Efficient Dynamic Slicing of Object-Oriented Program

Soon Hyung Park

Dept. of R&D, Korea Micro System
143-721, #1907 Technomart Bld., 546-4 Guui--dong, Gwangjin-gu, Seoul
Tel: +82- 2-3424-8420, Fax: +82-2-3424-8422, E-mail:nepaipark@nepaipark@hanmail.net

한국마이크로시스템 연구개발부
143-721, 서울특별시 광진구 구의동 테크노마트빌딩 1907
Tel: +82- 2-3424-8420, Fax: +82-2-3424-8422, E-mail:nepaipark@nepaipark@hanmail.net

Abstract

Traditional slicing techniques make slices through dependence graphs. They also improve the accuracy of slices. However, traditional slicing techniques require many vertices and edges in order to express a data communication links. Therefore the graph becomes complicated, and size of the slices is larger.

We propose the representation of a dynamic object-oriented program dependence graph so as to process the slicing of object-oriented programs that is composed of related programs in order to process certain jobs.

The efficiency of the proposed efficient dynamic object-oriented program dependence graph technique is also compared with the dependence graph techniques discussed previously. As a result, this is certifying that an efficient dynamic object-oriented program dependence graph is more efficient in comparison with the traditional dynamic object-oriented program dependence graph.

Keywords:

Program Slicing; Dynamic Program Slicing; Program Dependence Graph

1. Introduction

Program slicing is a progress of finding all statements in a program *P* that may directly or indirectly affect the value of a variable *var* at a point *p*. Accordingly, program slicing is a useful technique with other applications in program debugging by providing other programs that gather statements relating to an interested variable in a program [7]. Program slicing technique was proposed by Mark Weiser for the first time [10]. He introduced the first static slicing algorithm. It has been suggested a usage of this concept in the program testing, maintenance, debugging, and program understanding. During program debugging, the objective of slicing is to reduce the debugging effort by focusing the attention of the user on a subset of program statements which are expected to contain faulty code. Since debugging is performed by analyzing the statements of the program when it is executed using a specific input[4]. Dynamic object-oriented program slicing is working to get slices of object-oriented program by tracing the flow of classes that is the core of object-oriented program and objects when it is executed using a certain input data. Generally it is important that in the object-oriented program slicing we present polymorphism, dynamic binding, class inheritance, etc [8].

Traditional program slicing techniques often use graphs as a process of slicing to generate correct slices [9]. But the traditional dynamic object-oriented program dependence graph is complicated because that it need many vertexes and edges. So it is very difficult that programmer and tester use them to debug source programs.

In this paper, we proposed several processes to compute the result of dynamic object-oriented program dependence graph efficiently. We also demonstrated that this dynamic object-oriented program slicing technique is more effective than traditional object-oriented program slicing technique.

In section 2 and section 3, we review the studies concerning traditional dynamic program slicing approaches and dynamic object-oriented program dependence graph. In section 4, we account for the Efficient Dynamic Object-orient Program Dependence Graph (EDOPDG) that is proposed in this paper. In section 5, we introduce the processes to compute dynamic object-oriented program slices. In section 6, we apply the processes for the application programs. The EDOPDG technique is compared with traditional methods in section 6.

2. Dynamic Program Slicing

Program slicing is a course to generate program slices that is a set of statements that give effects to given variables directly or indirectly. The slicing technique is classified by the two criteria.

Firstly, it can be divided into static slicing and dynamic slicing by existence of execution history. Secondly, it can be divided into program slicing, system slicing and object-oriented program slicing by the number of programs that are objects of slicing [2][3][5].

Program slicing may be included the concept of system slicing. Especially, it may be called as procedure slicing where an object of the program slicing is single program. An important distinction of static slice and dynamic slice is that the former notion is computed without making assumptions regarding a program's input, whereas the latter relies on some specific test case[1].

Dynamic program slicing involves creating a slice of a program based on a specific input. The slice is therefore only valid for its associated input. A dynamic program slice criterion consists of *V* and *i*, just like a static backward slice. It also contains a sequence of input(s).

3. Dynamic Object-Oriented Program Dependence Graph

We use dependence graph notation traditionally in order to confirm the generation of correct program slices. Therefore, many types of dependence graph notation are introduced according to advanced program slicing techniques. In this section, we explain dependence graphs so as to express the program slicing. A dynamic dependence graph represents nodes with dependence edges based on the criterion node after the generation execution history for input value given. A new node for every occurrence of a statement in the execution history may need to be created if another node has the same transitive dependencies[12]. A slicing criterion of program *P* executed on specific input is a triple $C=(I^q, V)$, where *I* is an instruction at execution position *q* on execution history *H* and *V* is a subset of variables in *P*.

To find a dynamic slices of an object-oriented program, we construct a dependence-based representation namely, dynamic object-oriented program dependence graph (DOPDG) for a particular execution trace of the program. The DOPDG is a graph (V, A) where *V* is the multi-set of flow-graph vertices, and *A* is the set of edges represented dynamic control dependences and data dependences between vertices.

Control dependences represent control conditions on which the execution of a statement on expression depends. Informally, a statement *u* is directly control-dependent on the control predicate *v* of a conditional branch statement if *u* is executed or not is directly determined by the evaluation result of *v*.

Data dependences reflect the data flow between statements and expressions. Informally a statement u is directly data dependent on a statement *v* if the value of a variable computed at *v* has a direct influence on the value of a variable computed at *u*. figure 2 shows the DOPDG of the program in Figure 1 on input argv[1] = 3. The solid edges denote data dependencies and the dashed edges denote control dependencies.

An execution history is a set of the sequence $<v_1, v_2, \ldots\ldots, v_n>$ by order to be visited during execution of given test case. We use superscripts to distinguish between multiple occurrences of the same node in the execution history. The execution history of the example program shown Figure 1 is { 34, 35, 37, 2, 3, 4, 5, 38, 15, 16, 17^1, 18^1, 21^1, 22^1, 17^2, 18^2, 21^2, 22^2, 17^3, 11, 12, 39 } where argv[1] = 3.

```
E1 :   class Elevator {
         public:
2:       Elevator(int 1_top_floor)
3:      {  current_floor = 1;
4:         current_direction = UP;
5:         top_floor = 1_top_floor; }
6:      virtual ~Elevator() {}
7:      void up()
8:        { current_direction = UP; }
9:      void down()
10:       { current_direction = DOWN; }
11:     int which_floor()
12:       { return current_floor; }
13:     Direction direction()
14:       { return current_direction; }

15:     virtual void go (int floor)
16:       { if (current_direction == UP)
17:           { while ((current_floor != floor) &&
                        (current_floor <= top_floor))
18:              add(current_floor, 1); }
            else
19:           { while ((current_floor != floor) &&
                        (current_floor > 0))
20:                  add(current_floor, -1); }
          }

        private:
21:     add(int &a, const int& b)
22:       { a = a + b; };
        protected:
          int current_floor;
          Direction current_direction;
          int top_floor;
        };

23:   class AlarmElevator : public Elevator {
        public:
24:     AlarmElevator(int top_floor):
25:        Elevator(top_floor)
26:          { alarm_on = 0; }
27:     void set_alarm()
28:          { alarm_on = 1; }
29:     void reset_alarm()
30:          { alarm_on = 0; }
31:     void go(int floor)
32:       { if (!alarm_on)
33:            Elevator::go(floor)
          };

        protected:
          int alarm_on;
        };

34:     main(int argc, char **argv) {
          Elevator *e_ptr;
35:      if (argv[1])
36:          e_ptr = new AlarmElevator(10);
           else
37:          e_ptr = new Elevator(10);
38:        e_ptr->go(3);
39:        cout << "\n Currently on floor:"
                 << e_ptr->which_floor() << "\n";
        }
```

Figure 1 - Sample Program

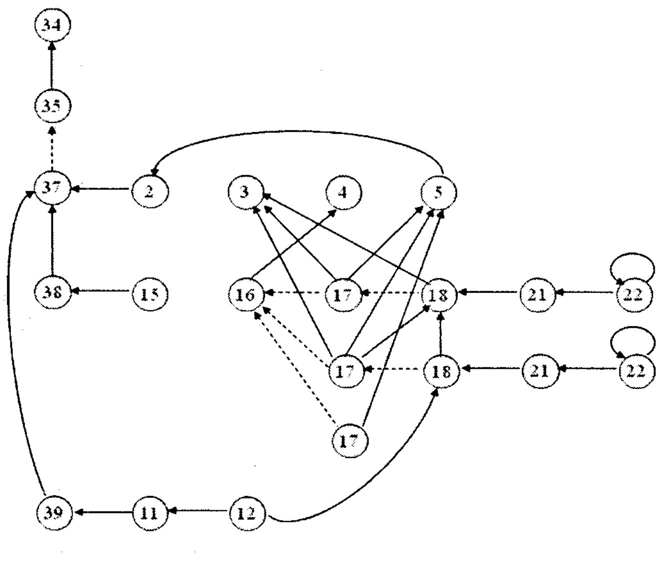

Figure 2 - DOPDG of Sample Program

4. Efficient Dynamic Object-Oriented Program Dependence Graph

An Efficient Dynamic Object-oriented Program Dependence Graph (EDOPDG) proposed in this paper is similar to the Program Dependence Graph(PDG) in the respect that the graphs represent the control dependence information by the control dependence edges and the data dependence information by the data dependence edges at the statements vertexes. The traditional object-oriented program dependence graphs is added the member variable edges, the call edges for construction of objects, the polymorphic call edges, the method call edges, etc. However, EDOPDG only is added the polymorphic call edges.

The process that is drawn up EDOPDG is as follows.

(1) We draw up edges in the graph using the static information of a source program within the limits of an execution history.

(2) After we compute the data dependence edges, we add them to the graph if the paths of them in the graph are not already existent.

(3) After we compute the control dependence edges, we add them to the graph if the paths of them in the graph are not in existence. Start nodes of control dependence are as follows.

• selection control nodes that are in the upper level of nodes that are in existence two times and over in the area from the criterion node to the exit node of data dependence.

• repetition control nodes that are in the upper level of nodes that exist in the area from the criterion node to the exit node of data dependence.

Control dependence nodes are as follows:
• class control dependence edges
• procedure control dependence edges
• method control dependence edges
• select control dependence edges
• repetition control dependence edges
• return control dependence edges

Figure 3 shows the EDOPDG of the program in Figure 1 on input argv[1] = 3. The solid edges denote data dependencies and the dashed edges denote control dependencies.

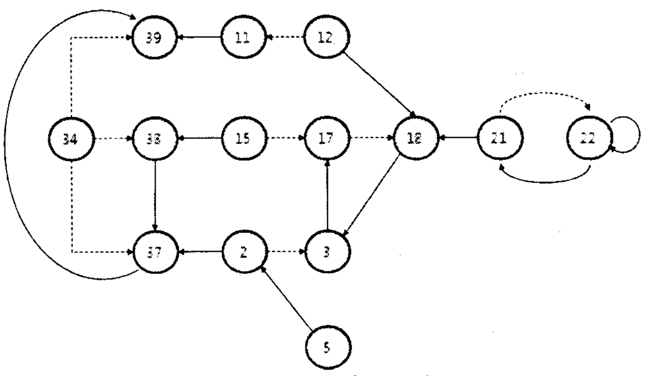

Figure 3 - EDOPDG of Sample Program

5. Computing Dynamic Slices of Dynamic Object-Oriented Program

The procedure that computes the dynamic object-oriented program slices using the efficient dynamic object-oriented program dependence graph (EDOPDG) is divided into four steps.

Firstly, a step of the program node analysis
Secondly, a step of the program execution history analysis
Thirdly, a step of the dynamic object-oriented program dependence graph generation
Finally, a step of the sliced program generation using the reverse tracking method.

$$\begin{tikzcd}
\bullet\ 39 \arrow[r] & 37 \arrow[r] \arrow[dr] & 38 \arrow[r] & 15 & & & & & \\
 & & 2 \arrow[r] & 5 \arrow[r] & 17 \arrow[r] \arrow[ul] & 3 \arrow[r] & 18 \arrow[r] & 21 \arrow[r] & 22
\end{tikzcd}$$

$$\bullet\ 39 \rightarrow 34$$

$$\bullet\ 39 \rightarrow 11 \rightarrow 12$$

Figure 4 - Reverse Traveling of EDOPDG where slicing criteria = (H, 39_{22}, which_floor)

We apply the dynamic object-oriented program slicing algorithm to an example program in Figure 1 in order to make dynamic object-oriented slices where argv[1] = 3 and slicing criteria = (H, 39_{22}, which_floor). A sliced program can be constructed by reverse- traversing the EDOPDG shown Figure 3 to compute dynamic object-oriented program slices where slicing criterion is which_floor of execution history order 39. Figure 4 shows the reverse traveling steps of nodes in EDOPDG where slicing criteria = (39^{22}, which_floor). The sliced program is illustrated in Figure 5.

```
class Elevator {
 public:
 Elevator(int 1_top_floor)
 { current_floor = 1;
     top_floor = 1_top_floor; }
 int which_floor()
   { return current_floor; }

 virtual void go (int floor)
       { while ((current_floor != floor) &&
                   (current_floor <= top_floor))
          add(current_floor, 1); }

 private:
 add(int &a, const int& b)
    { a = a + b; };
 protected:
   int current_floor;
   Direction current_direction;
   int top_floor;
 };

  main(int argc, char **argv) {
    Elevator *e_ptr;
      e_ptr = new Elevator(10);
    e_ptr->go(3);
    cout << "\n Currently on floor:"
           << e_ptr->which_floor() << "\n";
 }
```

Figure 5 - Sliced program

6. Efficiency Analysis

The efficiency data of the traditional dynamic object-oriented program dependence graph(DOPDG) and the efficient dynamic object-oriented program dependence graph(EDOPDG) proposed in this paper are represented.

6.1 Comparison of Complexities

The complexities that consist of the number of nodes and the number of edges are represented below.

Type	Complexities
ODPDG	56
EDOPDG	34

6.2 Comparison of the size of slices

The sizes of slices of the traditional DOPDG techniques and the EDOPDG technique proposed in this paper are represented below.

Type	Size of slices
DOPDG	17
EDOPDG	14

7. Conclusions

Static slices are a set of nodes that affect criterion variables. Dynamic slices are a set of nodes that affect actually the values of variables tracing on the test case. Therefore we can use usefully a dynamic concept in the field of the debugging through a test case.

We propose a dynamic object-oriented slicing technique using EDOPDG in this paper. We find that the complexity of the EDOPDG is 34 and the traditional complexity of the DOPDG is 56 with a result that we apply an example program of the figure 1 to the formulas of the complexities using the traditional DOPDG technique and the EDOPDG technique. The size of the slices of the EDOPDG is 14 where the slicing criterion is which_floor in the node 39. The sizes of the slices of the DOPDG is 17. The value of the complexities and the size of slices of the EDOPDG is smallest comparing with that of the DOPDG.

We find that the approach of the EDOPDG is more efficient compared with those of the DOPDG..

8. References

[1] Arpad Beszedes, Tamas Gergely, Zsolt Mihaly Szabo, Janos Csirik, Tibor Gyimothy, "Dynamic Slicing Method for Maintenance of Large C Programs.", Conference on Software Maintenance and Reengineering (CSMR), Lisbon, Portugal, pp.105-113, 2001.

[2] T. Wang and A. Roychoudhury, "Using Compressed

Bytecode Traces for Slicing Java Program", 26th International Conference on Software Engineering, pp. 512-521, Edinburgh, Scotland, UK, 2004.

[3] Yong, S. and Horwitz, S. "Using Static Analysis to Reduce Dynamic Analysis Overhead", Formal Methods in System Design Journal (FMSD), Nov. 2005.

[4] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, B. Korel, "Theoretical foundation of dynamic program slicing", Theoretical Computer Science Accepted, Jan. 2006.

[5] Hiralal. Agrawal and J. R. Horgan, "Dynamic Program Slicing.", Proc. ACM SIGPLAN'90 Conf. Programming Lang. Design and Implementaion, pp.246-256, 1990.

[6] J. Zhao, "Dynamic Slicing of Object-Oriented Programs," Technical-Report SE-98-119, pp.17-23, Information Processing Society of Japan (IPSJ), May 1998.

[7] X. Zhang, S. Tallam, and R. Gupta, "Dynamic slicing long running programs through execution fast forwarding.", In FSE, 2006.

[8] Loren D. Larsen and Mary Jean Harrold, "Slicing Object-Oriented Software.", Technicdal Report 95-103, Department of Computer Science, Clemson University, March 1995.

[9] Margaret Ann Francel, Spencer Rugaber, "The value of slicing while debugging.", Science of Computer Programming, Volume 40, Number 2-3, pp.151-169, July 2001.

[10] Mark Weiser, "Program slicing.", IEEE Trans. on Software Engineering, pp.352-357, July 1984.

[11] Park, S. H. and Park, M. G., "An efficient dynamic program slicing algorithm and its Application.", Proc. of the IASTED International Conference, Pittsburgh, Pennsylvania, pp.459-465, May 1998.

[12] Raghavan Komondoor, Susan Horwitz, "Tool Demonstration: Finding Duplicated Code Using Program Dependences.", European Symposium on Programming (ESOP), Genova, Italy, pp.383-386, 2001.