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 programs 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[6, 7, 10, 12, 21]. We have empirically investigated effectiveness of program slicing for program debugging and program maintenance processes, and its significance was validated by several experiments[13].

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[14, 21] and dynamic slicing[2, 22]. 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 static or dynamic information, and have proposed DC slicing technique which uses both static and dynamic information. Using DC slicing, 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 the bytecode, we define the slice calculation method on the 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 the DC 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 Dependence-Cache(DC) Slicing

In this section, we 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: Control Dependence Analysis and Data Dependence Analysis
We extract control dependence relations and data dependence relations between program statements.

Phase 2: Program Dependence Graph[17] Construction
We construct *Program Dependence Graph (PDG)* using dependence relations extracted on Phase 2.

Phase 3: Slice Extraction
We compute the slice for the slicing criterion specified by the user. In order to compute the slice for a slicing criterion $<s, v>$ (where s is a statement and v is a variable), PDG nodes are traversed in reverse order from V_s (V_s is a node corresponding to s.). The statements corresponding to the visited 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*).

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 Java program and its PDG (**Phase 1, 2**), and Figure 2 shows the slice ("$*$"-marked statements) for <6, c> on Figure 1 (**Phase 3**).

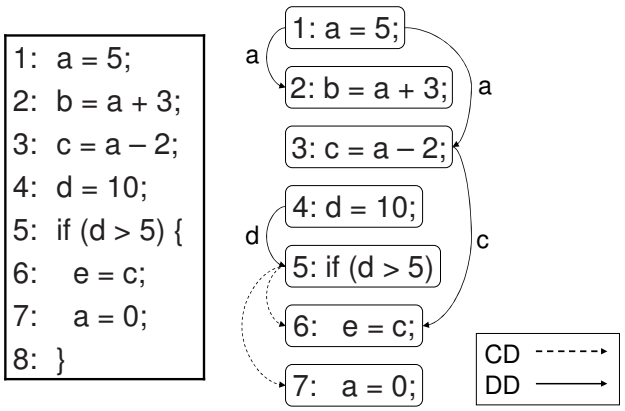
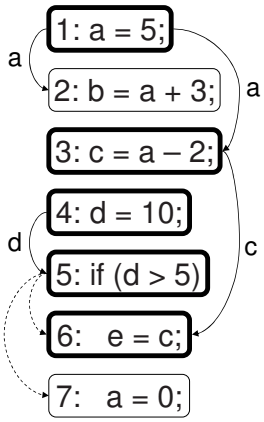


Figure 1. Sample Java Program and its Program Dependence Graph (PDG)

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[9, 19, 18]; however, it is difficult for static analyses to generate practical analysis results — *pointer alias problem*.




```
* 1:  a = 5;
  2:  b = a + 3;
* 3:  c = a – 2;
* 4:  d = 10;
* 5:  if (d > 5) {
* 6:    e = c;
  7:    a = 0;
  8:  }
```

Figure 2. Slice for <6, e> on Figure 1

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

Phase 2: 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 3: Slice Extraction

[**Dynamic Data Dependence Analysis**]

When variable v is referred to 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[1]. When variable v is referred to, 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.
At the execution of statement s,

- when variable v is referred to, 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.

2.3 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 4 shows PDGs constructed from a sample program on Figure 3 by Static, Dynamic and DC slicing technique; for dynamic and DC slicing, we passed integer value "2" to readLine() statement on program execution. PDG_S, PDG_D and PDG_{DC} represent the PDG for Static Slicing, Dynamic Slicing, DC Slicing, respectively.

Table 2. The number of nodes and dependence edges for each PDG on Figure 4

	PDG_S	PDG_D	PDG_{DC}
Node	10	13(8 statements)	10
Edge	22	20	15

The number of nodes in PDG_S or PDG_{DC} is equal to the number of program statements. On the other hand the number of nodes in PDG_D is the number of execution points. Since any node corresponding to unexecuted statement does not exist on PDG_D, the number of program statements executed is less than that of PDG_S and PDG_{DC}. However, if a certain statement in a loop block is executed repeatedly, the number of nodes in PDG_D will increase easily.

```
1:   a[0] = 1;
2:   a[1] = 2;
3:   a[2] = 3;
4:   b = Integer.parseInt(br.readLine());
5:   while (b > 0) {
6:     if (b < 10) {
7:       c = a[b];
8:     } else {
9:       c = a[b] – 10;
10:    }
11:    b = b – 1;
12:  }
13:  System.out.println(c);
```

Figure 3. Sample Program

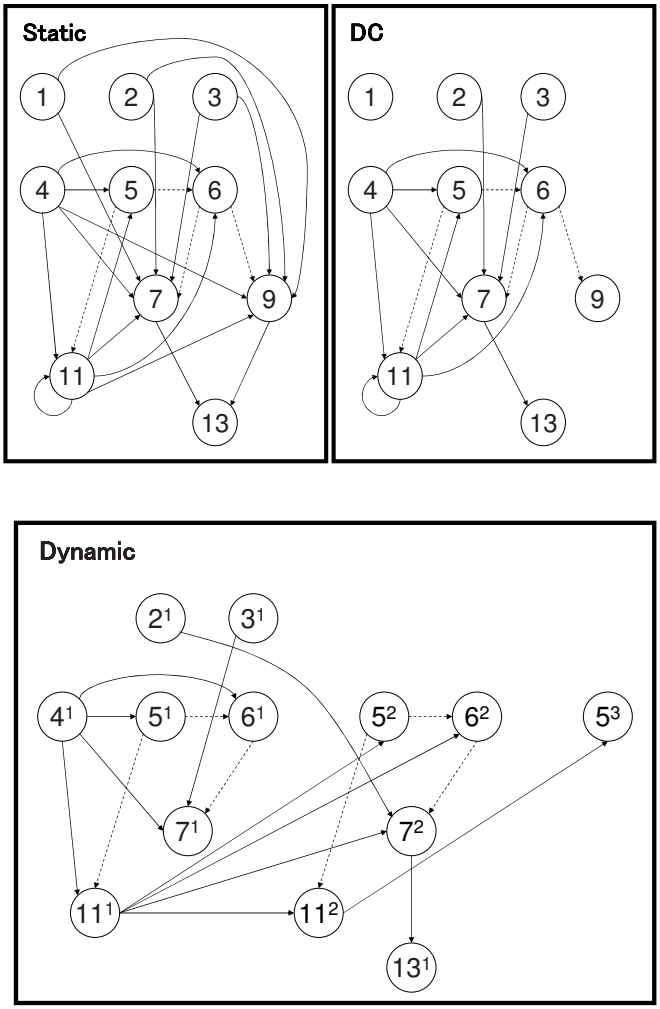


Figure 4. Program Dependence Graphs (PDG) of the Program shown in Figure 3.

PDG_S generally has more edges than others because of the consideration of possible execution path. And only the edges on execution path are extracted in dynamic slicing, so the number of edges in PDG_D can increase depending on the length of the execution path. We can see a merged dependence edge in PDG_{DC} for several distinct in PDG_D. For example, the two edges represented $11^1 \longrightarrow 5^2$, $11^2 \longrightarrow 5^3$ in PDG_D, are represented one edge $11 \longrightarrow 5$ in PDG_{DC}. The analysis cost of PDG_{DC} is less than PDG_D. The edges of PDG_{DC} are based on actual execution, so they are fairly accurate.

In DC slicing, PDG has redundant control dependence edge. So the size of DC slice may become large than that of dynamic slicing. However, edges of PDG are traversed from destination node to source node. If the node after branch of a condition predicate cannot be reached during slice extraction, a redundant statement is not added to a slice. So the existence of redundant edge does not make a big difference of the size of Dynamic Slise and DC Slice.

Since the data dependence edges in PDG_{DC} cannnot hold the information of multiple occurrence of a single statement as PDG_D, the size of DC slice may become larger than that of dynamic slice. However, the size of the execution trace is much larger than that of program statements generally. In many cases, the program which can apply static slicing cannnot apply dynamic slicing. With DC slicing, the slice of a near dynamic slice accuracy can be always obtained.

The characteristic of each slicing method is summarized as follows[5, 20].

analysis accuracy (slice size)

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

analysis cost (dependence relations analysis time and space)

Dynamic Slice $\gg$ DC Slice $>$ Static Slice

2.3.1 Example of Slice Extraction

For the program shown Figure 3 with slicing criterion <13, c>, we get the following slices.

Static Slice	$\{1, 2, 3, 4, 5, 6, 7, 9, 11, 13\}$
Dynamic Slice	$\{2, 3, 4, 5, 6, 7, 11, 13\}$
DC Slice	$\{2, 3, 4, 5, 6, 7, 11, 13\}$

For another slicing criterion <9, c>, we get the following.

Static Slice	$\{1, 2, 3, 4, 5, 6, 9, 11\}$
Dynamic Slice	$\{\}$
DC Slice	$\{4, 5, 6, 9\}$

The size of DC slice is smaller than that of static, and it includes redundant statements but is close to dynamic 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 [16], we proposed a method of embedding 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 grasp the slice on bytecode intuitively, so 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.

Our proposal method extracts DC slices 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 DD 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 the 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 between the source code and the bytecode also 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 5. By referring to this cross reference table, we can translate the slice criterion specified on the source code into the slice criterion on the bytecode. Related with the source code after slice calculation, we can grasp the slice on the bytecode.

```
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 5. Cross Reference Tables

3.2 Control Dependence Analysis

In the DC slicing method, control-dependence analysis is done statically. Here we define control dependence relation on the bytecode as follows[4]. This control dependence relations are computed by applying the algorithm on Figure 6 to each method in the bytecode.

Definition 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*[3] X.
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.

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[11, 15] for G'(the root is E)
(5) **foreach** basic block x **in** $\mathcal{N}$ **begin**
(6) find Dominance Frontier[8][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 6. Static Control Dependence Analysis Algorithm

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 DD relation for d using the cache of d. A DD 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.

Figure 7 shows the dynamic data dependence analysis algorithm. In this algorithm, each instance generated from the same class has independent cache, so that we can extract the DD relation of each instance independently. Table 3 shows a transition of caches and DD relations during the execution of bytecode shown in Figure 8.

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

(1) **foreach** field 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 7. Data Dependence(DD) Analysis Algorithm

Table 3. Transitions of the caches for figure8

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	-	-	$9 \rightarrow 10$

PDG_{DC} for the bytecode is constructed as shown in Figure 8. In this method, each node represents a bytecode statement, each edge represents a dependency relation.

3.4 Computation of a Slice

After constructing the PDG, we compose a slice from a given 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.

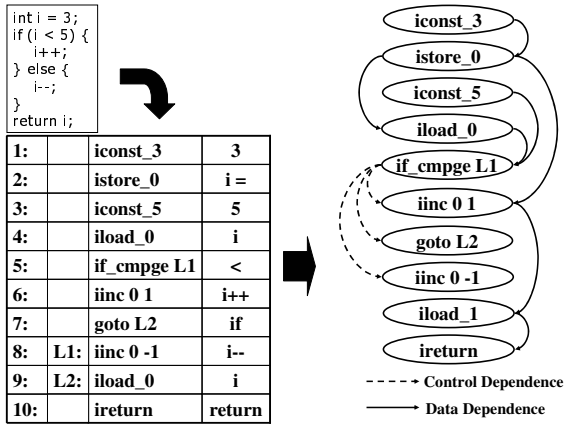


Figure 8. Example of Program Dependence Graph for Bytecode

3.5 System Architecture

In order to realize our proposal method, we hava developed a DC slicing system for Java programs. Figure 9 shows its architecture. Figure 10 is a screenshot of the main window of the system.

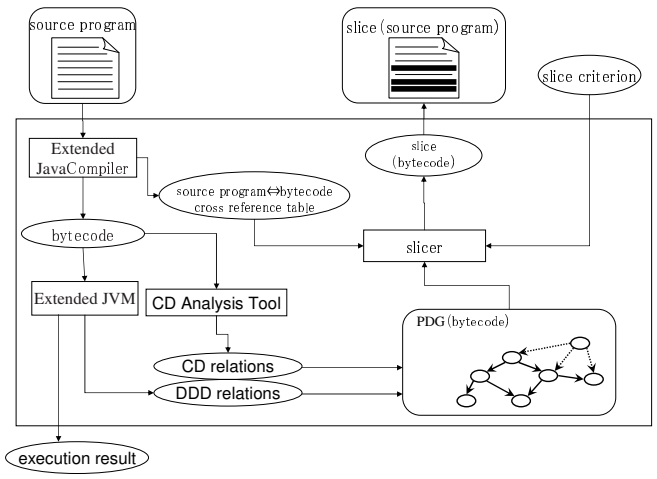


Figure 9. System Architecture

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.

Table 4. Slice target program

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

Table 5. 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 Evaluation

In this section, we evaluate our approach by a comparison with traditional slicing methods. We have made an experiment, and we have evaluated the slice size and analysis cost. Table 4 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 5 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

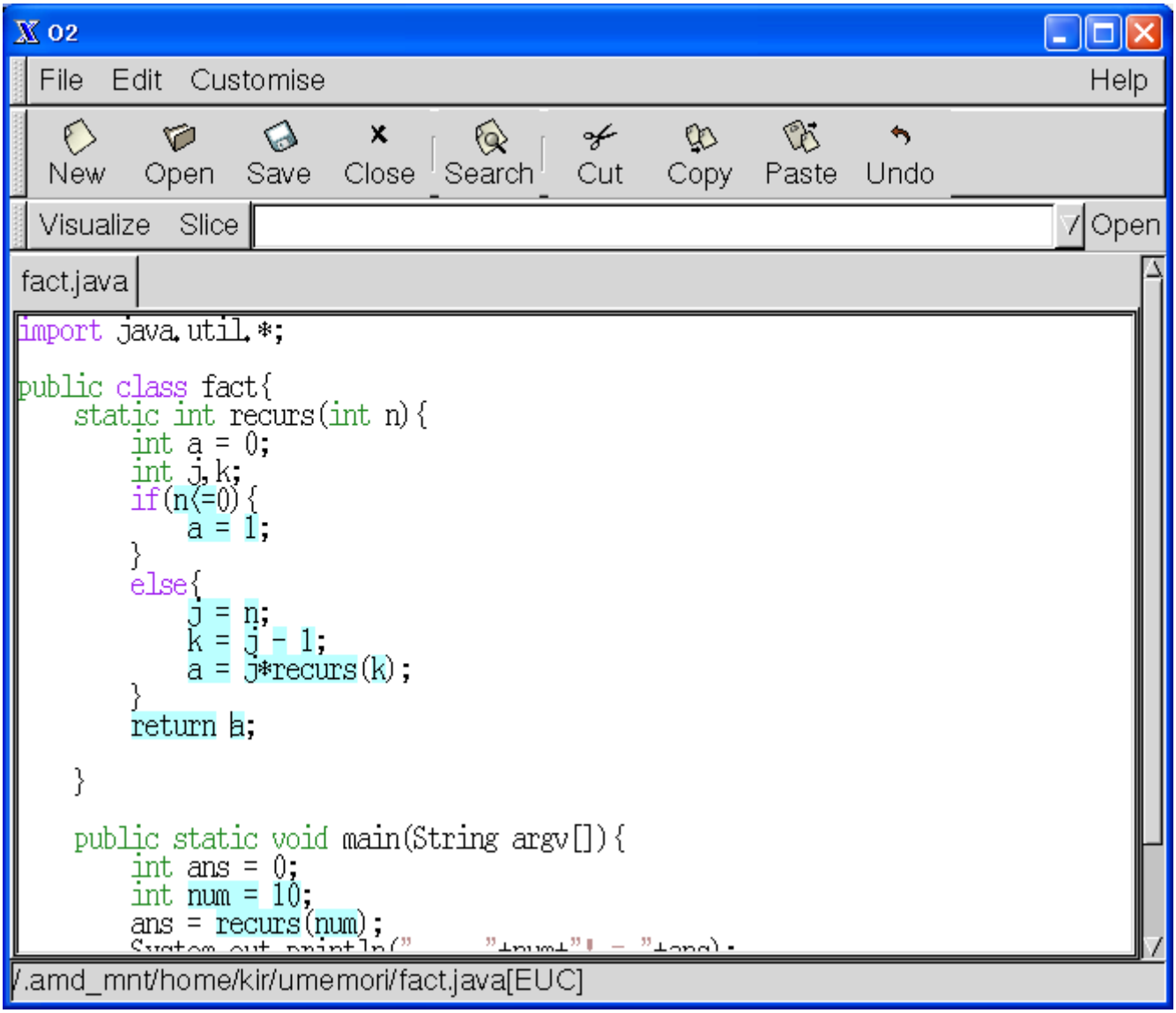


Figure 10. System Main Window

Table 6. 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

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.

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 4. Table

6 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 more space consuming than the original JVM. One reason for this is that the DD 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 7 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 in debugging environment.

Table 7. 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 Conclusion

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 performed 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, R. A. DeMillo and E. H. Spafford: “An execution backtracking approach to program debugging”, IEEE Software, pages 21-26, May 1991.

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

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

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

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

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

[11] 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).

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

[13] 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).

[14] 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).

[15] 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).

[16] 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).

[17] 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).

[18] 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).

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

[20] 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).

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

[22] J. Zhao: “Dynamic Slicing of Object-Oriented Programs”, Technical Report SE-98-119, Information Processing Society of Japan (IPSJ), pp.11-23, May (1998).