Dynamic Slicing Object-Oriented Programs for Debugging[*]

Baowen Xu Zhenqiang Chen
Department of Computer Science and Engineering, Southeast University, China
State Key Laboratory of Software Engineering, Wuhan University, China
{*bwxu, chenzq*}*@seu.edu.cn*
Hongji Yang
Department of Computer Science, De Montfort University, England

Abstract

Dynamic program slicing is an effective technique for narrowing the errors to the relevant parts of a program when debugging. Given a slicing criterion, the dynamic slice contains only those statements that actually affect the variables in the slicing criterion. This paper proposes a method to dynamically slice object-oriented (OO) programs based on dependence analysis. It uses the object program dependence graph and other static information to reduce the information to be traced during program execution. It deals with OO features such as inheritance, polymorphism and dynamic bindings. Based on this model, we present methods to slice methods, objects and classes. We also modify the slicing criterion to fit for debugging.

1. Introduction

Program slicing is an effective technique for narrowing the focus of attention to the relevant parts of a program. The slice of a program consists of those statements and predicates that may directly or indirectly affect the variables computed at a given program point [21]. The program point s and the variable set V, denoted by <s, V>, is called a slicing criterion. The basic idea of program slicing is to remove irrelevant statements from source codes while preserving the semantics of the program such that at the specified point the variable produces the same value as its original program. Program slicing has been widely used in many software activities, such as software analyzing, understanding, debugging, testing, maintenance, and so on [2, 12, 19]. Slicing algorithm can be classed according to whether they only use static information (static slicing) or dynamic execution information for a specific program input (dynamic slicing). This paper focuses on the dynamic slicing methods.

There have many dynamic slicing methods proposed in literature [1, 10, 11, 13, 18]. Most of them can be divided into two categories: forward and backward slicing. There is not much work on slicing object-oriented programs (OOPs) until recently [14, 15, 20], and only few concern dynamic methods [17, 26]. The main difficulty of dynamic slicing is to obtain the running-time information. To do this, the execution of the program has to be traced.

Agrawal proposes a dynamic slicing method by marking nodes or edges on a static program dependence graph during execution [1]. The result is not precise, because some dependencies might not hold in dynamic execution. Agrawal also proposes a precise method based on the dynamic dependence graph (DDG) [1], and Zhao applies it to slice object-oriented programs [26]. The shortcoming is that the size of the DDG is unbound.

Korel [10], Song [17] and Tibor [18] propose forward dynamic slicing methods and Song also proposes method to slice OOPs using dynamic object relationship diagram (DORD) [17]. In these methods, they compute the dynamic slices for each statement immediately after this statement is executed. After the last statement is executed, the dynamic slices of all statements executed have been obtained. However, only some special statements in loops need to compute dynamic slices.

In our previous work, we have developed static slicing methods for OOPs [5, 23] and dynamic methods based on dependence analysis [4]. This paper extends our previous methods to dynamically slice OOPs. It uses the object program dependence graph (OPDG) and other static information to reduce the information to be traced during execution. It combines forward analysis with backward one. In the forward process, it marks information on the OPDG and computes dynamic slices at the necessary points during the program execution. In the backward process, it traverses the OPDG marked to obtain the final dynamic slices.

[*] This work was supported in part by the National Natural Science Foundation of China (NSFC) (60073012), Natural Science Foundation of Jiangsu, China (BK2001004), Opening Foundation of State Key Laboratory of Software Engineering in Wuhan University, Foundation of State Key Laboratory for Novel Software Technology in Nanjing University, Opening Foundation of Jiangsu Key Laboratory of Computer Information Processing Technology, Visiting Scholar Foundation of Key Lab. in University, and SEU–NARI Foundation.

The rest sections are organized as follows. Section 2 provides an overview of the basic notions. Section 3 discusses the object program dependence graph used in our approach. Section 4 provides a detail description of our dynamic slicing criterion. Our dynamic slicing algorithms, including dynamic method slicing, object slicing and class slicing are presented in Section 5. Section 6 shows how to apply our dynamic slicing method to program debugging. Conclusion remarks are given in the last section.

2. Basic Notions

This section introduces some basic notions required by our dynamic slicing methods, including the program dependence graph, execution history and dynamic slicing criterion.

2.1. Program Dependence Graphs

Program dependencies are dependence relationships between statements in a program that are implicitly determined by the control and data flows in the program. Program dependence analysis is an analysis technique to identify and determine various program dependencies in program source codes and then represent them in some explicit forms convenient for various applications [6, 8, 22]. There are two basic types of dependence: *data dependence* and *control dependence*.

Informally, for statements s_1 and s_2, if the execution of s_1 determines whether s_2 can be executed or not, s_2 control depends on s_1, denoted by CD(s_2, s_1). If s_2 refers a variable defined by s_1, s_2 data depends on s_1, denoted by DD(s_2, s_1).

The program dependence graph (PDG) of a subprogram P is a direct graph, G = <S_S, S_E>, where S_S is the node set, each of which represents a statement or predicate expression. S_E is the edge set, in which each edge e = <s_1, s_2> represents DD(s_1, s_2) or CD(s_1, s_2).

To compute dependencies, control flow graph (CFG), a common way to represent programs, is often introduced. The CFG of a procedure is a direct graph, G = <S, E, s_{entry}, s_{exit}>, where S is node set, which represents statements or predicate expressions. E={<s_1, s_2>|s_1, $s_2 \in$ S and after the execution of s_1, it is possible to execute the statement s_2 immediately.}. s_{entry} is the entry node, and s_{exit} is the exit node of P. If <s_1, s_2>$\in$ E, s_1 is a direct predecessor of s_2, denoted by Pre(s_1, s_2). Let Pred(s) = {s′| Pre(s′, s)}.

Control dependencies can be obtained by analyzing the program structures in the CFG. To compute data dependencies, we define the following sets.

(1) Def(s) denotes the variables which values are defined (modified) at s.

(2) Ref(s) denotes the variables which values are refereed, but not modified at s.

(3) In(s) ={(s', x) | x was last defined at s' before the execution of s}.

(4) Out(s) ={(s', x)| (s', x)$\in$ In(s)$\wedge$x$\notin$ Def(s)} $\cup${(s, x)| x$\in$ Def(s)}.

The In set of the entry node of a subprogram is empty, and the Def set includes all the formal parameters. For any node s:

$$In(s) = \bigcup_{s' \in Pred(s)} Out(s')$$

According to the definitions above, let *v* be a variable, we have:

$$\exists v(v \in Ref(s) \wedge (s', v) \in In(s)) \Rightarrow DD(s, s')$$

In modern programming languages, such as Java, C/C++, one statement might define more than one variable. To get more precise dependence information, we add additional nodes to distinguish different definitions. For a statement s, if Def(s) ={v_i | v_i is defined at s, i>1}, add i nodes s_i in the PDG, each represents a unique definition of v_i, i.e. Def(s_i)={v_i}, and all s_i depend on s.

A CFG is a static representation for program. The path in a CFG might not be an actual executing path. Therefore, in dynamic analysis, execution history is introduced to represent the execution path [18].

2.2. Execution History and Dynamic Slicing Criterion

An execution history is a feasible path in the CFG that has actually been executed, denoted by EH [18]. The statements contained in the execution history are in the same order as they haven been executed. As the example shown in Figure 1, let the input be a =1, c=3, the execution history is EH1 = <1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 7, 8, 9, 11, 12, 7, 13>. EH (j) gives the serial number of the statement executed at j th step of EH, such as EH1(17) = 13. To distinguish the different occurrence of the same statement in an EH, a number pair (i, j), denoted by i^j, is introduced, where i is the serial number of the statement and j is the step number of i in the EH.

```
1. cin>>a;               8.    if (s>0)
2. cin>>c;               9.       x = 2
3. if (a >0)                   else
4.   i = 2               10.      x =s+1;
 else                    11.   s = s + x;
5.   i =1;               12.   i ++;
6. s = 0;                  }
7. while (i <= c) {      13.cout<<s;
```

Figure 1. A program segment

Each program slice is based on a rule, named as slicing criterion. In static slicing, the slicing criterion has the form <i, V>, where i is the serial number of a statement in a program point and V is a variable set. Dynamic slicing criterion is a triple <NPUT, i, V>, where INPUT denotes the actual input during an execution. When execution history is introduced, the dynamic slicing criterion will be $<i^j$, EH, V >, where EH is the execution history on a given input. i^j denotes the i statement at position j in the EH. If only the variables referred at i are concerned, V might be omitted. In the following discussion, V is omitted. For the execution history EH1 above, slicing criterion $C1=<13^{17}$, EH1> means slicing on statement 13 in fact. After the slicing criterion is determined, dynamic slices will be computed by different kinds of slicing methods.

3. Object Program Dependence Graphs

A PDG describes the dependencies in a single subprogram. To model the dependencies among subprograms, system dependence graphs (SDG) are introduced by Horwitz [8]. This section extends the traditional PDGs to represent OO features including class, inheritance and polymorphism. These are the bases of our dynamic slicing method, although they are static. In most OO programming languages (OOPL), the term *class* has three distinct and related aspects: *template*, *type* and *a set of objects*. A class consists of a set of attributes (data members) and methods (operations on the attributes). We first analyze the PDGs of methods.

3.1. Inter-Method PDGs

The PDG of a single method is the same as the traditional one presented in Section 2. We do not repeat them in this section. The paper supposes that a parameter can be passed by value or by reference. The non-local variables are treated as parameters. The return value is also treated as a parameter, whose name is the same as the method's name.

In this paper, we introduce *inter-parameter dependence set* to describe the dependencies among parameters (In SDG, the summary edges present these dependencies). Informally, let ParaO be a formal parameter passed by reference of a method M, if the definition of ParaO is directly or indirectly dependent on a formal parameter ParaI, ParaI belongs to ParaO's inter-parameter dependence set. All the formal parameters on which ParaO depends are denoted by Dep_P(M, ParaO).

A method invoking/call statement is a special node that might define more than one actual parameter. To get more precise information, we should add corresponding nodes at each call site, each of them represent a unique definition of an actual parameter. For such a node s which represents actual parameter ParaA, let ParaFA be the formal parameter of ParaA, then Def(s) ={ParaA}, and Ref(s) ={PA| PF$\in$ Dep_P(M, ParaFA), and PA is the corresponding actual parameter of PF at the call site}. For the parameters of M, if $PF_1 \in$ Dep_P(M, PF_2), the node represents the corresponding actual parameter of PF_2 depends on that of PF_1.

After these conversions, method-invoking statements are as easy to handle as other statements. In our method, when analyzing a method M, we must know the dependencies among parameters of methods invoked by M. Thus the methods should be analyzed in a given order. This can be done by a call graph. In a call graph, by repeatedly removing the node whose out degree is 0 and the edges connected to it we can get the analysis sequence, i.e. the sequence of removing nodes. A circle in the call graph means there are recursive calls in the program. We have proposed methods to analyze dependencies among recursive subprograms in our previous work [24]. Due to the space limitation, we do not discuss them in detail here.

3.2. Representing OO Features

This section presents our representation for OOPs. It proposes methods to represent classes, inheritance and polymorphism. The final representation of an OOP is called an object dependence graph (OPDG).

• **Classes (Encapsulation)** A class is an entity that encapsulates a set of data members and methods. Corresponding to this, in our representation, the PDG of a "class" encapsulates data members and a set of PDGs, each of which represents a method of the class. Note that the PDG of each method is the "original" PDG, i.e. parameter passing and dynamic binding are not considered when constructing a class's dependence graph.

• **Inheritance** Inheritance is one of the most important features of an OOPL. It enables programmers to define new classes by explicitly stating how they differ from existing ones, rather than by specifying all their properties from scratch. When constructing the PDG of a subclass, the parent class's PDG need not be reconstructed. The derived class just "inherits" its parent's PDG. Thus, in this representation, each method has only one PDG that is shared by all the instances of the class and its derived class.

• **Polymorphism** Most OOPLs support polymorphism and dynamic binding. In static analysis, a polymorphic object is represented as a tree: the root represents the polymorphic object and the leaves represent objects of possible types. When it receives a message, each possible object has a call-site.

For a statically typed OO programming language, the possible types of an object can be determined statically

[3]. In our representation, since the PDG of the whole program is not a connected graph, we use a different approach in which the polymorphism is represented by the inter-parameter dependence sets. Each set is associated with an object type. When an object invokes a method, the inter-parameter dependence set is computed in two cases, according to whether the object type is determined or not.

(1) In the case of that the object type is determined. If the method is not a polymorphic method, no more analysis is needed, else change the polymorphic method to the actual one and reanalyze the dependencies of the polymorphic method if needed.

(2) In the case of that the object type is undetermined. If the method is not a polymorphic method, then the inter-parameter dependencies are the union of dependencies in all possible methods, else for each possible type change the polymorphic method calls to the corresponding methods and reanalyze the dependencies, the inter-parameter dependencies are the union of dependencies in possible methods.

4. Dynamic Slicing Criterion

In our program debugging practices, we find that when debugging the variables at a given program point, we often insert a breakpoint before executing the program. The i in a slicing criterion is a breakpoint in nature. To fit for debugging, we modify the slicing criterion in our slicing method.

The slicing criterion used in our method has the form <INPUT, i, j, FILENAME>, where INPUT denotes the actual input during an execution. i is different from the previous slicing criterion. It is not the serial number of a statement, but the line-number of a statement. For the user can get line-number more easily than the serial number in the edit environment. j is the loop number i, i.e. the jth occurrence of i. FILENAME is the name of file includes i. In modern IDE, different components of a program are saved in different files to support separate compilation and component reuses, such as the class in C++, program unit in Ada. Thus we use filename to distinguish the different statements in different files.

- The default of i is the end of the program (when i is default, j should be default). Such case is used when only the final results are concerned by the user. If line-number i is not a valid statement (such as comment line), i means the nearest statement (predecessor).
- If j is not given, j is the real time that i is executed. Thus we must compute the dynamic slice for every occurrence of i.
- If FILENAME is not given, it means the current file.

After these modifications, the slicing criterion C1 in Section 2.2 is converted to <INPUT, 13, 1> or <INPUT, 13>.

The following sections will discuss our slicing approach based on this criterion. These modifications are for the user's convenience to use this slicing method. It is no difference in nature with other slicing criterion.

In the analysis above, we suppose that:

(1) The dynamic program flow can reach to i and i can be executed j times. Because our slicing criterion is determined before execution, in a certain execution i might never be executed or less than j times. If this does not hold, the following processes are provided as default.

- If the program can complete, give the warning message "i can not be executed! ", and then compute slice by treating i as default.
- If the program can not finish normally (such as there is a dead loop), the method can not compute dynamic slice, because we can determine the program flow and the proper point to compute slice.

(2) There is no more than one statement in a line of codes. Otherwise, the default is the first statement. If the user wants to slice on other statement in the line, the simple way is to separate each statement in a line. Anther way is to decompose the line.

In the following algorithms, we suppose the above two suppositions hold.

5. Dynamic Slicing Based on OPDG

5.1. Preliminaries

Our approach differs from traditional forward dynamic slices [10, 17, 18] in that we compute dynamic slices at some special points, not all the statements. In this paper, we suppose that a program includes three basic structures: sequence, branch and loop. The following is the discussion about these three structures.

- For a sequential structure, if a statement is executed, all statements in the block will be executed. The static dependencies are the same as the dynamic ones. If they are not executed, mark them as "removed" (In our slicing algorithm, the executed statements are marked as "executed", the statements having no such mark are "removed".). The "removed " statements will be deleted first before traversing the static OPDG.
- For a branch structure, in a certain execution only one branch is covered. Therefore only the executed statements might affect the dynamic dependencies. By removing the unexecuted statement the redundant dependencies can be removed from the OPDG.

- For the combination of sequential and branch structures, the result is the same as analyzing the two structures separately.
- As for a loop structure, the execution histories of some statements must be recorded. A vector LD [18] is introduced to record the last definition positions of variables. Each element of LD has the form <V, i>. For statement s, an element <v, i> of LD(s) denotes that there is no other statement on the path from i to s, which redefines the variable v, just before s is executed. In a loop structure, the LD will change in different loops. The dynamic dependencies change with the change of the LDs. Therefore, we need to record the history information. Like the relevant slicing method [18], we use dynamic slices to record such information. Whether a dynamic slice is needed is determined by the statement itself. If LD is changed after the second execution of s, the dynamic slice might change after s. Thus we must compute the dynamic slice of s.

Mark all the statements executed. After the program exits the loop, compute the difference between dynamic LD and the static LD, denoted by KILL. For the successive statement s, if s depends on the statements included in KILL, mark the corresponding edges as “removed”.

5.2. Methods Slicing

Alg. 1 shows our methods slicing algorithm. The main idea is that: Compute the OPDG first using static analysis, then mark the nodes and dependence edges that do occur during execution, finally transverse the marked OPDG and obtain the dynamic slice. To implement our method, we insert breakpoints to the necessary statements to record the execution information.

(1) There have efficient algorithms proposed to construct PDG in literature [5, 7, 8, 16]. By mapping the dependencies among formal parameters to actual ones we can obtain the OPDG easily. We do not present them in detail here.

(2) Determine the statements at which dynamic slices are computed: As mentioned above, if LD is changed after the second execution of s, the dynamic slice of s should be computed during execution. To improve the efficiency of the program analyzed, the statements at which dynamic slices are computed should be as few as possible. By analyzing programs we find that: LD can be changed when and only when there is more than one statement that defines the same variable. Thus, it is easy to find all such statements by statically analyzing the CFG.

(3) Breakpoints might be inserted in the following cases:

a. The statement in the slicing criterion.
b. The statements at which dynamic slices need to be computed.
c. Divide the program into basic blocks according to the program structure. A breakpoint is inserted to the beginning statement of the block. A basic block must be a sequential structure i.e. each has a single entry and exit. Every block includes one breakpoint.

(4) The main task of process codes: The process program at the point of slicing criterion is to abort the execution at time j, and call the traversing program. Others will mark the nodes and edges; compute the LD and dynamic slices when needed.

(5) Traverse the OPDG: To obtain the final dynamic slice, the following steps are needed to traverse the marked OPDG.

a. Traverse the marked OPDG, remove the statement nodes unexecuted and edges marked with “removed” from OPDG.
b. Traverse the OPDG refined by the step a from node i, and obtain the node set S.
c. For each node s in S, mark s, if its dynamic slice DynSlice(s) had been obtained during program execution, $S=S \cup \{s_i | s_i \in \text{DynSlice}(s)$, and s_i is not marked$\}$.
d. Repeat step c until S does not change. The final node set S is the dynamic slice about statement i.

Input: Slicing criterion <INPUT, i, j>.
Output: Dynamic slice

```
1.Construct the OPDG statically.
2.Analyze the OPDG, and determine the points to which breakpoints will be inserted and the
  statements at which dynamic slices will be computed during program execution.
3.Insert breakpoints and the corresponding process codes to the points obtained from step 2.
4.Execute the program after insert breakpoints in INPUT.
5.Traverse the OPDG and obtain the dynamic slice.
```

Alg. 1. Methods slicing algorithm

Alg. 1 is an intra-method slicing algorithm. To compute inter-method slices, we treat method-invoking statement as a special type of loop statement. Different calls with different actual parameters might bring different results. We must record all the dynamic information. Therefore, we introduce slices on parameters to describe the dynamic dependencies among parameters.

Let x be a formal parameter passed by reference of M, s_{exit} is the exit node of M, M's slice on x is the slice in M on the slicing criterion <s_{exit}, x>. M's dynamic slice on x is its slice on x in a given INPUT, denoted by DynSlicePara(M, x).

For a node s, which represents an actual parameter of M, let x be the formal parameter corresponding to s,

DynSlice(s) = DynSlicePara(M, x).

To get more precise information, we must compute the dynamic dependencies among formal parameters. Then map them to actual ones. Such information can be obtained when computing dynamic slices on parameters.

5.3. Objects Slicing

Class and object are the two basic concepts in OOPs. Therefore when slicing an OO program, we should provide methods to slice objects and classes. In the following two sections, we focus our attention on *dynamic object slicing* and *dynamic class slicing*.

For software engineering activities such as debugging and testing, the user would like to focus attention on one object at a time. I.e. we are more interesting about the effect of an object on the variable at the program point rather than the whole program.

Thus, we introduce dynamic object slicing. To slice an object, we change the slicing criterion to <INPUT, i, j, Object>, where Object is the object interested. Informally, given a slicing criterion <INPUT, i, j, Object>, object slicing identifies the statements in the methods of the Object that might affect the computation of i in the slicing criterion <INPUT, i, j>.

An intuitive simple way to compute dynamic object slices can be described by two steps: Firstly, compute the dynamic slice of the whole program on the slicing criterion <INPUT, i>. Then, select the methods and statements of the Object and obtain the object slice.

In general, it is redundant to compute the dynamic slice of the whole program. In our dependence analysis method, the PDG of a class is shared by all its instances. When given a slicing criterion, the statements in a method identified by a slicing algorithm include statements in different instances of the class.

Thus we must find ways to extract the methods and statements of a certain object from those of all the instances of the class. Alg. 2 shows our dynamic object slicing algorithm.

(1) Step 1 is used to simple the cost to construct OPDG, because only partial methods related to the i and Object are needed. Let M be the method (subprogram) including statement i, all the methods called by M belong to MS. For a method (subprogram), which calls M, if it is a method of Object or it is called by any method of Object directly or indirectly, it belongs to MS.

(2) We need only construct the PDGs of the methods related.

(3) Using the static slicing, we can further reduce the number of methods to be traced. Call graph represents simple type control information, while static slicing combines the control and data flow information, thus static slicing can refine the MS. We have proposed static object slicing algorithm in our previous work [4, 23]. Due to space limitation we do not discuss them in detail here.

(4) For a method, which is not a method of Object, if it does not call any method of Object directly or indirectly, only its dynamic parameter dependencies are needed. If the users do not care about the little difference between

Input: Object slicing criterion <INPUT, i, j, Object>.
Output: Dynamic object slice.

```
1. Construct the call graph and compute the methods related to Object and i, denoted by MS.
2. Compute the PDGs of the methods in MS statically.
3. Compute the static object slice of the Object on slicing criterion <i, Object>.
4. Mark the methods related according to the static slice.
5. Analyze the PDGs, and determine the points to which breakpoints will be inserted and the
   statements which dynamic slices will be computed during program execution.
6. Insert breakpoint and the corresponding process codes to the break points.
7. Execute the program after insert breakpoints in INPUT.
8. Traverse the OPDG and obtain the dynamic object slice.
```

Alg. 2. Object slicing algorithm

the dynamic and static inter-parameter dependencies, we can reuse the static inter-parameter dependencies and the method need not to be analyzed, otherwise we should compute the dynamic inter-parameter dependencies and map them to the actual ones.

Step 5-8 is the same as step 2-5 in Alg. 1. Remove the methods and statements that are not of Object from the final result slice.

5.4. Classes Slicing

A class consists of a set of data members (attributes) and methods (operations on attributes). An object is an instance of a class. When debugging, program understanding and maintaining we would like to slice a class independently. Dynamic *class slicing* is not just the union of dynamic objects slicing. It slices not only the methods but also all the data members. To slice a class, the slicing criterion is changed to <INPUT, i, j, Class>. Informally, given a slicing criterion <INPUT, i, j, Class>, the result of dynamic class slicing is a class includes partial data members and statements in the methods of the Class, and these data members and statements might influence the variables defined at i in its j^{th} execution.

To slice a class, one method is to union all the object slices of the class and record the data members used. When the number of objects is large, this method will be too expensive. Another way is that when constructing the PDGs, we do not distinguish different objects instantiated from the same class. A statement, which uses a data member, depends on the declaration of the data member in the class. After these preprocessing, the dynamic class slicing method is similar with that of object slicing, and we do not repeat them here.

6. Applications

As an important program filter technique, dynamic slicing is often used for localizing errors when debugging. In our previous work, we have applied program dependence graph to ripple analysis [25] and test case selection in regression testing. The following will introduce how to apply dynamic slicing and dependence analysis to program debugging. A typical debugging process might include the following steps:

(1) Finding errors: Execute the program by a given input. If the result is not coincident with the respected result, we can determine that there are errors in the program. Suppose the result of variable V at statement i is error when we input INPUT.

(2) Localizing errors: Localizing errors is a trivial and time-costing work in debugging. It is difficult to find errors quickly and precisely without tools. To find errors, we often insert breakpoints into the program point concerned. The purpose is to narrow the errors to the program section before the breakpoint. The i in a slicing criterion <i, V> is a breakpoint in nature and V is the variable set interesting. But program slice only includes the statements that influence the V at i, not all the statements before i. Thus, the error codes are localized to a less program section. If the slice is executable [13], the errors are to be localized by executing the slice, not the whole program repeatedly.

(3) Modifying errors: Modify the program according to the specification.

(4) Analyzing ripple effort [25]: Program is a system with interactions among statements. The modification of an error (statement) might cause other unexpected errors. Thus when a statement is modified we must check the statements affected by the modification and modify them if necessary. The process to analyze the affected statements is called ripple analysis [25]. Such statements can be obtained from OPDG quickly and precisely by the algorithm shown in Alg. 1.

7. Conclusions

This paper proposes a dynamic slicing method for OO programs based on dependence analysis. It computes dynamic slices by combining static dependence information and dynamic execution of the program. By analyzing the control flow graph, fewer breakpoints are inserted to trace the execution of the program. It is an approach combining forward analysis with backward one. In the forward process, it marks information on the object program dependence graph (OPDG) and computes dynamic slices (they are used to record dynamic execution information) at the necessary points during the program execution. In the backward process, it traverses the OPDG marked to obtain the final dynamic slices. Based on this model we propose methods to dynamically slice methods, objects and classes. The result can be widely used in software engineering activities.

Although here we presented the approach in term of C++, other versions for this approach for other OO programming languages such as Java are easily adaptable. Now we are implementing our dynamic slicing algorithm in our “C++ programs analyzing and testing system CUTER” [9], in which it works as a filtering technique to aid bug locating during debugging.

The shortcoming of our method is that when we only slice once or few times, the cost might be too much, because all the methods are analyzed first before slicing and the results are stored in libraries on disk. In our future work, we will improve our algorithm to slice OO programs step by step, not all the methods are analyzed in advance.

8. References

[1] H. Agrawal, J. Horgan. Dynamic Program Slicing. Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 1990: 246-256

[2] H. Agrawal, et al. Debugging with Dynamic Slicing and Backtracking. Software —— Practice and Experience, 1993, 23(6): 589-616

[3] R. Chatterjee, et al. Scalable, Flow-Sensitive Type Inference for Statically Typed Object-Oriented Languages. Technical Report DCS-TR-326, Rutgers University, August 1994.

[4] Z. Chen, B. Xu, Dependency Analysis Based Dynamic Slicing for Debugging, International Software Engineering Symposium 2001 (ISES'01), Wuhan, China, 2001: 398-404

[5] Z. Chen, B. Xu. Slicing Object-Oriented Java Programs. ACM SIGPLAN Notices,2001,36(4):33-40

[6] J. Ferrante, et al, The Program Dependence Graph and Its Use in Optimization, ACM transactions on Programming Languages and Systems, 1987, 9(3): 319-349

[7] M.J. Harrold, M.L. Soffa, Efficient Computation of Interprocedural Definition-Use Chains, ACM Transactions on Programming Languages and Systems (TOPLAS), 1994, 16(2): 175-204.

[8] S. Horwitz, et al. Interprocedural Slicing Using Dependency Graphs. ACM Trans. Programming Languages and Systems, 1990,12(1): 26-60

[9] J. Guan, X. Zhou, Z. Chen, B. Xu. Design and Implementation of C++ Program Analyzing and Testing System CATER. Journal of Software, 2002, 13(Suppl.)

[10] B. Korel, S. Yalamanchili. Forward Derivation of Dynamic Slices. Proceedings of the International Symposium on Software Testing and Analysis, 1994, pp. 66-79.

[11] B. Korel. Computation of Dynamic Slices for Unstructured Programs. IEEE Transactions on Software Engineering, 1997, 2391): 17-34.

[12] B. Korel. Application of Dynamic Slicing in Program Debugging. Third International Workshop on Automated Debugging, 1997: 59-74

[13] B. Korel, J. Rilling. Dynamic Program Slicing Methods. Information and Software Technology 1998, (40): 647-659

[14] D. Liang, M.J. Harrold. Slicing Objects Using System Dependence Graphs. ICSM'98, pp. 358-367, November 1998.

[15] L. Larsen, M. J. Harrold. Slicing Object-Oriented Software. ICSE'96, pp. 495–505, Mar. 1996.

[16] D.E. Maydan, J.L Hennessy, M.S. Lam. Efficient and Exact Data Dependence Analysis, ACM SIGPLAN Notices, 1991, 26(6): 1-14.

[17] Y. Song, D. Huynh, Forward Dynamic Object-Oriented Program Slicing, Application-Specific Systems and Software Engineering and Technology (ASSET '99), IEEE CS Press, 1999: 230 –237

[18] G, Tibor, et al. An Efficient Relevant Slicing Method for Debugging, Software Engineering Notes, Software Engineering ——ESEC/FSE'99 Springer ACM SIGSFT, 1999: 303-321

[19] F. Tip. A Survey of Program Slicing Techniques, J. Programming Languages, 1995, 3(3): 121-189.

[20] P. Tonella, et al. Flow In-sensitive C++ Pointers and Polymorphism Analysis and Its Application to Slicing. In 19th International Conference on Software Engineering, pages 433–443, May 1997.

[21] M. Weiser, Program Slicing, IEEE Trans. Software Engineering, 1984, 16(5): 498-509.

[22] B. Xu. Reverse Program Dependency and Applications. Chinese J. Computers, 1993, 16(5): 385-392

[23] B. Xu, Z. Chen, X. Zhou, Slicing Object-Oriented Ada95 Programs Based on Dependence Analysis, J. Software, 2001, 12(Suppl.): 208-213

[24] B. Xu, Z. Chen. Dependence Analysis of Recursive Java Programs. ACM SIGPLAN Notices, 2001, 36 (12): 70-76

[25] H. Yang, B. Xu. Design and Implementation of a PSS/Ada Program Slicing System. Journal of Computer Research and Development, 1997, 34(3): 217-222.

[26] J. Zhao, Dynamic Slicing of Object-Oriented Programs. Technical-Report SE-98-119, Information Processing Society of Japan, May 1998: 17-23