inoue: "Experimental Evaluation of Program Slicing for Fault Localization", Empirical Software Engineering (An International Journal), Vol.7, No.1, pp.49-76, March (2002).

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

[14] T. Lengauer and E. Tarjan: "A fast algorithm for finding dominators in a flow graph", ACM Transactions on Programming Languages and Systems, Vol.1, No.1, pp.121-141, July (1979).

[15] F. Ohata, K. Hirose and K. Inoue: "A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information", Proceedings of Eighth Asia-Pacific Software Engineering Conference (APSEC2001), pp.273–280, Macau, China, December (2001).

[16] K. J. Ottenstein and L. M. Ottenstein: "The program dependence graph in a software development environment", Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp.177–184, Pittsburgh, Pennsylvania, April (1984).

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

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

[19] T. Takada, F. Ohata and K. Inoue: "Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information", Proceedings of the 10th International Workshop on Program Comprehension(IWPC2002), pp.169-177, Paris, France, June (2002).

[20] M. Weiser: "Program Slicing", IEEE Transaction on Software Engineering, 10(4), pp.352–357 (1984).

[21] J. Zhao: "Dynamic Slicing of Object-Oriented Programs", Technical Report SE-98-119, Information

Processing Society of Japan (IPSJ), pp.11-23, May (1998).