IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 1 Issue 6, August 2014.
www.ijiset.com
ISSN 2348 – 7968

Program Slicing and Approaches to Dynamic Slicing for Testing

Toshi Sharma[#1], Madhuri Sharma[*2]

Computer Science and Engineering Department, Bharat Institute of Technology, Meerut
Partapur By-pass Road Meerut.

Abstract— **A program slice is a subset of a program. Program Slicing is a process of finding all the statements in a program that directly or indirectly affect the value of a variable occurrence. Program slicing makes various tasks like debugging, testing and maintenance easier. This paper discusses about various types of slicing, approaches used to create slices. It also includes a brief description to test java programs using slicing**

Keywords— **Dynamic Program Slicing, Program Dependence Graph, Reduced Dynamic Dependence Graph, Execution trace, Slicing Criterion**

I. INTRODUCTION

Program Slicing enables programmers to view subsets of a program by filtering out code that is not relevant to the computation of interest. It is a technique for finding all statements in a program that directly or indirectly affected the values computed at some point of interest. This point of interest is referred to as the **slicing criterion.** A program slice is a subset of a program. The process of computing slices is known as program slicing. The slice computes the same function as the original program for the variables defined in the slicing criterion for all test cases.

The original concept of a program slice was introduced by **Weiser** [1,2,3]. According to Weiser a slice corresponds to the mental abstraction that people make when they are debugging a program. Various different methods to compute program slices have been proposed.

Program Slicing is used in many applications such as program verification, automatic parallelization of program execution, automatic integration of program versions, testing[16], maintenance[17] etc [3,8,15]. In this paper we primarily focus on two types of slicing i.e. static slicing and dynamic slicing. The computation of static slices does on rely on specific program's input, whereas in case of dynamic slicing computation of slices relies on specific test cases. This paper includes description of static slicing, program dependence graph, dynamic slicing, and approaches to dynamic slicing.

II. STATIC SLICING

A static program slice S consists of all statements in program P that may affect the value of variable v at some point p. The slice is defined for a slicing criterion C=(x,V), where x is a statement in program P and V is a subset of variables in P. A static slice includes all the statements that affect variable V for a set of all possible inputs at the point of interest (i.e., at the statement x). We can say that static slicing is similar to simple slicing technique. In static slicing only the statically available information is used so the computed slices are referred to as static slices.

Various methods have been suggested by researchers to construct the static slices. Ottenstein and Ottenstein [4] suggested the concept of **program dependence graph** (PDG) [5,6] to construct the static slices. Various other program slicing approaches used the modified and extended versions of PDGs. Another approach proposed by Bergeretti and Carre [15] defined slices in terms of information-flow relations which are derived from a program in a syntax-directed fashion.

The slices computed by performing the backward traversal of the program's control flow graph (CFG) or PDG starting from slicing criterion is referred to as the backward static slices. Forward slices consists of all statements and control predicates dependent on the slicing criterion. Backward and forward slices are computed in a similar way, forward slices requires tracing dependencies in the forward direction.

Example

```
int i;
int sum = 0;
int product = 1;
for(i = 0; i < N; ++i)
{
  sum = sum + i;
  product = product *i;
}
write(sum);
write(product);
```

This new program is a valid slicing of the above program with respect to the **criterion (write(sum),{sum})**

```
int i;
```

ISSN 2348 – 7968

```
int sum = 0;
 for(i = 0; i < N; ++i)
{
  sum = sum + i;
 }
write(sum);
```

A. *Program Dependence Graph*

Static slicing can be approached in terms of program reachability using **Program Dependence Graph** [7,12]. A PDG is a directed graph with vertices equivalent to statements and control predicates, and edges equivalent to data and control dependences. Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

Data dependence edge from vertex v_i to vertex v_j implies that that the computation performed at vertex v_i directly depends on the value computed at vertex v_j . Control dependence edge from vertex v_i to v_j means that vertex v_i may or may not be executed depending on the Boolean outcome of the predicate expression.

Example 1

```
begin
S1:    read(X);
S2:    if(X<0)
            then
S3:          Y:= f1(X);
S4:          Z:= g1(X);
             else
S5:          if(X=0)
                  then
S6:               Y:= f2(X);
S7:               Z:=g2(X);
                  else
S8:               Y:=f3(X);
S9:               Z:=g3(X);
                  endif;
              endif;
S10:   write(Y);
S11:   write(Z);
End
```

To find the static slice of the program in above example with respect to variable Y at statement 10, we first find all reaching definitions of Y at node 10. These are nodes 3, 6, and 8. Then we find the set of all reachable nodes from these three nodes in the Program Dependence Graph of the program. The set, (1, 2, 3, 5, 6, 8) gives us the desired slice.

III. DYNAMIC SLICING

The computation of dynamic slices relies on a specific test case. The slicing criterion consists of triple (input, occurrence of statement, variable of interest) so the input is fixed [9]. Dynamic slicing is much smaller than Static slicing.

The exact terminology “dynamic program slicing” was first introduced by Korel and Laski [10,11]. Advantage of dynamic slicing is the run-time handling of arrays and pointer variables. Dynamic slicing treats each element of an array individually, whereas static slicing considers each definition or use of any array element as a definition or use of the entire array. For e.g. Original program and the slicing criterion is **($\{n=2\}, 8^1, x$)**

```
1.  read (n)
2.  i=1
3.  while (i<=n) do
        {
4.      if (i mod 2==0) then
5.          x=17
6.      else
7.          x=18
8.  i= i+1
        }
9.  write (x)
```

The resulting program slice as per the slicing criterion will be

```
1.  read (n)
2.  i=1
3.  while (i<=n) do
     {
4.     if (i mod 2==0) then
5.         x=17
8.    i= i+1
      }
9.  write (x)
```

A. *Dynamic Slicing Approach 1*

To obtain the dynamic slice with respect to a variable for a given execution history, first take the ” projection " of the Program Dependence Graph with respect to the nodes that occur in the execution history, and then use the static slicing algorithm on the projected Dependence Graph to find the desired dynamic slice [7].

- All nodes in the graph are drawn dotted in the beginning.
- As statements are executed, corresponding nodes in the graph are made solid.
- Then the graph is traversed only for solid nodes, beginning at node, that contains the last definition of variable of interest in the execution history.
- All nodes reached during the traversal are made bold.

ISSN 2348 – 7968

- The set of all bold nodes, gives the desired slice.

Its problem is that it does not yield the precise dynamic slice everytime.

B. Dynamic Slicing Approach 2

Mark the edges of the Program Dependence Graph as the corresponding dependencies arise during the program execution [6,7]; then traverse the graph only along the marked edges to find the slice.

- All edges to be drawn as dotted lines in the beginning.
- As statements are executed, edges corresponding to the new dependencies that occur are changed to solid lines.
- Then the graph is traversed only along solid edges and the nodes reached are made bold.
- The set of all bold nodes at the end gives the desired slice.

Disadvantage : In the presence of loops, the slice may sometimes include more statements than necessary.

C. Dynamic Dependence Graph

Create a separate node for each occurrence of a statement in the execution history, with outgoing dependence edges to only those statements (their specific occurrences) on which this statement occurrence is dependent.

Every node in the new dependence graph will have atmost one out-going edge for each variable used at the statement. We call this graph the Dynamic Dependence Graph [7].

Once we have constructed the Dynamic Dependence Graph for the given execution history, we can easily obtain the dynamic slice for a variable, var, by first finding the node corresponding to the last definition of var in the execution history, and then finding all nodes in the graph reachable from that node.

Disadvantage : The size of a Dynamic Dependence Graph (total number of nodes and edges) is, in general, unbounded.

D. Reduced Dynamic Dependence Graph

We know that every program can have only a finite number of possible dynamic slices each slice being a subset of the (finite) program. This suggests that we ought to be able to restrict the number of nodes in a Dynamic Dependence Graph so its size is not a function of the length of the corresponding execution history. “Instead of creating a new node for every occurrence of a statement in the execution history, create a new node only if another node with the same transitive dependencies does not already exist.”

We call this new graph the Reduced Dynamic Dependence Graph [7]. To build it without having to save the entire execution history we need to maintain two tables called DefnNode and PredNode. DefnNode maps a variable name to the node in the graph that last assigned a value to that variable. PredNode maps a control predicate statement to the node that corresponds to the last occurrence of this predicate in the execution history thus far.

IV. Proposed Work

In recent software development, a programmer uses not only procedural languages like C and Pascal but also Object-Oriented languages like Java and C++ [14]. Since Object-Oriented languages include new concepts such as class, inheritance, dynamic binding and polymorphism, Object-Oriented programs have many dynamically-determined elements.

Software is often modified to reflect new functionality, with the changes of its specification. In the modification, several bugs are usually injected and so debugging is an important task in software evolution. So in order to make the task of finding the faults easier, my thesis work would focus on developing a framework for program slicing.

The steps involved would be

- Taking the java program as an input.
- Input the slicing criterion i.e. <input (t), occurrence of statement (l), variable of interest (v)>.
- Executing the input program against the given test case.
- Find the trace of the program according to the test case.
- Draw the reduced dynamic dependence graph of the trace of the program
- Identify in the graph the node *n* labeled l and containing the last assignment to v
- Find in the graph the set of all nodes reachable from *n*, including *n*, is the dynamic slice of program with respect to variable v at location l for test t.

This process would generate the required executable slice and the it would be executed to perform the process of testing.

V. Conclusion

As a software is made up of a large number of lines of code and testing these lines of code is not an easy task. So we thought of applying the concept of slicing to the process of software testing.

ISSN 2348 – 7968

After reading all the theory of program slicing we have come to this conclusion that it can be used to make software testing easier.

Program slicing is the technique to compute slice of the program and reduce size of the program according to slicing criterion. Testing and debugging a smaller part of program is much easier than the complete very large program. It has been shown that dynamic slicing is efficient than static slicing as it takes less memory because the size of the slice produced dynamically is small. And there is no efficient tool for dynamic slicing for java programs. So my research work would focus on developing the framework for dynamic slicing of java programs.

References

[1] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

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

[3] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[4] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984. *SIGPLAN Notices* 19(5).

[5] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations.In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218,1981.

[6] Sonam Agarwal and Arun Prakash Agarwal, " Program Slicing using Test Cases", *International Journal of Computer Applications* (0975-8887), Volume 60 –No. 10 , December 2012.

[7] Hiralal Agarwal and Joseph R.Horgan , "Dynamic Program Slicing, " ACM SIGPALN'90 Conference on Program Slicing Design and Implementation, White Plains, New York, June 20-22 ,1990.

[8] Guo Suwei , ZHAO Ruilian, Ll Lijain, " Application of Dynamic Slicing in Test Data Generation", TSINGHUA SCIENCE AND TECHNOLOGY, ISSN 1007-0214 27/49 ,pp 150- 155, Volume 12, Number S1, July 2007.

[9] Frank Tip, A Survey of Program Slicing Techniques

[10] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13:187–195,1990.

[11] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[12] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[13] Tao Wang Abhik Roychoudhury School of Computing National University of Singapore 3 Science Drive 2, Singapore 117543 "Using Compressed Bytecode Traces for Slicing Java Programs".

[14] Harkishan Rathod and Kaushik Rana, "Testing Web Services by Applying Program Slicing," *International Journal of Advance Research in Computer Science and Management Studies* Volume 2, Issue 1, January 2014.

[15] J.-F. Bergeretti and B.A. Carr´e. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985.

[16] E. Duesterwald, R. Gupta, and M.L. Soffa. Rigorous data flow testing through output influences. In *Proceedings of the Second Irvine Software Symposium ISS'92*, pages 131–145, California, 1992.

[17] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[18] Aditya P. Mathur "Foundations of Software Testing," Pearson.

[19] Srinivasan Desikan and Gopalswamy Ramesh "Software Testing Principles and Practices," Pearson.

[20] http://jslice.sourceforge.net/

[21] https://www.st.cs.uni-saarland.de/**javaslicer**/