Program Slicing using Test Cases

Sonam Agarwal
Amity University, Noida

Arun Prakash Agarwal
Amity University, Noida

ABSTRACT

The main applications of program slicing include various software engineering activities such as program understanding, debugging, testing, program maintenance, complexity measurement and so on. Program slicing is a feasible method to restrict the focus of a task to specific sub-components of a program. It can also be used to extract the statements of a program that are relevant to a given computation. Applying slicing technique to software architectures can benefit software development in two main ways. The first one concerns maintenance of a component-based software. By using slicing tools on an architectural description, we can determine which components might be affected when a given component is modified. Second, architectural reuse can be facilitated. While reuse of code is important, reuse of software design and patterns are expected to offer greater productivity benefits and reliability enhancements.

Keywords

Program slicing, test cases, static slicing, dynamic slicing, control flow graph, program dependence graph.

1. INTRODUCTION

A program slice consists of various parts of the program that effect the values computed at some point of interest such a point of reference is known as a slicing criterion and is typically specified by a pair (program point, set of variables). The task of computing program slices is called program slicing [1][2]. Program slicing can be used in functional testing. Features of programming languages such as procedures, unstructured control flow, composite data types and pointers, and concurrency each require specific extensions of slicing algorithms. Static and dynamic slicing methods for each of these features are classified and compared in terms of accuracy and efficiency.

Program slicing is a program analysis technique which was first introduced by Weiser to aid in program debugging. A program slice is usually defined with respect to a slicing criterion. A slicing criterion is a pair < P, V >, where P is a program point of interest and V is a subset of the program's variables. A slice of a program P with respect to a slicing criterion SC is the set of all the statements of the program P that might affect the slicing criterion for every possible input to the program [1][15]. Since the publication of Weiser's seminal work, the concept of slicing has been extended and many slicing algorithms have been proposed in the literature for other areas of program analysis such as program understanding, compiler optimization, reverse engineering, etc.

The slices mentioned so far are computed by gathering statements and control predicates by way of a backward traversal of the program's control flow graph (CFG) or PDG, starting at the slicing criterion [1]. Therefore, these slices are referred to as backward (static) slices. Informally, a forward slice consists of all statements and control predicates dependent on the slicing criterion, a statement being "dependent" on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine the fact if the statement under consideration is executed or not.

2. PROPOSED WORK

2.1 Program Slicing

Program Slicing is a technique in which programs are decomposed into smaller parts after analyzing their data [2]. The main applications of program slicing include various software engineering activities such as program understanding, debugging, testing, and program maintenance. A program slicing consists of those program statements which are related to the values computed at some program point.

Suppose there is a program of binary search in which elements are in sorted position and an item is to be searched. This program consist of three conditions. After making the program, slicing can be performed by using different-different techniques.

```
(1) int a[50],i,n,mid,high,low,item;
(2) int loc=0;
(3) printf("enter the number of element");
(4) scanf("%d",&n);
(5) printf("enter the elements in sorted order");
(6) for(i=0;i<n;i++)
(7) scanf("%d",&a[i]);
(8) printf("enter the number to be searched");
(9) scanf("%d",&item);
(10) low=loc=0;
(11) high=n-1;
(12) while(low<=high)
        {
(13)    mid=((low+high)/2);
(14)    if(item==a[mid])
                {
(15)      printf("search is successful");
(16)            loc=mid;
(17)            printf("\n loc of item is%d",loc+1);
                }
```

```
(18)        elseif(item<a[mid])
(19)      high=mid-1;
(20)      else
(21)      low=mid+1;
           }
```

Fig 1: Program for slicing

Fig 1 shows the portion of a program which can read any item from the input elements and can search the item frequently. In order to extract a slice from a program, the dependencies between the statements must be computed first. The control flow graph (CFG) is a data structure which makes the control dependencies for each operation in a program explicit[3][4].

components and a statement is exercised by a test if it is executed then the program will run on that test.

Test-data adequacy criteria can be divided into three groups: control flow based criteria, data-flow based criteria, and program dependence graph (PDG) based criteria [9]. Satisfying an adequacy criterion provides some confidence that the test suite does a reasonable job of testing the program [8].

The all-statements criterion is satisfied by a test suite T if for each statement s there is some test case t in T that exercises s. A statement is exercised by test case t if it is executed when the program is run with input t.

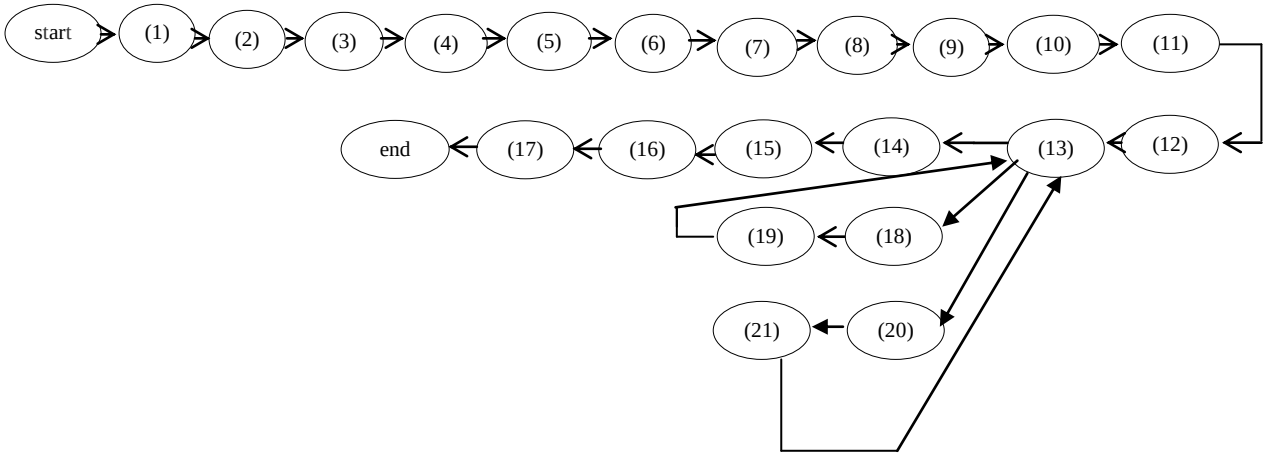

Fig 2: control flow graph (CFG)

2.2 Test Cases

A test data adequacy criterion is a minimum standard that a test suite for a program must satisfy. An adequacy criterion is specified by defining a set of "program components". An example is the all-statements criterion, which requires that all statements in a program must be executed by at least one test case in the test suite. Here statements are the program

Test-Sets	Array Elements	Search Item	Input		Output Loc
			Low	High	
t1	10,20,30,40, 50,60,70	20	0	6	2
t2	10,20,30,40, 50,60,70	40	0	6	4
t3	10,20,30,40, 50,60,70	60	0	6	6

Table 1: Test Cases

3. Program Slicing techniques

3.1 Static Slicing

A static slice for a given variable at a given statement contains all the executable statements that could possibly affect the value of this variable at the statement on all inputs.

A program slice consists of the parts or components of a program that affect the values which are referred to as a slicing criterion [5]. Typically, a slicing criterion consists of a pair < S, V >, where S is the statement number and V is a variable. The components of a program which have a direct or indirect effect on the values computed at a slicing criterion < S, V > are called the program slice with respect to the slicing criterion < S, V >.

Now static slicing is performed on the above program. Static slicing is similar to the simple slicing technique. Static slicing simply means that remove one variable and then eliminating all those conditions which are using removing variable [14].

```
(1) int a[50],i,n,mid,high,low,item;
(3) printf("enter the number of element");
(4) scanf("%d",&n);
(5) printf("enter the elements in sorted order");
(6) for(i=0;i<n;i++)
(7) scanf("%d",&a[i]);
```

```
(8) printf("enter the number to be searched");
(9) scanf("%d",&item);
(10) low=0;
(11) high=n-1;
(12) while(low<=high)
          {
(13)      mid=((low+high)/2);
(14)      if(item==a[mid])
                    {
(15)      printf("search is successful");
                    }
(18)         elseif(item<a[mid])
(19)      high=mid-1;
(20)      else
(21)      low=mid+1;
          }
```

Fig 3: sliced program or static slicing

Fig 4 shows the slice of the program with respect to the slicing criteria. All variables that are not relevant to the computation is sliced away. Statistically available information is used for slicing hence this type of slicing is called as static slicing.

Static slicing can be approached in terms of program reachability using Program Dependence Graph (PDG) [3]. A PDG is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependences. The slicing criterion is identified with a vertex in the PDG, and a slice corresponds to all PDG vertices from which the vertex under consideration can be reached. The slices are computed by gathering statements and control predicates by way of a backward traversal of the program's control flow graph (CFG) or PDG, starting at the slicing criterion. In PDG of the program solid arrows denotes the control dependencies and dotted arrow denotes the flow dependencies.

3.2 Dynamic Slicing

During program slicing, the slicing criterion contains the variables which produced an unexpected result on some input to the program [12]. However, a static slice may contain statements which have no influence on the values of the variables of interest for the particular execution. During execution of a program, the value inputted may cause unexpected result. Dynamic slicing takes the input supplied to the program during execution and the slice contains only the statement that caused the failure during the specific execution of interest. Dynamic slicing uses dynamic analysis to identify all and only the statements that affect the variables of interest on the particular anomalous execution trace [10]. The advantage of dynamic slicing is the run-time handling of arrays and pointer variables. Dynamic slicing will treat 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. Similarly, dynamic slicing distinguishes the objects that are pointed to by pointer variables during a program execution.

Dynamic slicing is much smaller than Static slicing because here slicer can compute the specific control and data flow dependencies produced by the provided input data. Now dynamic slicing is performed on fig 4. In dynamic slicing remove that part of the program which is no necessary to run a simple program.

```
(1) int a[50],i,n,mid,high,low,item;
(2) int loc=0;
(3) printf("enter the number of element");
(4) scanf("%d",&n);
(5) printf("enter the elements in sorted order");
(6) for(i=0;i<n;i++)
(7) scanf("%d",&a[i]);
(8) printf("enter the number to be searched");
(9) scanf("%d",&item);
(10) low=loc=0;
(11) high=n-1;
```

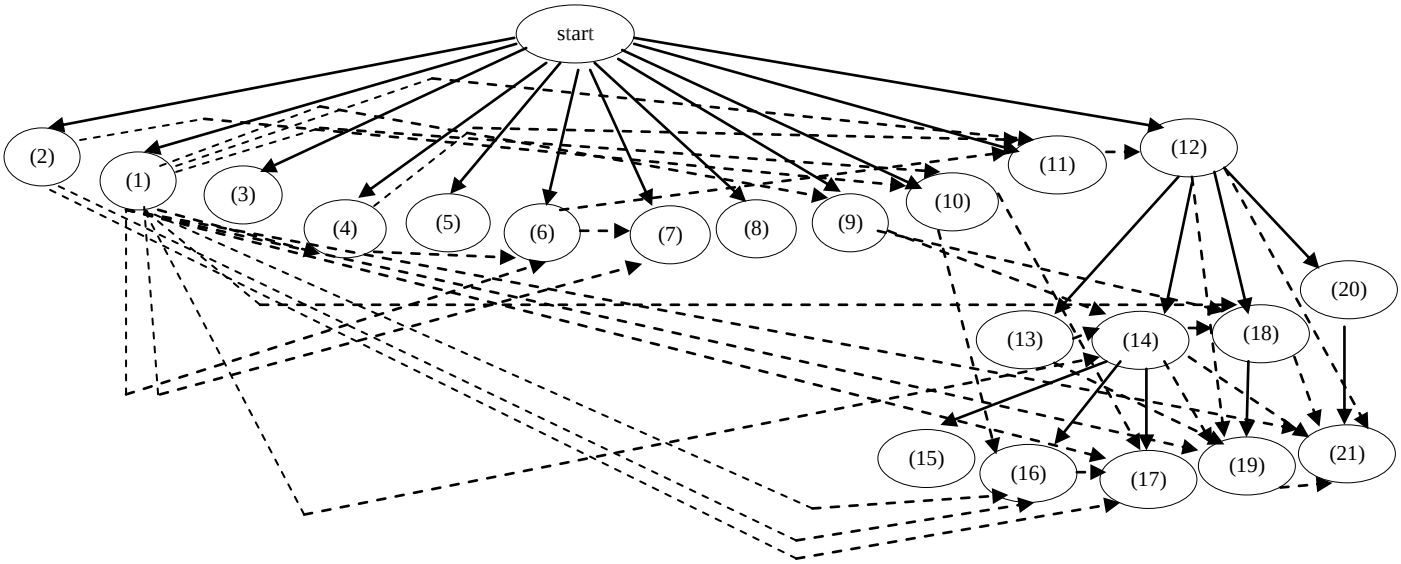

Fig 4: Program Dependence Graph (PDG)

```
(12) while(low<=high)
          {
(13)      mid=((low+high)/2);
(14)      if(item==a[mid])
                    {
(15)       printf("search is successful");
(16)                 loc=mid;
(17)                 printf("\n loc of item is%d",loc+1);
                    }
(18)        elseif(item<a[mid])
(19)      high=mid-1;
(20)      else
(21)                 ;
          }
```

Fig 5: Dynamic slicing

4. Conclusion

The use of the slice for regression testing is efficient in terms of both memory and time overhead. Unlike previous regression techniques, this approach neither needs to completely recomputed data flow information after a change nor maintain a history of previous data flow computation for incremental updates. Instead, the approach recomputes the partial data flow that is needed, as driven by the program changes. Also, the approach does not need the overhead of maintaining a test suite, which includes the input, output and updates of the test suite. If the test suite is maintained, the approach reduces the number of tests that must be rerun to provide full testing coverage and to update the test suite. Although the technique has been presented to satisfy the all-uses criterion, it could easily be modified for other data flow testing criteria.

Program slicing is a useful tool for working on the incremental regression testing problem. Unlike older approaches, which identify only directly affected components, slicing can identify indirectly affected down-stream components. Slicing can also be used to determine if two components have the same execution behavior. The use of the slice for regression testing is efficient in terms of both memory and time overhead. The application of slicing in various areas likes debugging, cohesion measurement, comprehension, maintenance and re-engineering and testing are highlighted.

5. Future Scope

Extended analysis of slicing in the form of object oriented programming (OOP), web development, procedural, component based, database aspect has to be focused. Here only two techniques of slicing are discussed. In future all other techniques like forward slicing, backward slicing, Quasi Static slicing, Conditioned slicing, Debugging slicing, Amorphous slicing, Functional Cohesion slicing etc. can be performed. Slicing techniques can be performed through the graph based approaches like Control Flow Graph (CGF), Program Dependence Graph (PDG), Object oriented Program Dependency Graph (OPDG), Dynamic Dependence Graph (DDG), Dynamic Object oriented Dependence Graph (DODG), Object oriented Dependency Graph(ODG), Class Hierarchy Sub graph (CHG), Control Dependence Sub graph (CDS) and Data Dependence Sub graph(DDS). Algorithmic approach to slicing can also be focused.

6. References

[1] Swarnendu Biswas and Rajib Mall, "Regression Test Selection Techniques: A Survey"**,** Information and Software Technology, Vol. 52, no. 1, January 2010

[2] Jaiprakash T Lalchandani, R Mall, "Regression Testing Based-on Slicing of Component-based Software Architectures", ISEC ,vol. 79, no. 06, pp. 19-22, 2008

[3] Rajiv Gupta, Mary Jean Harrold, Mary Lou Soffa, "An Approach to Regression Testing using Slicing", ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, pp. 26-60, January 1990

[4] N.Sasirekha, A.Edwin Robert, Dr.M.Hemalatha, "Program slicing techniques and its applications", International Journal of Software Engineering & Applications (IJSEA), Vol. 2, No. 3, pp.85-92, July 2011

[5] Mithun Acharya, Brian Robinson, "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems", ICSE, vol. 11, pp. 21–28, may 2011

[6] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang, Wu Lin Chen," A Brief Survey Of Program Slicing", ACM SIGSOFT Software Engineering, Vol. 30, no. 2, pp. 1-36, March 2005

[7] Josep Silva, "A Vocabulary of Program Slicing-Based Techniques**",** ACM Computing Surveys, Vol. 44, No. 3, Article 12, June 2012

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs", ACM Transaction on Programming Languages and Systems, 1990, pp. 26-61

[9] Frank Tip, "A Survey of Program Slicing Techniques**",** Journal of Programming Languages, Vol. 3, No. 3, pp. 121– 189

[10] David Binkley, "The Application of Program Slicing to Regression Testing"

[11] Debasis Mohapatra, **"**GA Based Test Case Generation Approach for Formation of Efficient Set of Dynamic Slices", International Journal on Computer Science and Engineering (IJCSE),Vol. 3, No. 9, September 2011

[12] Amogh Katti, Sujatha Terdal, "Program Slicing for Refactoring: Static Slicer using Dynamic Analyser"**,** International Journal of Computer Applications, Vol. 9, No. 6, November 2010

[13] Hiralal Agrawal, Joseph R. Horgan, "Dynamic Program Slicing", ACM SIGPLAN Notices, Vol. 25, No. 6, pp. 246-256, June 1990

[14] Z. Chen, B. Xu, and J. Zhao, "An Overview of Methods forDependence Analysis of Concurrent Programs", ACM SIGPLAN Notices, Vol. 37, No. 8, pp. 45-52, 2002

[15] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen, "Regression testing for web applications based on slicing. In Proceedings of the 27th Annual International Computer Software and Applications Conference",IEEE Computer Society, pages 652–656, Los Alamitos, CA, USA, November 2003

[16] J. Bible, G. Rothermel, and D. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques", ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2, pp. 149–183, April 2001

[17] S.S. Anju, P. Harmya, Noopa Jagadeesh, R. darsana, "Malware detection using assembly code and control flow graph optimization", ACM Digital library, No. 52, 2010