An Efficient Technique for Dynamic Slicing of Concurrent C++ Programs

D. P. Mohapatra, R. Mall, R. Kumar
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India
e-mail: {durga, rajib, rkumar}@cse.iitkgp.ernet.in

Abstract

In this paper, we propose a novel dynamic slicing technique for concurrent C++ programs. We introduce the notion of *Concurrent Program Dependence Graph* (CPDG). Our dynamic slicing technique uses CPDG as the intermediate representation and is based on marking and unmarking the edges in the CPDG appropriately as and when the dependencies arise and cease during run-time. Our approach eliminates the use of trace files.

1 Introduction

The concept of a program slice was introduced by Weiser [1]. A static slice consists of those parts of a program that affect the value of a variable selected at some program point of interest. The variable along with the program point of interest is referred to as a slicing criterion. A slicing criterion $< s, V >$ specifies a location s and a set of variables (V). A dynamic slice contains only those statements that actually affect the value of a variable at a program point for a given execution. Dynamic slices are smaller than static slices and have been found to be useful in debugging, testing and maintenance etc. [1, 2].

Many of the real life OOPs are concurrent. Generally, understanding and debugging of concurrent OOPs are much harder compared to those of sequential programs. An increasing number of resources are being spent in debugging, testing and maintaining these products. Slicing techniques promise to come in handy at this point. But research reports dealing with slicing of concurrent OOPs are scarce in literature [2]. Although researchers have extended the concept of program slicing to static slicing of concurrent OOPs, the dynamic slicing of concurrent OOPs is still being missing until now.

Efficiency is especially an important concern for slicing concurrent OOPs, since their size is often large. With this motivation, in this paper we propose

a novel dynamic slicing algorithm for computing slices of concurrent C++ programs. Only the concurrency issues in C++ are of concern, many sequential Object-Oriented features are not discussed in this paper. The representation for O-O features given by Larson and Harrold [3], can be used in our algorithm. We have named our algorithm *edge-marking dynamic slicing* (EMDS) algorithm. Our algorithm allows to completely eliminate the use of a trace file at run time to record the execution history.

The rest of the paper is organized as follows. In section 2, we present some basic concepts and definitions. In section 3, we discuss the intermediate program representation: *concurrent program dependence graph* (CPDG). In section 4, we present our edge-marking dynamic slicing (EMDS) algorithm. In section 5, we compare our algorithm with related algorithms. Section 6 concludes the paper.

2 Basic Concepts and Definitions

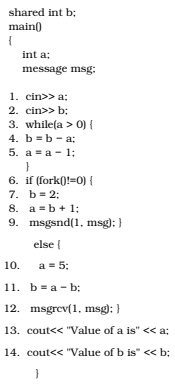
Definition 1. *Concurrent Control Flow Graph (CCFG).* A *concurrent control flow graph* (CCFG) G of a program P is a directed graph *(N, E, Start, Stop)*, where each node $n \in N$ represents a statement of P, while each edge $e \in E$ represents potential control transfer among the nodes. Nodes *Start* and *Stop* are unique nodes representing entry and exit of P respectively. There is a directed edge from node a to node b if control may flow from node a to node b.

Definition 2. *Concurrent Program Dependence Graph (CPDG).* A concurrent program dependence graph (CPDG) G_C of a concurrent OOP P is a directed graph (N_C, E_C) where each node $n \in N_C$ represents a statement in P. For x, y $\in N_C$, (y,x) $\in E_C$ *iff* one of the following holds:
(i) y is *control dependent* on x. Such an edge is called a *control dependence edge.*
(ii) y is *data dependent* on x. Such an edge is called a *data dependence edge.*
(iii) y is *fork dependent* on x. Such an edge is called a *fork dependence edge.*
(iv) y is *communication dependent* on x. Such an edge is called a *communication dependence edge.*

3 Construction of the CPDG

We have named our intermediate representation as *Concurrent Program Dependence Graph* (CPDG). This representation is later used to compute dynamic slices of concurrent object-oriented programs.

A CPDG of a concurrent OOP captures the program dependencies that can be determined statically as well as that may exist at run-time. Control dependency can be determined statically at compile time. The dependencies which dynamically arise at run-time are data dependencies, fork dependencies and communication dependencies. A CPDG can contain the following types of nodes: (i) *definition (assignment)* (ii) *use* (iii) *predicate* (iv) *fork* (v) *send*



```
shared int b;
main()
{
   int a;
   message msg;

1.  cin>> a;
2.  cin>> b;
3.  while(a > 0) {
4.  b = b − a;
5.  a = a − 1;
    }
6.  if (fork()!=0) {
7.  b = 2;
8.  a = b + 1;
9.  msgsnd(1, msg); }
      else {
10.    a = 5;
11.  b = a − b;
12.  msgrcv(1, msg); }
13.  cout<< "Value of a is" << a;
14.  cout<< "Value of b is" << b;
      }
```

Figure 1: An Example Program

and (vi) *receive*. Also, to represent different dependencies that can exist in a concurrent program, a CPDG may contain the following types of edges: (i) *control dependence edge* (ii) *data dependence edge* (iii) *fork dependence edge* and (iv) *communication dependence edge*. Fig. 2 shows the CPDG of the concurrent C++ program given in Fig. 1.

4 EMDS Algorithm

Before execution of a concurrent C++ program P, its CCFG and CPDG are constructed statically. During execution of the program P, we mark an edge when its associated dependence exists, and unmark when its associated dependence ceases to exist. We consider *data dependence* edges, *fork dependence* edges and *communication dependence* edges for marking and unmarking.

During execution of the program P, let *Dynamic_Slice (p, u, var)* with respect to the slicing criterion $< p, u, var >$ denotes the dynamic slice with respect to the most recent execution of the node u in process p. Let $(u, x_1), \ldots, (u, x_k)$ be all the *marked* outgoing dependence edges of u in the updated CPDG after an execution of the statement u. Then, it is clear that the dynamic slice with respect to the present execution of the node u, for variable *var* is given by
Dynamic_Slice(p, u, var) $= \{((p, x_1), \ldots, (p, x_k))\} \cup Dynamic\_Slice(p, x_1, var) \cup \ldots \cup Dynamic\_Slice(p, x_k, var)$.

Let *var_1*, *var_2*, ..., *var_k* be all the variables used or defined at statement u in the process p. Then, we define dynamic slice of the whole statement u as
dyn_slice(p, u) $= Dynamic\_Slice(p, u, var\_1) \cup Dynamic\_Slice(p, u, var\_2) \cup \ldots \cup Dynamic\_Slice(p, u, var\_k)$.

Our slicing algorithm operates in three main stages. In the first stage the CCFG is constructed from a static analysis of the source code. Also, in this stage using the CCFG the static CPDG is constructed. The stage 2 of the algorithm executes at run-time and is responsible for maintaining the CPDG as the

execution proceeds. The maintenance of the CPDG at run-time involves marking and unmarking the different dependencies such as data dependencies, fork dependencies and communication dependencies, as they arise and cease. The stage 3 is responsible for computing the dynamic slices for a given slicing criterion using the latest CPDG. Once a slicing criterion is specified, the dynamic slicing algorithm computes the dynamic slice with respect to any given slicing criterion by looking up the corresponding *Dynamic_Slice* computed during run time.

Working of the EMDS Algorithm

Consider the example program of Fig. 1. The CPDG is given in Fig. 2. Let the process ID for the start process of Fig. 1 be p_1. For the input values a=3 and b=1, we explain how our algorithm computes the slice. A *fork* call is executed at statement 6 in p_1. Process p_1 (parent) forks a new process p_2 (child) after execution of statement 6. So the statements (numbered from 7 to 14) executed after statement 6 would have fork dependence on statement 6 in p_1.

We are interested to compute the dynamic slice for the slicing criterion $< p_1, 14, b >$. The updated CPDG of the program is shown in Fig. 2. We first unmark all the edges of the CPDG and set dslice(p, u) $=\phi$ for every node u of the CPDG. In the example program of Fig. 1 since variable b is a shared variable it can be updated by either of the processes p_1 or p_2. Assuming that the most recent update of the shared variable b was done by process p_2 at statement 11, we compute the dynamic slice. At node 14, the outgoing fork dependence edge (14, 6) and data dependence edge (14, 11) are shown as marked. Therefore, the dynamic slice for the slicing criterion $< p_1, 14, b >$ is computed at stage 3 of the algorithm as $\{(p_1, 6), (p_1, 7), (p_2, 10), (p_2, 11)\}$.

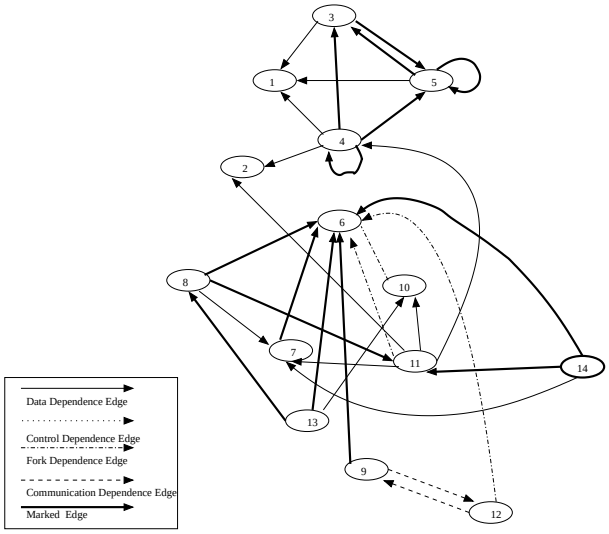


Figure 2: The updated CPDG of Fig. 1

5 Comparison With Related Works

Zhao computed the static slice of a concurrent OOP based on the *multi-threaded dependence graph* (MDG) [2]. He did not take into account that dependences between concurrently executed statements are not transitive. So, the resulting slice is not precise. Again, he has not addressed the dynamic aspects. Since our algorithm marks an edge only when the dependence exists, so this *transitivity* problem does not arise at all. So, the resulting slice is precise.

Krinke introduced an algorithm to get more precise slices of concurrent OOPs [4]. She had handled the *transitivity* problem carefully. So, in her algorithm, the *interface dependence* is not transitive. But she has not considered the dynamic aspects.

6 Conclusion

We have proposed a novel algorithm for dynamic slicing of concurrent C++ programs. We have named this algorithm *edge-marking dynamic slicing* (EMDS) algorithm. It is based on marking and unmarking the edges of the CPDG as and when the dependences arise and cease at run-time. Our algorithm does not use trace files to store the execution history. Our algorithm does not require to create additional nodes during run-time. This saves the expensive file I/O and node creation steps. Our technique can easily be adapted to other object-oriented languages such as Java.

References

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

[2] J. Zhao. Slicing concurrent java programs. In *Proceedings of the 7th IEEE International Workshop on ProgramComprehension*, May 1999.

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

[4] J. Krinke. Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33:35–42, April 1998.