A Novel Approach for Computing Dynamic Slices of Object-Oriented Programs with Conditional Statements

Durga Prasad Mohapatra, Rajib Mall and Rajeev Kumar

Abstract— **We propose a dynamic program slicing technique for object-oriented programs. We introduce the notion of compact dynamic dependence graph (CDDG) which is used as the intermediate program representation. Our dynamic slicing algorithm is based on the CDDG. We show that our algorithm is more time and space efficient than the existing ones. The worst case space complexity of our algorithm is $O(n)$, where n is the number of statements of the program.**

I. Introduction

The concept of a program slice was introduced by Weiser [8]. A static backward program slice consists of those parts of a program that affect the value of a variable v selected at some program statement s. The pair $< s, v >$ is referred to as a slicing criterion.

The program slices introduced by Wesier [8] are called static slices because they are computed as the solution to a static analysis problem that is computed independent of the program input. A static slice accounts for all possible input values. Therefore conservative assumptions are made, which often lead to relatively large slices. To overcome this difficulty, Korel and Laski introduced the concept of dynamic program slicing. A dynamic program slice contains only those statements that actually affect the value of v at a program statement s for a given input.

Slicing object-oriented programming languages presents new challenges which are not encountered in traditional program slicing. To slice an object-oriented program, features such as classes, dynamic binding, encapsulation, inheritance and polymorphism need to be considered carefully. Larson and Harrold were the first to consider these aspects in their work [4]. They extended the system dependence graphs (SDG) to represent object-oriented programs. After the SDG is constructed, the two phase algorithm of Horwitz et al. [3] is used with minor modifications for computing slices. Larson and Harrold have reported only a static slicing technique for object-oriented programs [4], and did not address dynamic slicing aspects.

Although several approaches about slicing object-oriented programs have been developed there are several areas to be improved. The intermediate representation can be constructed in a more compact manner. So, the space requirement will be reduced substantially. One of the major goal of any dynamic slicing technique is efficiency since the results are normally used during interactive applications such as program debugging. The response time of an inefficient dynamic slicer would be unacceptably large for OOPs. With this motivation, in this paper we propose a new dynamic slicing algorithm that is more time and space efficient than the existing dynamic slicing algorithms [9], [7]. In this paper, handling of conditional statements in computing dynamic slices is of important concern, so many object-oriented features have not been discussed. However, representation of object-oriented features can be incorporated into our technique from the work of Larson and Harrold [4].

The rest of the paper is organized as follows. In section 2, we review the related works. In section 3, we present the proposed dynamic slicing algorithm. In section 4, we compare our algorithm with related algorithms. Section 5 concludes the paper.

II. Related Work

In this section we first briefly review some important work concerning static slicing of object-oriented programs and then discuss how these have been extended subsequently by researchers to handle dynamic slicing of object-oriented programs. Static slicing of object-oriented programs has drawn considerable research interest [4], [5], [2]. Horwitz developed the *system dependence graph* (SDG) as an intermediate program representation and proposed a two-phase graph reachability algorithm on the SDG to compute inter-procedural slice. Larson and Harrold extended the SDG to represent object-oriented programs [4]. Their extended SDG can be used to represent many object-oriented features such as classes and objects, inheritance, polymorphism and dynamic binding. After constructing the SDG, they used a two-phase algorithm to compute the static slice. Liang used a more comprehensive intermediate graph representation by taking the *parameter object* (when an object is used as a parameter) as a tree in which the root of the tree represents the object itself and the leaves represent the data members of the object.

All the work discussed above deal with static slicing of object-oriented programs. But research results on dynamic slicing of object-oriented programs have scarcely been reported in the literature [9], [7]. For procedural programs,

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur - 721302, India. E-mails: {durga, rajib, rkumar}@cse.iitkgp.ernet.in

0-7803-8909-3/04/$20.00 ©2004 IEEE

Agrawal and Horgan were the first to present algorithms for finding dynamic program slices using program dependence graphs. They proposed a dynamic slicing method by marking nodes on a static program dependence graph. The computed slice may not be precise, because some dependencies might not hold during execution. They also proposed a precise slicing method based on the dynamic dependence graph (DDG) [1]. A DDG has a vertex for each occurrence of a statement in the execution history and contains only the executed edges. This removes the imprecision of their previous algorithms. However, the disadvantage of this approach is its space requirement. Since a loop may execute an unbounded number of times, there may be unbounded number of vertex occurrences. To represent object-oriented programs, Zhao extended the DDG of Agrawal and Horgan [1], to *dynamic object-oriented dependence graph* (DODG) [9]. After constructing the DODG, he used a two-phase algorithm to compute dynamic slices of object-oriented programs. The disadvantage of this approach is that the number of nodes in a DODG is equal to the number of executed statements, which may be unbounded for programs having loops.

Song proposed a method to compute forward slice of object-oriented programs using *dynamic object relationship diagram* (DORD) [7]. In this method, the dynamic slices for the variables in each statement is computed immediately after the statement is executed. When the last statement is executed, the dynamic slices of all executed statements are obtained. However, such run time slice computation is essential for only some special statements in the loops. For other statements, this approach incurs unnecessary run time overheads. So, computation of dynamic slices using this technique is unnecessarily expensive.

III. Our Proposed Algorithm

We use a dependence based graph named as *Compact Dynamic Dependence Graph* (CDDG) as the intermediate representation for dynamic slicing of object-oriented programs. Based on the CDDG, we propose a new algorithm to compute dynamic slices of object-oriented programs.

Object-oriented programs containing many conditional statements require special techniques for computing dynamic slices with respect to a variable at a particular point. For applications where the program execution remains unspecified, a *more static* approach is required. However, it is not always helpful to make the criterion *completely* static. Often there is a desire to put some constraints on the program's possible executions. These constraints are the *conditions* of slicing. This slicing technique in which the slicing criterion contains some conditions, is known as *conditioned slicing*. Condition slicing removes the parts of the original program which can not affect the variables at the point of interest, when the conditions are satisfied. This produces a conditioned slice, which preserves the behavior of the original program with respect to the slicing criterion. Since, here the condition could simply be either *true* or a conjunction of equalities which define the input, it is possible for conditioned slicing to subsume both static and dynamic forms of slice.

For example, suppose the programmer wants to understand the behavior of the original program when some condition is satisfied. Here, condition slicing will remove parts of the program which can not affect the slicing criterion, when the condition is met. Another possibility is that the program has some conditions already available in the form of subdomain from partition analysis, from the specification, or from the safety constraints. Also, in this case conditioned slicing can be helpful by focusing attention on those parts of the original program which are relevant to the conditions under consideration. To illustrate, the difference between static, dynamic and conditioned forms of slicing, consider the program fragment given in the first column of Fig. 1. This fragment determines the type of a triangle based upon the three side lengths p, q, and r. The dynamic slice for the variable t at the end the program and for the input which sets p, q, and r to 1 is shown in the second column of the figure. The static slice on the final value of t is the entire program as every line affects the final value of t in some way. Notice that, the dynamic slice very specific. It only applies when the variables are all set to one. This criterion can be implemented by conditioned slicing, using the condition $p = q = r = 1$, but a more general condition would produce the same slice, namely $p = q = r$.

The conditioned slice for the final value of t when the condition is

$p = q \bigvee p = r \bigvee q = r$

is shown in the third column of the figure.

It is observed that while computing the dynamic slice of a program the main problem arises when loops containing conditional statements in a program execute for a unbounded number of times. Different iterations may execute different statement sequences when conditional branching in the loop influence the relevant variables at the point of interest. For programs that are well structured, the number of different paths in the control flow graph that may be taken during different iterations of the loop is bounded by a small number. We take the advantage of this fact in reducing the space and execution time overhead for computing dynamic slices.

Before introducing our method, first, we explain it with the help of an example. Fig. 2 represents the CFG of a program fragment with a loop containing conditionals within the loop body. In the rest of the paper we will abbreviate the CFG of a *loop containing conditionals* as LCC. In Fig. 2, each square block with a label represents a *basic block* of the program. A basic block is a sequence of statements in which flow of control enters at the beginning and leaves at the end without any halt or branching except at the end. *A basic block* is executed either in its entirety or not at all. In Fig. 2, the block K contains a loop predicate. Blocks S, V, and W end with conditionals.

The CFG for a simple loop containing many conditional statements can be divided into several levels $L_1, L_2, \ldots, L_q$. Each level corresponds to an *independent conditional statement*. An independent conditional statement is not nested within other conditional statements. Each level might contain different independent paths, e.g., level L_1 consists of three independent paths: *S-A-W, S-B-W, and S-C-W*. Level L_2 consists of two independent paths. Let N_i be the total number of

if (p == q)
if (p == r)
t = "equilateral";
else t ="isosceles";
else
if (p == r)
t = "isosceles";
else
if (q == r)
t= "isosceles";
else t = "scalene";
	if (p == q)
	if (p == q)
	t = "equilateral";
		if (p ==q)
		if (p == q)
		t = "equilateral";
		else t = "isosceles";
		else
		if (p == r)
		t = "isosceles";
		else
		if (q == r)
		t = "isosceles";

Original	Dynamic Slice for p=q=r=1	Conditioned Slice for

Fig. 1. Comparing Static, Dynamic and Conditioned Slicing

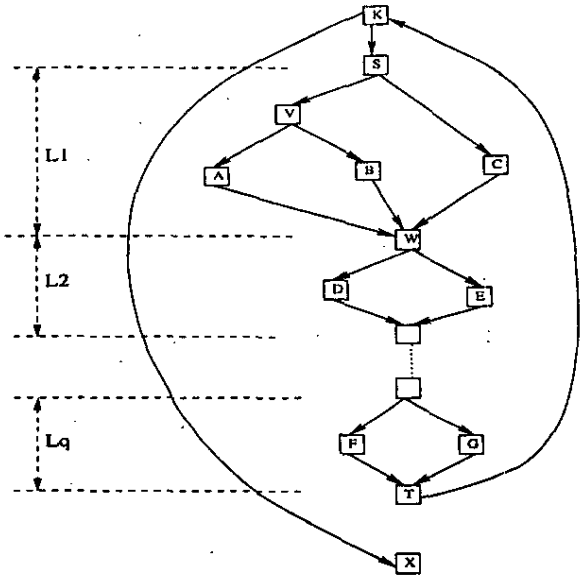

Fig. 2. CFG of a Loop Containing Conditional Statements

independent paths in level L_i. So, the total number of different paths that may be taken in different iterations of the loop is given by the product $N = N_1.N_2 \ldots N_q$, where q is the total number of levels within the loop. It is an accepted software engineering principle to restrict the value of N to below 10 [6]. Even if, this value is large, it is always bounded (this bound obviously equals the number of statements in the basic block) and therefore our method will still work. In our approach, we construct the dynamic dependence graph (DDG) at run-time. However, during execution of an LCC we do not create a new node every time a statement is executed. We only remember the path taken for an iteration and the corresponding iteration number. A path that is taken during the execution of an LCC is associated with the corresponding iteration number of the loop. If we find that the current path was already taken, we replace this path's iteration number by the current iteration number. Each path can be identified by a unique sequence of basic blocks (or statements) of length q and the number of such paths is limited. Hence, the space requirement is bounded. The run-time overhead is also much smaller compared to the Agarwal's method [1] and Zhao's method [9], since for any iteration we need only to identify which path is taken. After execution of the loop, the different paths taken during the loop execution are ordered according to the associated iteration numbers. Thus, we know the exact node and the edge sequence which gets executed during the loop iteration. Therefore, we can construct the relevant nodes and edges in the DDG after termination of the loop. This new DDG for the program will be compact one and we name it as *Compact Dynamic Dependence Graph* (CDDG). CDDG represents different paths that were taken during execution of a program with LCCs.

For computation of the CDDG, we first construct the control dependence (cd) edges at compile time. Control dependence does not change at run-time and therefore can be computed statically. Data dependence (dd) edges are added on-the-fly as the program execution proceeds. At compile time all the LCCs are identified and corresponding CFGs for the LCCs are constructed statically. The set of paths that may be taken during execution for each of these LCCs is then determined and stored in terms of the basic blocks and are unmarked. During execution of the program, data dependence edges that get executed during any iteration of the loop, the path that is taken is marked with the corresponding iteration number. After the loop terminates, the CDDG can be constructed considering only those vertices and edges that were marked and omitting the rest. The pseduo-code of our proposed algorithm for constructing CDDG is given below.

```
    int  i, n, x, y;
1.  cin>>n;
2.  y=4;
3.  i=1;
4.  while(i<n) {
5.      x=2;
6.      if(i mod 2 == 0) {
7.          if(n mod 4 == 0)
8.              x=4;
            else
9.              x=6;
        }
10. y=x;
11. i=i+1;
    }
12. cout<<y;
```

Fig. 3. An Example Program

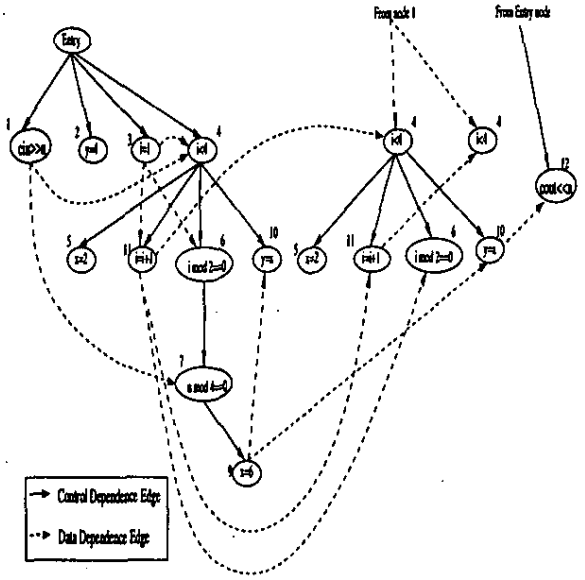

Fig. 4. Compact Dynamic Dependence Graph of the Example Program in Fig. 2.

Algorithm Generate CDDG

1) Run the program for the given input value
2) while (not termination) do
   a) n = node representing currently executed statement
   b) if ($n \neq L_{entry}$) // L_{entry} and L_{exit} are the entry and exit points of an LCC
      add all control dependence and data dependence edges to n
   c) else //an LCC is encountered
      i) repeat
         mark the path taken within the LCC with the iteration number
         // if the path is already taken update iteration number with the current iteration number
         until (n = L_{exit})
      ii) arrange the paths in increasing order of iteration numbers
      iii) for each path from the *lowest to* highest iteration number do
         A) create node for every statement executed during the iteration represented by this path
         B) add all control dependence and data dependence edges to each of these nodes

It is necessary to maintain the Ordering of the different paths taken, to correctly add the data dependence edges. For example, there are three paths p_1, p_2, p_3 and variable x is defined in two different statements in path $p_1 and p_2$ and is used in a statement in path p_3. Let the paths be executed in the order p_1, p_3, p_2, p_2, p_3 . At the termination of the loop, the data dependence edge from the statement in path p_2 defining the variable x to the statement in path p_3 using x will be relevant and added to the CDDG instead of the edge from the statement in path p_1 defining the variable x. If the order of execution is stored, CDDG construction is simple. Fig. 4 shows the CDDG of the example program given in Fig. 3 for input n=6. The two paths (4, 5, 6, 10, 11) and (4, 5, 6, 7, 9, 10, 11) are repeated alternately starting from iteration 1 to iteration 5 and finally the loop terminates when i=6. These two sequences are marked with their latest iteration counts 4 and 5 respectively. All the vertices and edges that are executed when taking these two paths are marked. For instance, Fig. 4 includes a data dependence edge from the right most vertex 10 to the vertex 12 instead of a data dependence edge from the left most vertex 10. After construction of the CDDG, we apply the reachability criteria as usual to compute the dynamic slice. The dynamic slice with respect to the out put statement 12 consists of all the statements from which this vertex is reachable in the CDDG, namely, (1, 3, 4, 6, 7, 9, 10, 11, 12). The shaded vertices in Fig. 4 are included in the slice.

IV. Comparison with Related Work

In this section, we show that our method is efficient in terms of space and time, than the related algorithms.

A. Space Requirement

Zhao [9] computed the dynamic slice of an object-oriented program based on the *dynamic object-oriented dependence graph* (DODG). But for program shaving loops, the size of the DODG becomes unbounded. Also the worst case space complexity of the DODG based algorithm is $O(2^n)$, where n is the number of statements in the program.

In our method, it is necessary to store the different paths that are taken when an LCC is executed. Let the length of the sequence which represents the paths be q, which is equal to the number of levels as discussed earlier. Let p be the total number of paths in an LCC and k be the total number of LCCs. The space requirement to store all the paths for all

LCCs is $O(kpq)$. The space requirement to store the CDDG is $O(n + mkp)$, where n is the number of statements in the program and m is the number of statements in an LCC. The total space requirement is therefore $O(n)$, since k and p are constants. So, our method is space efficient than Zhao's [9] method.

B. Time Requirement

The dynamic slicing algorithm of Zhao [9] based on DODG, finds the dynamic slice for each occurrence of the nodes. If the dynamic slice corresponding to the present execution of the node u is different from all the dynamic slices corresponding to its previous executions, then it creates a new node for this execution along with its required dependence edges and stores the associated dynamic slice. When the number of stored dynamic slices for a node u becomes very large, the time required for performing comparison also increases substantially. In the worst case, the time complexity becomes exponential in the number of statements of the program.

But in our method, during any iteration of an LCC, only the path taken is recorded. The overhead for this step is $O(p.q^2)$, where p is the total number of paths in an LCC and q is the length of sequence which represents the paths. Both p and q are much less than n. So, our method is time efficient than that of Zhao [9].

V. Conclusions

We have proposed an efficient technique for computing dynamic slices of object-oriented programs based on the intermediate representation *CDDG*. We have shown that our algorithm is more time and space efficient than the existing algorithms. Although we have presented our slicing technique using C++ examples, the technique can easily be adapted to other object-oriented languages such as Java. We are now extending this approach to compute the dynamic slice of concurrent object-oriented programs.

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programmimg Lanuages Design and Implementation, SIGPLAN Notices, Analysis and Verification*, volume 25, pages 246–256, White Plains, NewYork, 1990.

[2] Z. Chen and B. Xu. Slicing object-oriented java programs. *ACM SIGPLAN Notices*, 36:33–40, 2001.

[3] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[4] L. D. Larson and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, German, March 1996.

[5] D. Liang and L. Larson. Slicing objects using system dependence graphs. In *Proceedings of International Conference on Software Maintenance*, pages 358–367, November 1998.

[6] R. Mall. *Fundamentals of Software Engineering*. Prentice Hall, India, 2nd Edition, 2003.

[7] Y. Song and D. Huynh. *Forward Dynamic Object-Oriented Program Slicing, Application Specific Systems and Software Engineering and Technology (ASSET'99)*. IEEE CS Press, 1999.

[8] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[9] J. Zhao. Dynamic slicing of object-oriented programs. Technical report, Information Processing Society of Japan, May 1998.