# Dynamic Forward Slicing for Object-Oriented Programs

Zhi Cui

A Thesis

in

The Department

of

Computer Science

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

© Zhi Cui, August 2003

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisisitons et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 0-612-83898-6*
*Our file Notre référence*
*ISBN: 0-612-83898-6*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

**Canada**

# Abstract


## Dynamic Forward Slicing for Object-Oriented Programs

Zhi Cui

Program slicing, a program reduction technique, identifies codes that are related to a given function or variable of interest in a given program. It fulfills the task of decomposing and filtering a large program to restrict the focus to some specific parts. Program slicing has applications in software maintenance, reverse engineering, testing, and debugging. Program slicing can be mainly classified into static slicing and dynamic slicing. In this thesis, we introduce a new forward slicing approach for computing dynamic slices for OO programs. Our algorithm computes dynamically slices for all program components executed at run-time, without requiring any major recording of the program execution trace. We also propose an optimized algorithm as a solution to compute slices in the presents of exception handling in OO programs. The presented algorithm addresses additional issues related to the elimination of the notion of TopSlices from the based algorithm. Instead, our algorithm applies a registration strategy for slice computation, so that reduces the run-time storage overhead.



*Keywords*

Software maintenance, reverse engineering, program slicing, dynamic slicing, OO program

# Acknowledgements

I want to thank my supervisor Dr. Juergen Rilling for his supervision of this research and for his ongoing encouragement and patience. The CONCEPT (Comprehension Of Net-Centred Programs and Techniques www.cs.concordia.ca/CONCEPT) system is an excellent model and an appropriate base for the work presented in this thesis. Haixia Yu and Frederic Plouffe proofread earlier versions of this thesis and provided valuable comments and improvements. Last but not least, I wish to thank my colleagues who work at CONCEPT laboratory for our constructive discussions, and my wife Li Jin for her steady encouragement and help.

# Contents

**Abstract** .......... iii

**Figures** .......... 3

**Tables** .......... 4

**1 Introduction** .......... 5

Theory .......... 5
Applications of program slicing .......... 6
Motivation .......... 7
Contributions .......... 8
Outline .......... 9

**2 Literature Review** .......... 10

*2.1 Basic Slicing Terminology* .......... *10*

*2.2 Static Slicing* .......... *11*
2.2.1 PDG Based Approaches .......... 12
2.2.2 SDG Based Approaches .......... 15
2.2.3 Limitation of Static Slicing Techniques .......... 16

*2.3 Dynamic Slicing* .......... *17*
2.3.1 Dynamic Backward Slicing .......... 20
Korel's Approach [Kor88] .......... 21
Agrawal and Horgan's DDG Approach .......... 22
Kamkar's DDSG Approach .......... 25
2.3.2 Dynamic Forward Slicing .......... 26
Korel and Yalamanchili's Algorithm .......... 26
Csaba Faragô and Tamàs Gergely's Algorithm .......... 32
Kamkar, Fritzson, and Shahmehri's Algorithm .......... 34

*2.4 Slicing Object-Oriented Programs* .......... *34*

**3 Dynamic Forward Slicing Algorithm for Object-Oriented Programs** .......... 36

*3.1 Overview of OO specific language issues* .......... *36*
General terminology .......... 36
Variables .......... 36
Dependencies .......... 36

3.2 *Supported OO features* ... 40

3.3 *Data Structures* ... 41

3.4 *Slicing Criteria* ... 42

3.5 *Algorithm* ... 43

3.6 *Illustrating a Slice Computation* ... 48

3.7 *Comparison with Korel and Yalamanchili's Algorithm* ... 54

**4 Optimization of Dynamic Forward Slicing for OO Program Algorithm ... 57**

4.1 *Exception Handling* ... 57

4.2 *Additional Dependencies* ... 60

4.3 *Optimized Algorithm* ... 61

4.4 *Sample Computation* ... 63

**5 Experimental Analysis ... 66**

5.1 *Implementation* ... 68

5.2 *Experiment settings* ... 71

5.3 *Results from the experimental analysis* ... 72

5.3.1 Memory usage ... 72

5.3.2 Execution time ... 73

5.3.3 Precision ... 74

**6 Conclusions ... 76**

**References ... 78**

**Appendix: Additional Procedures of DFS ... 81**

# Figures

Figure 1 sample Java program ........ 13
Figure 2 partial PDG for statement 13, 14, and 15 of program in Figure 1 ........ 14
Figure 3 simplified SDG for partial statements of program in Figure 1 ........ 16
Figure 4 a sample program Mouse.java *Computer* model ........ 18
Figure 5 Korel's backward algorithm for computing dynamic slice ........ 21
Figure 6 a simple sample program illustrating a calling card timer ........ 23
Figure 7 RDDG for program in figure 6 by Agrawal and Horgan's approach ........ 24
Figure 8 Structure program marked with removable blocks ........ 27
Figure 9 Forward algorithm for dynamic slicing ........ 29
Figure 10 Sample program to show the necessity to notify changed slice of a statement ........ 38
Figure 11 Sample Java program modelling an "alarm elevator" ........ 44
Figure 12 Dynamic forward slicing algorithm for OO programs ........ 45
Figure 13 Executed parts of sample program in Figure 11 ........ 49
Figure 14 A simplified Java style program illustrating the application of exception handling ........ 60
Figure 15 dynamic forward slicing algorithm of OOP with exception handling ........ 62
Figure 16 procedures to deal with exception handling ........ 63
Figure 17 CONCEPT system architecture ........ 67
Figure 18 High-level view of the algorithm implementation ........ 68
Figure 19 Class diagram for modeling the static information ........ 69
Figure 20 Class model for representing the Expression structure ........ 70

# Tables

Table 1 Dynamic forward computing slices for program shown in figure 8, for input (N = 2, X = -4, 3) ........ 31
Table 2 Procedure of internal storage when slicing sample program in Figure 11 ........ 50
Table 3 Mappings from name to correspondences for computed variables in table 2 ........ 52
Table 4 Dynamic slices of variables to execute program in figure 11 with input of "true" ........ 52
Table 5 processing of internal storage when slicing sample program in Figure 14, given assumption that function *readFromFile()* is eventually executed ........ 64
Table 6 memory usage when slicing various execution trace of two program ........ 73
Table 7 Execution time of pattern recovering process ........ 74
Table 8 accuracy of computed slices ........ 75

# 1 Introduction

***Theory***

Mark Weiser originally introduced the notion of program slicing [Wei82]. His intend was to formally define the process of debugging. As part of his observation, he identified that the debugging consisted of analysis of dependencies from error statements back to related program subsets. According to his original definition, the notion of slicing was the procedure of removing statements from a program to identify an executable subset of statements that preserves the original behavior of the program with respect to variables of interest at a given program point.

A *program slice* [Wei82] is a set of statements and predicates of program *p* that might affect the value of a variable *v* that is defined or used at a program point. <*v, p*> is known as *slicing criterion*. Weiser's approach is based on static program dependencies, and obtained slices are consecutive sets of statements, upon which variables have data dependencies or control dependencies.

A refinement of Weiser's static slicing approach is referred to as dynamic slicing. Dynamic slicing was originally introduced by Korel [Kor88] in order to overcome some of the limitations of static slicing. *Dynamic slicing* technique emphasizes the dynamic aspects of the program execution, by considering the dependencies existing between executed program components. The dynamic slicing approaches can be classified into two different

types. One is called *dynamic backward slicing* [Kor88] [Kor90], and the other is *dynamic forward slicing* [Kor94]. *Dynamic backward slicing* requires the whole execution trace to be recorded, and then traditional program dependency analysis is applied to either program dependency graph (PDG) or system dependency graph (SDG). *Dynamic forward slicing* does not rely on the prior recording of the whole execution trace; on the contrary, slice computation for each executed variable is at run time of the program execution.

***Applications of program slicing***

Program slicing [Wei82] is a program analysis and reverse engineering technique that reduces a program to those statements that are relevant with respect to a particular slicing criterion. Originally, program slicing was introduced by Weiser [Wei82] to solve problems encountered in regression testing and debugging. Informally, a slice provides the answer to the question "What program statements potentially affect the value of a variable at a particular point of the program?" [Ste99]. To a larger extend, it is used by maintainers in the field of reverse engineering for program understanding and maintenance, e.g. coupling metrics analysis [Li01], model checking [Zha01], etc.

Some researchers apply program slicing for Ripple Effect Analysis (REA) [Wan96]. REA is an iterative process used to find out and correct errors caused from software changes. It also helps to ensure consistency and integrity of the software after corrections are made. Due to the increasing need in maintaining legacy system, which may be inconsistently or poorly documented, slicing techniques are also being used in recovering architectural documentation by recreating architectural views and abstractions of

subsystems.

Existing program slicing tools can be used to automatically compute and visualize slices of program components. The Wisconsin Program Slicing System [Wis00] is a commercially available tool for slicing C programs and is marketed by GrammaTech Inc.

***Motivation***

For applications such as testing and debugging, the computation of dynamic slice has significant advantages over static slicing. Dynamic slicing not only produces smaller and more precise slices but also considers the program behavior for a particular program execution. Current dynamic slicing approaches are mostly based on the backward analysis, requiring the recording of the whole execution trace as the necessary preprocessing. One of the major drawbacks of the backward algorithm is the space and time complexity caused by the recording of the trace and the requirement to traverse and analyze the whole recorded execution trace.

The major motivation of this research is to address this shortcoming of current dynamic backward slicing algorithms, and utilize a dynamic forward slicing approach for the computation of program slices. Furthermore, we are proposing a new forward slicing algorithm that overcomes the limitations of existing dynamic forward slicing algorithms. Current forward slicing algorithms are limited in their applicability to structural and/or procedural programs [Tip95] [Jeff01] [Lar96]. Korel and Yalamanchili's algorithm [Kor94] is limited in computing program slices only for structured, non-procedural

programs. Csaba Faragô and Tamàs Gergely [Csa02] introduced a dynamic forward approach to slice large C programs. However, few of the existing algorithms [Csa02] are addressing the specific language challenges of OO programs, such as inheritance, and polymorphism.

***Contributions***

In this paper, we present a dynamic forward slicing approach for OO programs that is based on an algorithm originally presented by Korel and Yalamanchili [Kor94]. Our algorithm is extended in the following areas:

- Extending the applicability of the original algorithm to OO program including inter-procedure slicing.
- Slicing objects and classes instead of only primitive type of variables.
- Computing precise slices for OO program components with inheritance and polymorphism.
- Providing an approach to slice OO programs in the presence of exception handling.
- Eliminating the notion of TopSlice, which generates additional space complexity.
- Introducing the notion of "Registration Strategy" that registers parental components with their children and delays the concern of control dependency, and therefore allows for the forward computation of slices for arbitrary programs.
- Improving the efficiency of the algorithm by immediate access to computed and stored program components.

### *Outline*

Subsequent chapters explore problems and advantages of existing slicing approaches, explain our proposed solution in slicing practical OO program, and present implementation issues and initial experimental analysis.

Chapter 2 describes current slicing algorithms together with their applications and limitations. The review includes the original approach where slicing is seen as a static data flow and control flow problem to the state-of-the-art where slicing is viewed as a runtime data flow and control flow problem.

Chapter 3 proposes a dynamic forward slicing algorithm that computes program slices for the major OO programming language constructs. Then, an optimized algorithm that supports slicing for Java programs with exception handling is illustrated.

Chapter 4 describes in detail design and implementation issues and analyzes experimental results.

Chapter 5 provides a summary of the contributions of this thesis and outlines some areas for future work.

# 2 Literature Review

In this section, after the introduction of basic slicing terminologies, we review the notion of program dependency graph (PDG) based and system dependency graph (SDG) based static slicing algorithms. We also survey current dynamic slicing algorithms and discuss their trade-offs.

## 2.1 Basic Slicing Terminology

In what follows, we present some of the basic slicing related terminologies and their definitions that we will adopt in the remainder of this thesis.

- *Removable Block*: corresponds to a statement, a function, a class, and a namespace within a program. More specifically, it corresponds to the smallest part of program text that can be removed without violating the syntactical correctness of the program.
- *Used variable*: variables that are evaluated or dereferenced in an expression.
- *Defined variable*: variables that are assigned or modified by value as result of a computation.
- *Data dependence*: captures a data flow from where a program component is defined or assigned to where this component is used.
- *Control dependence*: represents a control condition for which the execution of a statement or expression depends within a single method/procedure [Zha02].

- *Variable slice*: a set of relevant program components that are directly or indirectly affecting the value of this variable at a program point of interest.
- *Execution trace*: corresponds to a sequence of executed program components for a particular input. Execution traces are typically used for dynamic backward slicing to derive a dynamic program slice.
- *Expression slice*: expressions contain variables, function calls, and operators. An expression slice is a set of slices of those variables and function calls in it.
- *Statement slice*: a statement contains expressions. A statement slice is a set of slices of those participating expressions in that statement.

## 2.2 Static Slicing

Weiser [Wei82] originally introduced the notion of static program slicing – the process identifying all statements in a program $P$ that may potentially affect the value of variable $v$ at some point of $p$. A slicing criterion C can be defined as $C=(x, V)$, where $x$ is a statement in program $P$ and $V$ is a subset of the variables defined in $P$. Given program $P$, the slices are consisted of all statements in $P$ that potentially satisfy the criteria $C=(x, V)$.

In order to identify all directly and indirectly relevant statements in a program $P$ with respect to a particular slicing criterion $C=(x, V)$, static slicing algorithms typically apply an iterative approach to find all existing dependencies for a variable $v$ within a Program Dependence Graph (PDG) or System Dependence Graph (SDG). We simply classified static slicing into PDG based approaches and SDG based approaches. In what follows we

illustrate the major differences between PDG and SDG based approaches.

### 2.2.1 PDG Based Approaches

Ottenstein [Ott84] originally defined Program Dependency Graph (PDG), Horwitz et al [15] refined the original PDG. The static data dependency and static control dependency among statements in a program *P* form a PDG. Each arc in the graph represents a reachable relationship from one statement to another (*static control dependency*) or from one definition of a variable to its usage (*static data dependency*).

Static Control Dependency represents "a control condition on which the execution of a statement or expression depends on in a single method/procedure" [Zha02]. Generally speaking, statement $u$ is control dependent on statement $v$ if and only if the execution of statement $u$ is determined by the computation of statement $v$. Under this circumstance, statement $u$ is nested within statement $v$. Meanwhile, Static Data Dependency captures a data flow from where a program component is defined or assigned, to where it is used. Data dependencies are carried by variables – between its definition and its usage; specially, "there exists no other intervening definition for variable $v$ between its definition position $p$ and the evaluation position $q$" [Ril98].

```
1 package  computerSystem;
2 public class Computer {
3   private double case_in_price;
4   private double case_out_price;
5   public Computer( doublemachine, double auxiliary ) {
6      case_in_price = machine;
7      case_out_price=auxiliary;
8   }
9   public double getPrice() {
10     return case_in_price+case_in_price;
11  }
12  public static void main(String argv[]) {
13    Computer aComputer = new Computer(900, 400);
14       if( true ) {
15           System.out.println( "Price: " + aComputer.getPrice() );
16       }
17  }
18}
```

**Figure 1** sample Java program

We take the sample program from Figure 1 to illustrate these static control and data dependencies. Figure 1 illustrates a class *Computer* and one method *getPrice()*, they are tested by a set of particular assumptions. Each program statement has a unique identifier. For example, statement 15 is control dependent on statement 14 (a predicate test block), and variable *aComputer* in statement 15, has a data dependency on statement 13, where *aComputer* is defined.

*Computing a static slice using a PDG*

Parsing program source code is the first step in order to build a PDG that identifies all existing data dependencies and control dependencies in the source code. To construct a PDG, all directly reachable statements from an executable point of the program will be

included and marked if either data dependency or control dependency exists. All statements and relationships between statements are considered. Each node *x* within a PDG represents a program statement, and the outgoing edges of *x* represents the data dependencies or control dependencies other nodes have upon *x*.

In the second step, within a PDG the static slice for a variable *v* at a node *x* can be easily computed as follows: the algorithm traverses backwards all reachable edges of the PDG starting from node *x*. This process iterates until all reachable nodes are visited and included into slice of *x*. A partial PDG for statement 15 is shown below (Figure 2).

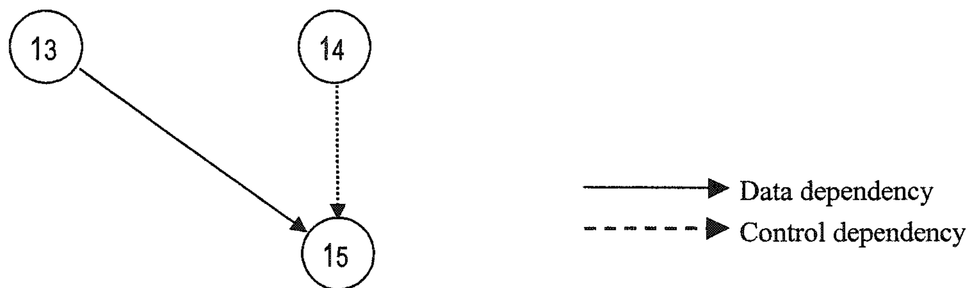



**Figure 2** partial PDG for statement 13, 14, and 15 of program in Figure 1

For example, to compute a static slice for the variable *aComputer* in the sample program shown in Figure 1, the algorithm traverses backward the simplified PDG in Figure 2 from node 15, the found nodes are node 13 and node 14 that are the elements of slices for *aComputer*. At this stage, we include only data dependencies and control dependencies for illustration purpose; statements 12, 2, and 1 are excluded to avoid unnecessary complexity, even though they are logical parental dependent components (parent function, parent class, and parent namespace).

### 2.2.2 SDG Based Approaches

Static slicing of inter-procedural programs provides additional challenges. The traditional notion of data and control flow is no longer sufficient since additional dependencies in the form of function calls have to be considered. A System Dependency Graph (SDG), which considers additionally the Procedure Dependence Graph for function calls [JFe87] [Lia98], attempts to address this problem.

Within a SDG, the data flows are represented by data dependencies; control flows are represented by control dependencies between statements or expressions. A Procedure Dependence Graph [JFe87] represents an action as a graph where vertices are functions, function calls, actual-in parameters, actual-out parameters, formal-in parameters, and formal-out parameters. Each pair of actual-in parameter and actual-out parameter corresponds an actual parameter used in the procedure call and each pair of formal-in parameter and formal-out parameter corresponds a formal parameter used in the procedure.

Figure 3 provides a simplified system dependency graph for the program shown in Figure 1. It illustrates that procedure dependencies introduce additional slices for variable *aComputer* at statement 15. By traversing this SDG, the computed slice consists of the following statements {3,4,10,12,13,14,15}. Among them statement 12 represents above mentioned parent dependency, and dependency from statement 9 to statement 15 corresponds to a procedure dependency. Variable *v* corresponding to *aComputer.getPrice()* at statement 15 is considered as *parameter_out* dependent on

variables *v'* corresponding to *case_in_price+case_in_price* in statement 10.

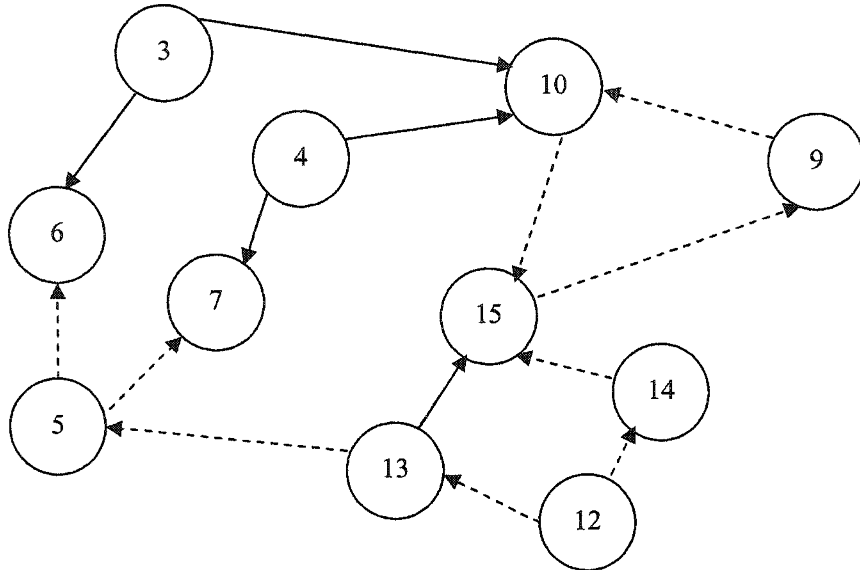



**Figure 3** simplified SDG for partial statements of program in Figure 1

### 2.2.3 Limitation of Static Slicing Techniques

The original motivation for using program slicing was to reduce the comprehension complexity during debugging, by focusing the programmer's attention only on these parts of the program that are related to a specific bug and variable of interest. However, static slicing normally constructs rather large slices due to the assumption that all reachable nodes within the PDG or SDG have to be included in the slice. In addition, slicing a well-constructed program that is typically highly cohesive might result in the whole program to be included in the slice. "The high level of cohesion results in programs where the computation of the value of each variable is highly dependent upon the values of many

other variables" [Harm01]. Other limitations of the static slicing techniques are related to different types of dynamic language constructs, e.g. dynamic binding, polymorphism and pointer handling. In most cases static slicing algorithms have to make, caused by a lack of run-time information, conservative assumption with respect to the potential program flow, and therefore the number of statements that have to be considered for the slice might be larger than necessary. In what follows we introduce dynamic slicing as an extension to static slicing that tries to overcome some of these limitations of static slicing.

## 2.3 Dynamic Slicing

Korel and Laski [Kor88] [Kor90] originally introduced dynamic slicing that considers only one particular program execution rather than all possible program executions.

A dynamic slice is the executable subset of a program, for a variable at a specific execution position; the behavior of a dynamic slice is identical to that of the original program, given the same program input. A slicing criterion of program $P$ executed on input $x$ is a duple $C=(x,y^q)$ where $y^q$ is a variable at execution position $q$. [Kor88]

Agrawal and Horgan [Agr90] proposed a similar definition for a dynamic slice: given an execution history ***EH*** of a program ***P***, the dynamic slice of ***var*** is the set of all statements in ***EH*** whose execution had effects on the value of ***var*** as it is observed at the end of the execution. The main difference between Agrawal and Horgan's algorithm and Korel's algorithm is that the former may compute non-executable slices that can't be compiled and executed afterwards, whereas the later emphasizes executable slices.

In what follows we present in Figure 4 a sample program that is used to illustrate the computation of a dynamic slice.

```
1 package  computerSystem;
2  public class Mouse {
3     private boolean left_button;
4     private boolean right_button;

5     public Mouse( boolean left_status, boolean right_status ){
6        left_button = left_staus;
7        right_button = right_status;
8     }
9     public boolean getLeft( ){
10       return left_button;
11    }
12    public boolean getRight(){
13       return left_button;
14    }

15    public static void main( String argv[] ){
16       Mouse aMouse = new Mouse( false, true );
17       String show = new String("NULL");
18       if( aMouse.getLeft() == true )
19          show  =  "RUN";
20          if( aMouse.getRight() == true )
21             show  =  "Property";
22             System.out.println( show );
23    }
24 }
```

**Figure 4** a sample program Mouse.java *Computer* model

The sample program shows a simple class called mouse, and a group of values that are used as inputs for variable *left_button* and *right_button* when an object of the type aMouse is instantiated. The program produces a wrong output at line 22 that corresponds "NULL". For a static slicing algorithm to identify the bug, a PDG of the whole program must be built, and a slice for the variable of interest, in this case the variable *show*, may be computed by

traversing a PDG. This static analysis has to consider, for example, both *if* conditions (line number 19 and 21) in the *main* method, because all possible definition have to be considered before *show* is used in the execution of line 22. This conservative approach will lead to a slice larger than necessary to reveal the bug, and therefore limits its usefulness in the application of debugging.

In practice, bugs may be revealed by executing the target program for a given set of inputs. Collection {15, 16, 5, 6, 7, 17, 18, 9, 10, 20, 12, 13, 22} is a trace for the above program after executed by the default input. Based on it, dynamic slicing could find the slice of variable *show*, inside of which bug statement 13 is included. There are two major classifications when dynamic slicing approaches are considered, *Dynamic Backward Slicing* and *Dynamic Forward Slicing*. Most of the existing methods of dynamic slicing algorithm are "backwards" analysis oriented [Kor88] [Kor90] [Dar01] [Zhan03]. They can be described as "after the execution trace of the program is recorded, the dynamic slicing algorithm traces backwards the execution trace to derive dependence relations that are then used to compute the dynamic slice" [Ril98].

In order to present both dynamic backward slicing and dynamic forward slicing approaches in detail, the following terminologies [Ril98] are to be used.

- *Action*: the execution of a statement of program ***P***.
- *Dynamic Data Dependency*: data dependency represented within the execution trace in between data elements. By Rilling's formal definition, *Dynamic Data Dependency* is

that one action assigns a value to an item of data, while another action uses this value by referencing the assigned item. [Ril98]

- *Dynamic Control Dependency*: control dependency represented by the executed elements of program ***P*** from a nested statement to a predicate test node. By Rilling's formal definition, *Dynamic Control Dependency* is that one test action determines one nested action to be sequentially executed. [Ril98]

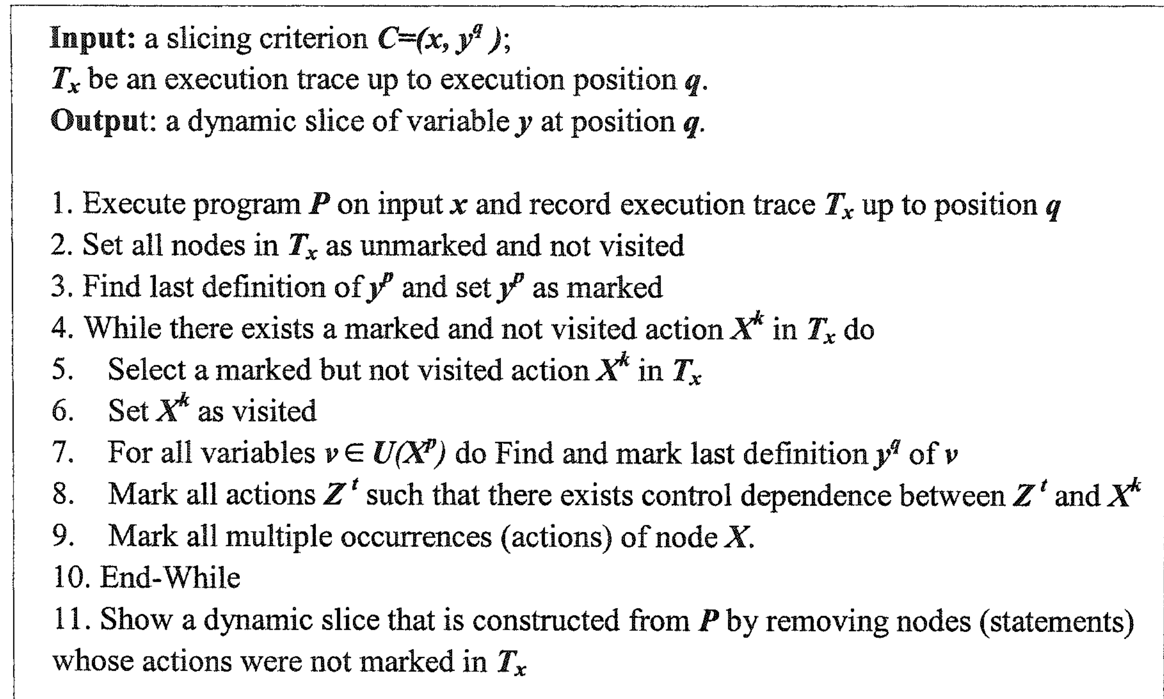
### 2.3.1 Dynamic Backward Slicing

Korel [Kor88] originally introduced the concept of *Dynamic Backward Slicing*. Informally it can be described as: given a set of inputs, a program is executed and traces of executions are recorded. For a variable $\boldsymbol{v}$ at a point of interest $q$, the dynamic backward algorithm traces back the recorded execution trace to derive the data dependence and control dependence that contribute to the computation of $\boldsymbol{v}$ at $q$.

The execution trace consists of executed statements, and used variables and defined variables in the corresponding statements. One approach to record an execution trace is to instrument the source code with monitoring statements. Every executed statement is recorded, as well as information of defined variables and used variables. In order to obtain the data and control dependencies, a syntax parsing of the source code has to take place. The parser will identify the types of statements, e.g. either a simple statement or a complex statement, and the relationship among them.

***Korel's Approach [Kor88]***

Korel's dynamic backward algorithm is described in figure 5.


**Input:** a slicing criterion $C=(x, y^q)$;
$T_x$ be an execution trace up to execution position $q$.
**Output**: a dynamic slice of variable $y$ at position $q$.

1. Execute program $P$ on input $x$ and record execution trace $T_x$ up to position $q$
2. Set all nodes in $T_x$ as unmarked and not visited
3. Find last definition of $y^p$ and set $y^p$ as marked
4. While there exists a marked and not visited action $X^k$ in $T_x$ do
5. Select a marked but not visited action $X^k$ in $T_x$
6. Set $X^k$ as visited
7. For all variables $v \in U(X^p)$ do Find and mark last definition $y^q$ of $v$
8. Mark all actions $Z^t$ such that there exists control dependence between $Z^t$ and $X^k$
9. Mark all multiple occurrences (actions) of node $X$.
10. End-While
11. Show a dynamic slice that is constructed from $P$ by removing nodes (statements) whose actions were not marked in $T_x$

**Figure 5** Korel's backward algorithm for computing dynamic slice

The following additional notations are introduced by Korel's algorithm: an execution trace $T$ is recorded and a starting position $q$ where variable $v$ is identified as $v^q$ is specified. By traversing backwards the execution trace to a position $p$ where $v$ is defined $v^p$ and this position corresponds to the only definition of $v$ between $p$ and $q$, then $v^p$ is referred to as the *Last Definition* ***LD(q)*** of $v^q$. For a statement $X$ in $T$, all variables used to be evaluated and assign a value to other data element(s) are stored into a *Used Variable Set* ***U(X)***. All variables in a statement $X$ that are declared/defined/assigned a value, are stored in another set -- *Defined Variable Set* ***D(X)***.

Korel's algorithm first executes the program and records the whole execution trace $T_x$ up to a position $q$, then initializes all nodes in $T_x$ as unmarked. Secondly, according to the slicing criterion, the algorithm finds the ***LD(q)*** and sets its status. Then, in the loop, two functions are performed: find and mark all data dependent statements; find and mark all control dependent statements. Finally, the dynamic slice for variable $y$ is displayed. The following shows the computation of slicing for the sample program in Figure 4.

Based on the default input for the sample program in Figure 4, the following execution trace {15, 2, 1, 16, 5, 6, 3, 7, 4, 17, 18, 9, 10, 20, 12, 13, 22} can be recorded after execution. Using the slicing criterion *C(show,22)*, one can identify the last definition of *show* at line 17. Next, the algorithm will identify all the variables used at line 17 and then identify their last definitions in a recursive process. The algorithm will iterate through this loop until no more data dependencies and control dependencies need to be identified. If we do not consider any other contributing aspects such as parent function, parent class, parent namespace in OO program, the final slice for variable *show* at execution position 22 will be: {17, 22} .

***Agrawal and Horgan's DDG Approach***

Agrawal and Horgan's slicing approach [Agr90] constructs a dynamic PDG from the execution trace before slicing. They refer to their extended PDG as a dynamic dependence graph (DDG).

When building the DDG, a separate node is created for each occurrence of a statement

in the execution history, with outgoing dependence edges to only those statements upon which this particular statement occurrence depends on. To minimize the number of nodes in the DDG, a new node is only created if another node with the same transitive dependencies does not yet exist. The obtained graph is called as Reduced Dynamic Dependence Graph (RDDG). "The size of the RDDG is proportional to the actual number of dynamic slices that arose during the execution and not to the length of execution." [Agr90]

After creating the RDDG, the slicing algorithm simply examines all the data and control dependencies by traversing backwards the graph. The following example illustrates the use of this algorithm:

```
1 package  computerSystem;
2    public class CallingCardTimer {
3       public static void main( String argv[] ){
4           int lefttime, offhook;
5           lefttime = atoi( argv[1] );

6           while( lefttime ){
7              offhook = Math.random()%2;
              //check with an interface to see if the call is over
8             if( offhook == 1 ) //if talk is finished or timeout
9                  break;
10            // delay one second;
11            lefttime--;
12          }
13          System.out.println( lefttime );
       }
   }
```

**Figure 6** a simple sample program illustrating a calling card timer

Suppose the input value for *lefttime* is 10 and *offhook* is set to 0, 0, and 1 respectively by the first three calls of *random()*. Then the following RDDG (shown in Figure 7) can be created from the execution trace {$3^1$, $4^2$, $5^3$, $6^4$, $7^5$, $8^6$, $10^7$, $11^8$, $6^9$, $7^{10}$, $8^{11}$, $10^{12}$, $11^{13}$, $6^{14}$, $7^{15}$, $8^{16}$, $9^{17}$, $14^{18}$}.

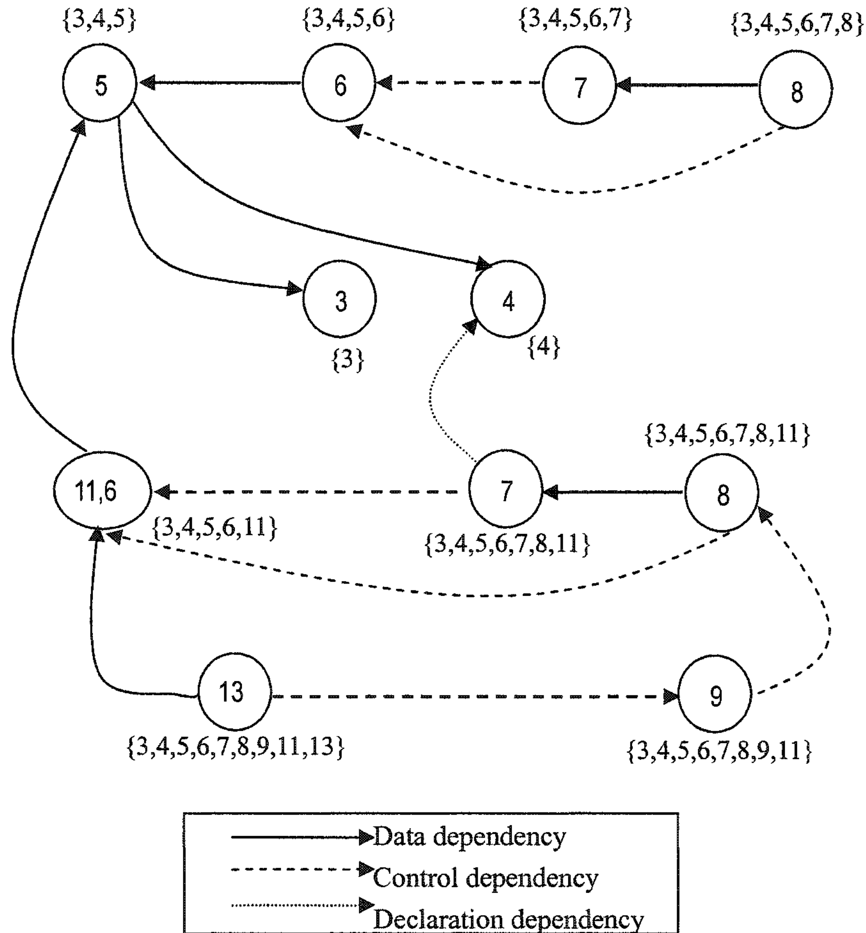



**Figure 7** RDDG for program in figure 6 by Agrawal and Horgan's approach

The numbered statements in the execution trace are presented as nodes in the above figure, each node is annotated with *ReachableStmts*, set of all statements reachable from the node. One may notice that there is only one occurrence of node 11 and two occurrences

for each node of 6, 7, and 8 in this graph, although node 11 is executed twice and 6, 7, and 8 are actually executed three times. This reduced graph is obtained by observing the principle that nodes with same transitive dependencies as existing nodes will not be created twice.

Hiralal Agrawal and Horgan's algorithm based on RDDG may reduce the cost to load a PDG in memory, with most of the benefits occurring whenever loops are executed. However, the algorithm doesn't compute executable slices and the algorithm is also limited to structural programming languages.

***Kamkar's DDSG Approach***

In [Kam93b], Kamkar presented a dynamic slicing method, which is based on the concept of graph reachability in dependence graphs. Inter-procedural slicing, which is primarily concerned with procedure level slices, was introduced. During program execution, a *Dynamic Dependence Summary Graph* is constructed, which is similar to the DDG presented by Agrawal. The algorithm requires a program to be executed and recorded to allow for the generation of an execution trace with dependency information. The slice can be computed by traversing backwards the execution trace according to dependencies. Kamkar's algorithm may also be used for slicing inter-procedural programs with the existence of procedure calls and procedures. The algorithm will not compute correct slices for OO programs or programs with unstructured language constructs (e.g. goto, exception handling).

### 2.3.2 Dynamic Forward Slicing

Although dynamic backward slicing approaches provide many benefits for program debugging, software analysis, maintenance, and abstraction, it is inevitable that the overhead for recording an entire execution trace will involve a significant space and time consumption for the computation of a program slice.

***Korel and Yalamanchili's Algorithm***

In order to address the space complexity problem of the dynamic backward slicing approaches [Kor88] [Agr90] [Kam93b], Korel and Yalamanchili [Kor94] introduced a dynamic forward approach that computes dynamic slices at run-time without the need for any major recording of execution traces.

According to Korel and Yalamanchili's [Kor94] definition, a structural program (shown in figure 8) is considered as a set of blocks. Initially, any block may be removed from source code. During dynamic forward computation, the goal is to identify these non-removable blocks that have to be included in the program slice. Korel and Yalamanchili categorized removable blocks into *simple blocks* and *complex blocks*. Assignment statements, input, output and jump statements (e.g. RETURN, BREAK) are regarded as *simple blocks*. Predicate statements (test nodes) including *IF-THEN-ELSE*, *WHILE*, *DO*, and *FOR* are considered as *complex blocks*. However, the predicate itself is not allowed to be removed from a statement due to the fact that predicate is not considered as an independent block. Blocks may be nested, such that leads to the situation when an

inner block becomes non-removable, the outer block automatically becomes non-removable and has to be therefore included in the slice.

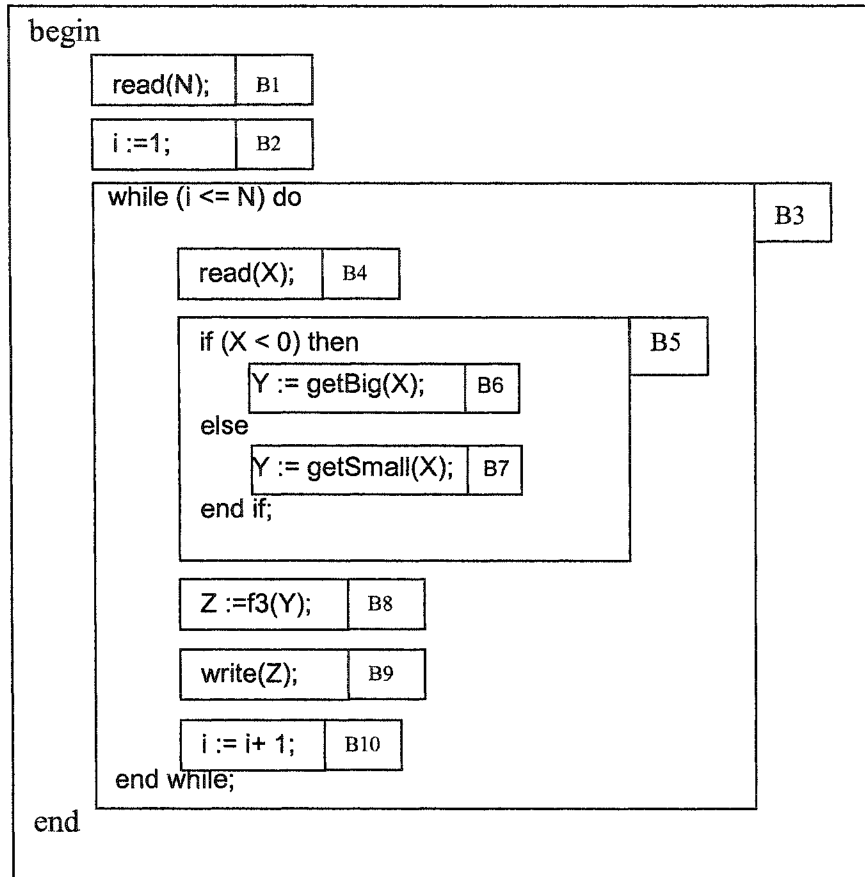



**Figure 8** structure program marked with removable blocks

In what follows we use a sample program shown in figure 8, to illustrate the notion of a block as introduced by Korel and Yalamanchili's algorithm. Blocks in the program are individually identified by boxes and associated numbers. ***B1***, ***B2***, ***B4***, ***B8***, ***B9***, and ***B10*** are simple blocks. ***B3*** and ***B5*** are complex blocks. ***B4*** is nested by ***B3*** and ***B6***, ***B7*** are nested by ***B5***.

An executed block ***B*** is not included in the slice of a variable ***v*** if any one of the

following two conditions is satisfied.

- ***B*** is a simple block and ***v*** is not defined (assigned) during the execution of ***B***.
- ***B*** is a complex block, during the execution of ***B*** and all its nested blocks, variable ***v*** is not defined (assigned), and block ***B*** does not already belong to currently computed slice of ***v*** at run-time.

Since our forward slicing algorithm is a direct extension of Korel's forward algorithm, we will provide a more detailed description of Korel's algorithm. First, we introduce the major data structures used by the algorithm.

- **B** is a block;
- **BL** is a stack of blocks containing the part of program being currently executed;
- **BV(B)** is a set of variables defined/modified during the current execution of block **B**;
- **Slice(v) = Slice(k, {v})** is a dynamic slice of variable v at the current execution position **k**;
- **NodeSlices(X)** contains a union of slices of all used variables at current action **X**;
- **TopSlice(B, v)** is the copy of dynamic slice for variable v at the entry of block **B**;
- **BlockFlag(B, v)** is a tag for variable v as to block **B**, which is marked for each block **B'** in **BL** if currently executed block already exists in **TopSlice(B', v)**;

Next, the updated algorithm is shown in figure 9. This is the refined version by Rilling [Ril98] based on Korel and Yalamanchili's dynamic forward slicing algorithm.

```
Dynamic Forward Algorithm:
Input:      a slicing criterion C= (x, n^q,{y})
Output:     a dynamic slice of variable y at position q
1.      Execute program P on input x. On entry node do:
2.          for all v ∈ V do SLICE (v):= ϕ;
3.          for all X ∈ N do NODESLICE (X):={X};
4.
5.      Action X^k : On each action X^k the following steps are performed
6.
7.      1. Action X^k
8.      a.
9.          NODESLICE(X) := NODESLICE(X) ∪ SLICE(v), v ∈ U(X^k);
10.     b.
11.         for all v ∈ V do
12.             if v ∈ D(X^k) then
13.                 SLICE(v) := NODESLICE(X);
14.                 for all B in BL do BV(B) := BV(B) ∪ {v};
15.             else
16.                 if X ∈ SLICE(v) or (X is not simple block)
17.                 then SLICE(v) := SLICE(v) ∪ NODESLICE(X);
18.             endif
19.             for all B in BL do
20.                 if X ∈ TopSlice(B,v) then
21.                 BlockFlag(B,v) := marked;
22.             endfor
23.         endfor
24.
25.     2. Entry into block B
26.         BL := Push B into BL
27.         BV(B) := ϕ;
28.         for all v ∈ V do
29.             TopSlice(B,v) := SLICE(v);
30.             BlockFlag(B,v) := unmarked;
31.         endfor
32.     3. Exit from block B
33.         BL := Pop B off BL;
34.         for all v ∈ V do
35.             if (v ∉ BV(B)) and (BlockFlag(B,v) = unmarked)
36.             then SLICE(v) := TopSlice(B,v);
37.         endfor
38.     4. k = q (execution reaches position q);
39.         Display SLICE (y)
```

**Figure 9** forward algorithm for dynamic slicing

Korel's algorithm starts with initializing all data structures, and then continues on computing program slices for all defined variables at run time. When execution arrives the entry of an action ***X*** in step 2, the undefined variables in current block and any unmarked variable corresponding to the current block make a copy of their own slice as TopSlice. When slicing an action ***X*** in step 1, algorithm registers all defined/modified variables and computes/updates slice for each of them in step ***1a***. If current block ***B*** is a complex block, then ***B*** is temporarily included as slice to all variables except those currently defined in step ***1b***. Until execution arrives the exit of block ***B*** in step 3, this assumption may not be clarified true if ***B*** has been included into slices of ***v***; otherwise, ***B*** is removed from slice of ***v*** (copy back ***v***'s TopSlice). The algorithm terminates when the execution comes to an end $n^q$, which is a defined element in the slicing criteria $C = (x, n^q, \{y\})$.

Table 1 illustrates the procedure that Korel's algorithm computes forward slices for the program shown in figure 8. The program is executed given input ***(N = 3, X = -4, 3, -2)***.

**Table 1** dynamic forward computing slices for program shown in figure 8, for input (N = 2, X = -4, 3)

| Execution trace | Slices of variables | | | | |
|---|---|---|---|---|---|
| | N | I | X | Y | Z |
| $0^0$ | {} | {} | {} | {} | {} |
| $1^1$ | {1} | {} | {} | {} | {} |
| $2^2$ | {1} | {2} | {} | {} | {} |
| $3^3$ | {1,2,3} | {1,2,3} | {1,2,3} | {1,2,3} | {1,2,3} |
| $4^4$ | {1,2,3} | {1,2,3} | {1,2,3,4} | {1,2,3} | {1,2,3} |
| $5^5$ | {1,2,3,4,5} | {1,2,3,4,5} | {1,2,3,4,5} | {1,2,3,4,5} | {1,2,3,4,5} |
| $6^6$ | {1,2,3,4,5} | {1,2,3,4,5} | {1,2,3,4,5} | {1,2,3,4,5,6} | {1,2,3,4,5} |
| $8^7$ | {1,2,3} | {1,2,3} | {1,2,3,4} | {1,2,3,4,5,6} | {1,2,3,4,5,6,8} |
| $9^8$ | {1,2,3} | {1,2,3} | {1,2,3,4} | {1,2,3,4,5,6} | {1,2,3,4,5,6,8} |
| $10^9$ | {1,2,3} | {1,2,3,10} | {1,2,3,4} | {1,2,3,4,5,6} | {1,2,3,4,5,6,8} |
| $3^{10}$ | {1,2,3,10} | {1,2,3,10} | {1,2,3,4,10} | {1,2,3,4,5,6,10} | {1,2,3,4,5,6,8,10} |
| $4^{11}$ | {1,2,3,10} | {1,2,3,10} | {1,2,3,4,10} | {1,2,3,4,5,6,10} | {1,2,3,4,5,6,8,10} |
| $5^{12}$ | {1,2,3,4,5,10} | {1,2,3,4,5,10} | {1,2,3,4,5,10} | {1,2,3,4,5,6,10} | {1,2,3,4,5,6,8,10} |
| $7^{13}$ | {1,2,3, 4,5,10} | {1,2,3, 4,5,10} | {1,2,3,4,5,10} | {1,2,3,4,5,6,7,10} | {1,2,3,4,5,6,8,10} |
| $8^{14}$ | {1,2,3,10} | {1,2,3,10} | {1,2,3,4,10} | {1,2,3,4,5,6,7,10} | {1,2,3,4,5,6,7,8,10} |
| $9^{15}$ | {1,2,3,10} | {1,2,3,10} | {1,2,3,4,10} | {1,2,3,4,5,6,7,10} | {1,2,3,4,5,6,7,8,10} |
| $10^{16}$ | {1,2,3,10} | {1,2,3,10} | {1,2,3,4,10} | {1,2,3,4,5,6,7,10} | {1,2,3,4,5,6,7,8,10} |
| $3^{17}$ | {1} | {1,2,3,10} | {1,2,3,4,10} | {1,2,3,4,5,6,7,10} | {1,2,3,4,5,6,7,8,10} |

Compared with the dynamic backward slicing approach, the forward approach does not need a dependence graph. The backward approach must record the whole execution trace, which may result in an unbounded space complexity. The dynamic forward slicing algorithm on the other hand does not require the recording of the execution trace. An additional benefit of the dynamic forward algorithm is that it computes dynamic slices for all variables defined during the particular program execution, rather than only for a specific variable v.

However, it should be noted that Korel and Yalamanchili's algorithm has also

limitations:

- The algorithm is unable to compute a slice for a variable that is defined inside a complex block. When the execution arrives at the entry of a complex block, variables defined within its scope are not known yet. Such variables may never have a chance to include its parent block, referred as control dependence, to be slice. An assumption made by the algorithm is that all-variable set have to be known before the program execution.
- An important overhead is associated with storing the TopSlices. In particular, for large software system with deep nesting structures, the space complexity for temporarily storing these TopSlices can become significant.
- The algorithm is limited to compute forward slices for structural program.

***Csaba Faragô and Tamàs Gergely's Algorithm***

In [Csa02], Csaba Faragô, Tamàs Gergely introduced a forward approach to slice large C program. They considered the existence of pointer and jump statements. This algorithm is a dynamic forward approach, computing program slices in parallel with the program execution. Compared with Korel and Yalamanchili's algorithm, Faragô and Gergely's algorithm may be used to slice larger real world C programs.

This algorithm is based on the concept **D/U** items, which means that in a program instruction the defined variables (scalar variable, predicate variable, output variable, dereference variable, function call argument variable, function call return variable) in **D** are

all dependent on a set of **U**. Not only used variables are included in **U**, like ***U(X)*** defined in Korel and Yalamanchili's algorithm, but also the parent statement (control dependence) is included into **U**. **LS(d)** records the current executed statement **s** where variable **d** is defined or statement **s** occurs. Slice of **d** is determined by a equation: *DynSlice*(**d**) = *DynSlice*(**U**) + **LS(d)**.

What is more important is that Faragô and Gergely's algorithm handles pointers and jump statements in a C program. During the execution some runtime info, such as memory addresses of scalar variables, actual values of pointers, beginnings/endings of functions, etc, is to be extracted. (That technical procedure for the extraction of the addresses is called program instrumentation, which means modifying the program code in such a way that the program retains its original behavior.) Having the memory address, then, a pointer variable and its points-to data may be dealt with universally and explicitly. For example, a pointer dereference *p, the precise dependence for *p includes the pointer itself and the dereferenced memory location (points-to data) as well. Other dynamic language constructs, like arrays are also benefiting from this additional run-time information. This allows the dynamic slicing algorithms to compute often more precise and smaller slices than the static approaches.

Where a jumping statement occurs, the **D/U** structure is built up as follows. First, a "jumping-command-named variables" ***j*** is introduced as defined. ***j*** will be inserted into the use set (**U**) of the statements that occurs after the corresponding label within the function.

However, the obtained slice may not be precise because the algorithm has to make rather conservative assumptions with respect to the jump statements.

***Kamkar, Fritzson, and Shahmehri's Algorithm***

In [Kam93a], Kamkar, Fritzson, and Shahmehri presented approaches to dynamically slice procedural program. During execution, a *summary dependence* (dynamic) *graph* is constructed. Within a SDG, vertices are referred as *procedure instances*, i.e.: procedure activation annotated with their parameters. The edge is either an activation edge (procedure call) or summary dependence (transitive data and control dependence between input and output parameters of the procedure instances).

A slicing criterion is defined as a pair consisting of a procedure instances, and input or output parameters of the associated procedure. Having a summary graph, a slice can be obtained: the parts of summary graph from which dependent nodes can be reached with respect to slicing criterion.

## 2.4 Slicing Object-Oriented Programs

The motivation of program slicing was originally used to analyze structural language such as Pascal. Most of the existing research in program slicing has focused on the computation of program slices for procedural languages, including extensions with respect to jump statements [Csa02] and unstructured programs [Kor97]. Over the last few years, Object-Oriented program languages such as C++ and Java have become of general interest

due to their extraordinary usage in software development. Researchers in the program slicing area addressed the trend towards OO programming languages by introducing program slicing algorithms that are capable of computing slices for OO programs [Zha01] [Bat99].

Interaction among procedures is unavoidably encountered context when we consider an OO program.

*Intra-procedural slicing* computes slices within one procedure. Calls to other procedures may be handled either by static inter-procedural slicing or dynamic inter-procedural slicing. In computing slices for procedural programs, *Static interprocedural slicing* is commonly used based on an SDG [Hor90] [Lia98] [Har98]. In the SDG, the algorithms have to consider all possible call paths to guarantee the correctness of the computed slice. This conservative approach will often lead to program slices larger than necessary. *Dynamic interprocedural slicing* on the other hand can handle data dependence between actual and formal parameters using definition and use of memory locations. The dynamic approaches have the advantage of resolving some of the variable analysis (by value, by reference) by using run-time information.

Besides inter-procedure function calls, object oriented programs result in additional challenges with respect to inheritance, polymorphism, dynamic binding, and the difficulty in slicing objects, classes, and packages.

# 3 Dynamic Forward Slicing Algorithm for Object-Oriented Programs

The purpose of this research is to develop and implement an efficient dynamic forward slicing algorithm for Object-Oriented programs that is applicable for programming languages like C++ or Java. This chapter discusses about the details of the algorithm, as well as some of the implementation issues.

## 3.1 Overview of OO specific language issues

***General terminology***

In Object-Oriental programs, a class is a logical unit of data providing various functions operating on the data. A class is also a schema or a representation of the behaviour of the data. A class can also inherit properties of other classes. Functions are used to exchange messages. Finally, an object is simply an instance of a class.

***Variables***

In an OO program, a variable may be a local variable, a global variable, an object, an attribute of an object, a class variable (e.g. public static in Java class), a function argument variable, or even a class. In order to slice a variable, dependencies existing between OO program components must be clarified.

***Dependencies***

OO programs include similar *Data dependence* and *control dependence* as traditional *non-OO programming* language. Data dependency represents the data flow as part of an

intra-function or inter-function. Control dependency represents the control logic on which the execution of a statement depends within a function. Informally, a defined variable ***v*** in statement ***s1*** may be data dependent on the last definition and may be control dependent on statement ***s0*** which nests ***s1***.

*Declaration dependency* refers to the usage of a variable depends on its declaration. In OO program, any variable must be declared before it is used.

*Type dependency* is a basic dependency. In OO programs, complex data types may correspond to a definition of a structure, a union, a record, a class, or an interface. Any complex variable type has a dependency on its definition. In the context of our algorithm we refer to this dependency as a *type dependency*.

*Ownership dependency* represents the relationship between one of the following language constructs:

- An attribute variable and an object (any data member is dependent on an object);
- A statement and a function that contains the statement;
- A function and a class (class owns functions/methods);
- A class and a namespace (a class is dependent on a namespace).

*Dynamic deliverable dependency* exists between a statement and defined variables within the statement's scope. If a defined variable *v* has already an existing dependency on statement *s* (either a simple or a complex) and the slice for that statement has been changed due to re-computation, then the changed slices must be transformed to *v* if v in case *v* would

not have a chance to re-contain slice of *s*. Actually, this is the extension of control dependence handled by dynamic slicing to ensure a consistent behavior of the original program after being updated. The following sample program will illustrate the situation.

```
public static avoid main( String argv[] ){
1   int i = 0;
2   while( i<2 ){
3       if( i==0 ){
4           int k = 0;
        }
5       i++;
    }
}
```

**Figure 10** sample program to show the necessity to notify changed slice of a statement

In our algorithm we are no longer utilizing Korel's notion TopSlice; instead, slices of defined variables are to be computed when they are executed actually. Therefore, during the execution of the sample program in Figure 10, the following execution trace can be observed: $\{1^1,2^2,3^3,4^4,5^5,2^6,3^7,5^8,2^9\}$, and a slice can be computed consisting of statements 1, 2, 3, and 4, respecting the slicing criterion $C(k, 2^9)$. However, it can be observed that this slice is not correct, because statement 5 is not included in the slice and slice is inconsistent with the behavior of original program. A more detailed examination of the situation reveals this problem. After the first execution of the "while" loop, slice - {1,2} is computed for block 2, and after the second and third execution of the loop, block 2 has slice - {1,2,5}. Because statement 4 is only executed once, recomputed slice of block 2 is unknown to

variable $k$ although $k$ is defined within the scope of statement 2. Alternately, we let block/statement 2 notify its changed slice to $k$.

*function-call* dependency, *parameter-in* dependency, and *parameter-out* dependency. To illustrate the above three dependencies, a sample program is given:

```
void ff( ){
   y = f( a, b );
}

int f( int in, int out ){
   int value = 1;
   return value;
}
```

The *function-call* dependency exists between function *f()* and function call *y=f(a,b)*. The formal parameter *in* depends on the evaluation of actual parameter *a*, this scenario corresponds to a *parameter-in* dependency. The actual parameter *b* depends on the evaluation of formal parameter *out*; this scenario corresponds to a *parameter-out* dependency. The evaluation of variable *y* at call site depends on the value returned by last statement in the exit of function *f*; this scenario corresponds to a *parameter-out* dependency.

*Call-site dependency*: for a defined variable *v* within a function *f()*, beside the function-call dependency that is inherited from the function directly, *v* also depends on statement *s* which invokes the function call *f()*. If the slice of *s* is re-computed, e.g. caused by multiple function calls, *v* has to be notified the modified slice of *s*.

*Inheritance dependency*: represents the inheritance relationship between classes. If a slice includes the identification of subclasses, it must also includes the base classes to guarantee a semantically correct program.

*Function/method to function/method dependency*: a method implementation depends on the existence of the corresponding pure virtual method in a base/abstract class. To address *inheritance dependency* and *method-to-method dependency* in a dynamic slicing approach, we utilize static information from a preprocessing parsing step. The parser is capable to provide a list of existing inheritance and method-to-method dependencies that can be identified by parsing the source code.

## 3.2 Supported OO features

The following checklist [Ste99] for object oriented programming languages was used as a guideline to identify OO features that should be supported by a slicing algorithm for OO programs.

- The slicing algorithm should support the entire language syntax, which includes user-declared data types, structured types (classes and arrays), local/global variables, argument variables, object/class variables, function calls and functions with external effects, nested statements, and procedure variables.
- Slicing object-oriented features such as inheritance, dynamic binding and polymorphism has to be fully supported by the algorithm. Object-oriented programs make heavy use of dynamic binding, both of which are traditionally

handled imprecisely by static analysis.

- The computation of the slices should be fast, but the resulting slices should still be as precise as possible.
- Information that has already been computed for a component should be able to be reused in order to improve efficiency.
- An executable slice has priority over a smaller slice. Executable slices allow for easier validation of the correctness of the slices (the execution of the slice and the execution of the original program have to produce the same behavior with respect to the slicing criterion). In addition, our final program algorithm should scale to support the slicing of larger systems.

## 3.3 Data Structures

The following is a list of data structures used by our dynamic forward algorithm.

*NS*: Namespace, a set for namespaces and their corresponding slices.

*CL*: Class, a set for all classes and their corresponding slices. Every class belongs to a parent namespace.

*TYPE*: Variable type, a set for all encountered types and their corresponding slices.

*FUN*: Function, a set for all functions and their corresponding slices. Every function may have a parent class.

*FUNC*: Function call, a stack to store function calls that are currently executed. Each function call has its returning (parameter-out dependence), actual parameter

(expression) list. Any function call may be an element of an expression. When the execution exits from a function, the associated function call must be popped up from FUNC.

*ST*: Statement, a stack of statements (within the scope of currently executed block) and their corresponding slices. When execution leaves the scope of a statement *s*, *s* must be popped up.

*EXP*: Expression, consists of constants, variables, and function calls; its creation is "on the fly".

*VAR*: A set that dynamically records all defined variables during the program execution and their corresponding slices. A variable may be a local variable, a global variable, an object, a member variable of an object, a pointer variable, a point-to variable, a member variable of a class, and a function call argument variable.

AST: Abstract syntax tree, expressions may be built by accessing the static AST.

We use a few containers to store those computed static program components in case of reuse. Once the slicing requires computation of a particular component, the algorithm will try to search for that particular component and try to reuse it. However, the slice of an existing statement will not be reused and that must be recomputed.

## 3.4 Slicing Criteria

We defined the dynamic forward slicing criteria as **(x, $n^q$, V, C)** where **x** denotes the input

to run the program $\boldsymbol{p}$, and $\mathbf{n}^{\mathbf{q}}$ is an action, **V** is the set of the variables for which the dynamic dependences should be computed, and **C** is the set of program components with their corresponding slices.

## 3.5 Algorithm

Some basic concepts and notations have already been introduced in the background section prior to the description of our algorithm. For clarity, most of the concepts used in our algorithm are based on Korel's definition [Kor94], but in some cases modifications and extensions have been made. We demonstrate our algorithm by applying it on the sample program shown in figure 11.

```
0      package elevator;
1      class Elevator {
       public:
2          Elevator(int top_floor){
3          current_floor = 1;
4          current_direction = 0;  //up
5          this.top_floor = top_floor;
           }
6        void up(){
7          current_direction = 0;
           }
8          void down(){
9          current_direction = 1;
           }
10         int which_floor(){
11         return current_floor;}
12         int direction()
13         return current_direction; }
14         void go(int f){
15            if (current_direction == 0){
16            while ((current_floor != f)&& (current_floor <= top_floor))
17              add(current_floor, 1);}
             else{
18              while ((current_floor != f)&& (current_floor > 0))
19              add(current_floor, -1); } };
        private:
20         void add(int a, const int b){
21         a = a + b;};
        protected:
22         int current_floor;
23         int current_direction;
24         int top_floor; };
25      class AlarmElevator extends Elevator {
        public:
26        AlarmElevator(int top_floor){
27             super(top_floor);
28             alarm_on = 0;}
29         void set_alarm(){
30             alarm_on = 1; }
31         void reset_alarm(){
32             alarm_on = 0; }
33         void go(int floor){
34             if (!alarm_on)
35                super.go(floor); }
        protected:
36            int alarm_on;
37            public static void main(String[] argv) {
38                Elevator elevator = null;
39                if (argv[1])
40              elevator = new AlarmElevator(10);
                  else
41              elevator = new Elevator(10);
42                elevator.go(3);
43                System.out.println("Current floor:"+elevator.which_floor());
            }
       }
```

**Figure 11** sample Java program modelling an “alarm elevator”

**Program dynamicForwardSlicingAlgorithmofOOP**

Initialize set of NS, CL, ST, TYPE, FUN, FUNC, and VAR. Source code parsed and stored in repository
Collect declared global variables, and compute their declare slices, and put them into VAR.
Execute program *P* on input *x*. On each execution of statement *s* the following steps are performed.

```
1   switch(type of s) {
2     case entry into a function:
3       if( !this function is already in FUN ){
4          create this function f;
5          compute "declare-slice" for it and insert it into FUN;
6       }
7       functionProcess ( f );

8     case return from a function:
9       compute ret-slice from expressions of s;
10      function-call = FUNC.pop( );
11      slice of function-call ∪= above ret-slice;

12    otherwise: s=get statement currently executed from ST; if s==null go 13 else go 14;
13      s = createStatement ( );
14      s.depth-degree=getDegree ( );
15      while(s.depth-degree <= ps.depth-degree && they share common parent function ||
              they have different parent functions && s is not beginning to implement a function){
16        parent-statement ps = ST.pop( );
17        if( ps is a complex statement ) {
18           computeStatementSlice( ps );
19           if( ps's new slice set differs from its old )
20              notify all registered variables latest slice of ps;
21        }
22      }
23      computeStatementSlice ( s );

24      push s and its slice (dependence) into ST;
25      for( each defined or declared variable v in s ){
26         variableComputing ( v, s );
27         put v into VAR;
28         register v with all statements within ST;
29      }
    }
30  for each v in VAR{  //output
31     print out both declare and dynamic slices of v;
32  }
```

**Figure 12** dynamic forward slicing algorithm for OO programs

Our algorithm starts with initializing the various data stores, including the storages for namespaces, classes, functions, function calls, statements, and variables. Declared global variables are identified by analyzing the parsed source code. Then, the algorithm starts analyzing each executed statement (depending on its type) at run-time.

If the executed statement corresponds to an entry of a function implementation, then the algorithm stores the function information into the *FUNC* container (lines 2 to 7). The relationship between formal parameters and actual parameter is also identified at this time (*parameter-in* dependency). However, in this algorithm, we assume that any defined variable in a function is directly control dependent on its associated function call, which is directly control dependent on the associated function call. After computing the slices for each formal parameter variable, algorithm stores these variables in the *VAR* container.

If a return statement within a function is executed, the slice of this return statement is computed and assigned to the caller of the function where this return statement exits. The function call can then be popped from the function call stack, hence it is no longer considered for further computation (lines 8 to 11).

If the current executed statement is either a simple statement (e.g. an assignment, a standalone function call) or a complex statement (e.g. an IF-ELSE, a WHILE, a FOR), the algorithm first analyzes the relationship between the currently executed statements and the last executed statement. After the slice of last executed statement is re-computed, all the variables within its scope have to be notified about the change of its slice. Therefore, any

newly defined/modified variable at the current executing position must be registered with its parent statements (complex statement) in advance (lines 14 to 22).

Slice of current statement is computed (line 23) by utilizing the static information from the parsed AST. Control dependency is contributed by parent statement. Data dependency is contributed by the expressions current statement contains. For example, a simple statement *x=f1(*y)+f2(i++)* may host complex expressions including more than one function calls. The slice for variable *x* in the example can only be computed correctly once the execution exits from this statement, i.e. execution of function call *f1* and *f2* has been completed. After the slice computation for all defined variables, any registered variables will be notified the final slice of this statement. That is also the reason why we may register a defined variable within a function *f* not only with its parent statement (line 28), but also with the statement invoking *f* in need to update all existing control dependencies. Therefore, the obtained slice is precise because it considers all the existing data dependencies and control dependencies. Finally, the current statement and its associated slice are stored on the stack.

Although the semantic restriction and dependence on a particular expression exist, the processing sequence for the defined variables might vary. Any expression is a non-removable part of a statement and can only be removed if the whole statement is removable. In order to analyze each individual expression, information about the used variables and function calls are accessed from static information in the AST.

When computing the slice for a defined/modified variable (line 26), the algorithm handles objects and function argument variables respectively. The definition/modification of an object corresponds to the definition/modification of its data member(s). The algorithm re-computes the slice for each object data member if the object is defined or modified. In the presented approach all data member variables are registered with the corresponding object. For the same purpose, an actual parameter variable in the function call is registered with an argument variable once the execution arrives at the entry of the associated function. During the execution, if an argument is defined/modified within the function, the actual parameter variable, which has been registered with the argument variable, is notified the results of the argument slice.

Finally, each of newly defined/modified variables is (including data members of an object) pushed onto the *VAR* container (line 27). When a variable and its slice are displayed during the program execution (line 30 to 32), the slice from both declaration and definition are counted in. When the algorithm displays the slice of an object variable, all the slices for its data member are included.

## 3.6 Illustrating a Slice Computation

In what follows we use the sample program in figure 11 to illustrate the dynamic slice computation for a small OO program. The program is executed given input: argv[1] as "true". Figure 13 shows the corresponding execution statements for the program.

```
0   package elevator;
1   class Elevator {

2        Elevator(int top_floor){
3             current_floor = 1;
4             current_direction = 0;  //up
5             this.top_floor = top_floor;
         }

10       int which_floor(){
11            return current_floor;
         }

14       void go(int f){
15            if (current_direction == 0){
16                 while ((current_floor != f)&& (current_floor <= top_floor))
17                      add(current_floor, 1);
         }

20       void add(int a, const int b){
21            a = a + b;
         };

22       int current_floor;
23       int current_direction;
24       int top_floor;
    };

25  class AlarmElevator extends Elevator {
26       AlarmElevator(int top_floor){
27            super(top_floor);
28            alarm_on = 0;
         }

33       void go(int floor){
34            if (!alarm_on)
35                 super.go(floor);
         }

36       int alarm_on;


37       public static void main(String[] argv) {
38            Elevator elevator = null;
39            if (argv[1])
40                 elevator = new AlarmElevator(10);
42            elevator.go(3);
43            System.out.println("Current floor:"+elevator.which_floor());
         }
    }
```

**Figure 13** executed parts of sample program in Figure 11

**Table 2** procedure of internal storage when slicing sample program in Figure 11

| *trace** | *NS** | *CL** | *FUNC** | *FUN** | *ST** | *VAR** |
|---|---|---|---|---|---|---|
| $37^{1}$ | elevator | Elevator<br>AlarmElevator | main() | main() | | **100**; |
| $38^{2}$ | ... | ... | ... | ... | 38 | 100; **200**; |
| $39^{3}$ | ... | ... | ... | ... | 39 | 100; 200; |
| $40^{4}$ | ... | ... | AlarmElevator()<br>main() | ... | 40,39 | 100; **200**; **300**; **310**; **320**; **330**; |
| $26^{5}$ | ... | ... | ... | main()<br>AlarmElevator() | 39 | 100; 200; 300; 310; 320; 330; **400**; |
| $27^{6}$ | ... | ... | super()<br>AlarmElevator()<br>main() | ... | 27,39 | 100; 200; 300; 310; 320; 330; 400; |
| $2^{7}$ | ... | ... | ... | main()<br>AlarmElevator()<br>Elevator() | 39 | 100; 200; 300; 310; 320; 330; 400; **500**; |
| $3^{8}$ | ... | ... | ... | ... | 3,39 | 100; 200; 300; **310**; 320; 330; 400; 500; |
| $4^{9}$ | ... | ... | ... | ... | 4,39 | 100; 200; 300; 310; **320**; 330; 400; 500; |
| $5^{10}$ | ... | ... | ... | ... | 5,39 | 100; 200; 300; 310; 320; **330**; 400; 500; |
| $28^{11}$ | ... | ... | AlarmElevator()<br>main() | ... | 28,39 | 100; 200; **300**; 310; 320; 330; 400; 500; |
| $42^{12}$ | ... | ... | elevator.go()<br>main() | ... | 42 | 100; 200; 300; 310; 320; 330; 400; 500; |
| $33^{13}$ | ... | ... | ... | main()<br>AlarmElevator()<br>Elevator()<br>go() | | 100; 200; 300; 310; 320; 330; 400; 500; **600**; |
| $34^{14}$ | ... | ... | ... | ... | 34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; |
| $35^{15}$ | ... | ... | super.go()<br>elevator.go()<br>main() | ... | 35,34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; |
| $14^{16}$ | ... | ... | ... | main()<br>AlarmElevator()<br>Elevator()<br>go()<br>go() | 34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; **700**; |
| $15^{17}$ | ... | ... | ... | ... | 15,34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; |
| $16^{18}$ | ... | ... | ... | ... | 16,15,<br>34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; |
| $17^{19}$ | ... | ... | add()<br>super.go()<br>elevator.go()<br>main() | ... | 17,16,<br>15,34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; |
| $20^{20}$ | ... | ... | ... | main()<br>AlarmElevator()<br>Elevator()<br>go()<br>go()<br>add() | 16,15,<br>34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700;<br>**800**; **900**; |
| $21^{21}$ | ... | ... | ... | ... | 21,16,<br>15,34 | 100; 200; 300; **310**; 320; 330; 400; 500; 600; 700;<br>800; 900; |
| $16^{22}$ | ... | ... | super.go()<br>elevator.go()<br>main() | ... | 16,15,<br>34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700;<br>800; 900; |
| $17^{23}$ | ... | ... | add()<br>super.go()<br>elevator.go()<br>main() | ... | 17,16,<br>15,34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700;<br>800; 900; |
| $20^{24}$ | ... | ... | ... | ... | 16,15,<br>34 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700;<br>**800**; **900**; |
| $21^{25}$ | ... | ... | ... | ... | 21,16,<br>15,34 | 100; 200; 300; **310**; 320; 330; 400; 500; 600; 700;<br>800; 900; |
| $16^{26}$ | ... | ... | super.go() | ... | 16,15, | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; |

| | | | elevator.go()<br>main() | | 34 | 800; 900; |
|---|---|---|---|---|---|---|
| $43^{27}$ | ... | ... | elevator.which_floor()<br>main() | ... | 43 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; 800; 900; |
| $10^{28}$ | ... | ... | ... | main()<br>AlarmElevator()<br>Elevator()<br>go()<br>go()<br>add()<br>which_floor() | | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; 800; 900; |
| $11^{29}$ | ... | ... | ... | ... | 11 | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; 800; 900; |
| end | ... | ... | | ... | | 100; 200; 300; 310; 320; 330; 400; 500; 600; 700; 800; 900; |

* *NS*: set of namespaces, *CL*: set of classes, *FUNC*: stack of function calls, *FUN*: set of functions, *ST*: stack of statements, *VAR*: set of variables

In order to illustrate the dynamic process of the slice computation, we listed *NS*, *CL*, *FUNC*, *FUN*, *ST*, and *VAR*. The far left column in table 2 lists the execution trace for the given input $\{37^{1}, 38^{2}, 39^{3}, 40^{4}, 26^{5}, 27^{6}, 2^{7}, 3^{8}, 4^{9}, 5^{10}, 28^{11}, 42^{12}, 33^{13}, 34^{14}, 35^{15}, 14^{16}, 15^{17}, 16^{18}, 17^{19}, 20^{20}, 21^{21}, 16^{22}, 17^{23}, 20^{24}, 21^{25}, 16^{26}, 43^{27}, 10^{28}, 11^{29}\}$. The contents of *NS*, *CL*, *FUNC*, and *FUN* illustrate "the dynamics" of the slice computation process. One can observe how the different storage and their content are updated, depending on the execution position, the statement type, and the influence of particular variables at the current execution position. We used a bold typeface in table 2 to indicate places when a variable or object definition/declaration took place at runtime. Mappings from variables to their corresponding internal representations are listed in table 3.

It should be noted that the function *main()* and its caller appeared at the same row 1 in table 2, caused by the fact that the main function is never visually called within the program and we do not have any caller information. A similar scenario exists for *system function*(s). Corresponding function implementation of called *system function* can't be

found from the source code. Therefore, *system function* calls are not stored on the *FUNC* stack.

A blank cell in a table corresponds to an empty storage at the current execution position.

**Table 3** mappings from name to correspondences for computed variables in table 2

| Three Digits Representation | Variable Name | Within which object | Within which function defined | Within which class |
|---|---|---|---|---|
| 100 | argv[] | | main() | AlarmElevator |
| 200 | elevator | | main() | AlarmElevator |
| 300 | alarm_on | elevator | | AlarmElevator |
| 310 | current_floor | elevator | | Elevator |
| 320 | current_direction | elevator | | Elevator |
| 330 | top_floor | elevator | | Elevator |
| 400 | top | elevator | AlarmElevator(int top) | AlarmElevator |
| 500 | ceiling | elevator | Elevator(int ceiling) | Elevator |
| 600 | floor | elevator | go() | AlarmElevator |
| 700 | f | elevator | go() | Elevator |
| 800 | a | elevator | add() | Elevator |
| 900 | b | elevator | add() | Elevator |

**Table 4** dynamic slices of variables to execute program in figure 11 with input of "true"

| trace | Slice | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 100 | 200 | 300 | 310 | 320 | 330 | 400 | 500 | 600 | 700 | 800 | 900 |
| $37^1$ | 37,25,1,0 | | | | | | | | | | | |
| $38^2$ | ... | 38,37,25,1,0 | | | | | | | | | | |
| $39^3$ | ... | ... | | | | | | | | | | |
| $40^4$ | ... | 40,39,38,37,25,1,0 | 40,39,37,25,1,0,36,38 | 40,39,37,25,1,0,22,38 | 40,39,37,25,1,0,23,38 | 40,39,37,25,1,0,24,38 | | | | | | |
| $26^5$ | ... | ... | ... | ... | ... | ... | 26,25,1,0,40,39,37 | | | | | |
| $27^6$ | ... | ... | ... | ... | ... | ... | ... | | | | | |
| $2^7$ | ... | ... | ... | ... | ... | ... | ... | 2,1,0,27,26,25,40,39,37 | | | | |
| $3^8$ | ... | ... | ... | 40,39,37,25,1,0,22,38,3,2,27,26,37 | ... | ... | ... | ... | | | | |
| $4^9$ | ... | ... | ... | ... | 40,39,37,25,1,0,23,38,4,2,27,26,37 | ... | ... | ... | | | | |
| $5^{10}$ | ... | ... | ... | ... | ... | 40,39,37,25,1,0,24 | ... | ... | | | | |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | ,38,5,2,27,26,37 | | | | | | |
| $28^{11}$ | ... | ... | 40,39,37,25,1,0,36,38,28,26,37 | ... | ... | ... | ... | ... | | | | |
| $42^{12}$ | ... | ... | ... | ... | ... | ... | ... | ... | | | | |
| $33^{13}$ | ... | ... | ... | ... | ... | ... | ... | ... | 33,25,1,0,42,37,40,39,38 | | | |
| $34^{14}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | | | |
| $35^{15}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | | | |
| $14^{16}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | 14,2,1,0,35,33,25,42,37,40,39,38,34 | | |
| $15^{17}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | | |
| $16^{18}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | | |
| $17^{19}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | | |
| $20^{20}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | 20,17,40,39,37,25,1,0,22,38,3,2,27,26,37,16,15,23,4,14,35,33,42,34 | 20,17,40,39,37,25,1,0,22,38,3,2,27,26,37,16,15,23,4,14,35,33,42,34 |
| $21^{21}$ | ... | ... | ... | 21,20,17,40,39,37,25,1,0,22,38,3,2,27,26,37,16,15,23,4,14,35,33,42,34 | ... | ... | ... | ... | ... | ... | ... | ... |
| $16^{22}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| $17^{23}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| $20^{24}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | 21,20,17,40,39,37,25,1,0,22,38,3,2,27,26,37,16,15,23,4,14,35,33,42,34 | 21,20,17,40,39,37,25,1,0,22,38,3,2,27,26,37,16,15,23,4,14,35,33,42,34 |
| $21^{25}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| $16^{26}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| $43^{27}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| $10^{28}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| $11^{29}$ | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| end | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

In table 4, the forward computation for the sample program in figure 11 is shown. The algorithm computes slices for all variables defined up to the end of execution.

One specific issue should be noted that existing slice is reused for variables defined at the same execution position. For example, at execution position $40^4$, a variable *elevator* is defined. At the same time, four other data member variables *alarm_on (AlarmElevator),*

*current_floor (Elevator), current_direction (Elevator), top_floor (Elevator)* are also defined at the execution position (as shown in table 2). For their slice computation we can reuse the slice {statements: 40, 39, 38, 37, 25, 1, 0} from variable *elevator* because that they are data members of the object of *elevator*.

## 3.7 Comparison with Korel and Yalamanchili's Algorithm

In [Kor88] [Kor90], Korel and Yalamanchili introduced a forward method for computing dynamic program slices. The dynamic forward slicing algorithm for OO program presented in our research is basically based on Korel and Yalamanchili's algorithm. However, we extend Korel's original algorithm to adapt it for OO program components and language constructs. In what follows we provide a detailed comparison with Korel's algorithm and list the major extensions of our algorithm.

Commonalities between the two algorithms:

- The slicing criterion for both algorithms is similar, computing slices for all variables defined or modified during a program execution.
- Both algorithms compute the program slices at run-time in a forward direction, without requiring any major recording.
- Both algorithms compute executable program slices that guarantee the same behaviour as original program if given same input to run. Although executable slices are often larger than non-executable slices, executable slices have the benefit

that one can verify their correctness by comparing the behavior with original program.

The following major differences between the two algorithms can be identified:

- Our forward slicing algorithm differentiates a declaration slice and a definition slice. A declaration slice is computed only once and never over-written; slice contributed by one definition may be substituted by next definition/modification.
- We have not inherited the concept of TopSlice that was used by Korel's algorithm. The TopSlice generates additional overhead with respect to space complexity. In our approach, this additional overhead is avoided by utilizing some static information at run-time to compute the dependencies stored originally in the TopSlice.
- The capturing of variable declarations within *limited* scope. Defined variables within the scope of a statement *s* are automatically registered with the statement at run-time. Whenever the statement slice changes, all variables registered with this particular statement will be dynamically notified of the change. We refer to this strategy as *backward registration* (executed components register with early executed one). We further extend this strategy to handle relationships among function arguments and their corresponding actual parameter variables. This strategy is referred as *forward registration*. The general idea is that dependent components are informed about a change in the control dependent component.

- Slicing a procedural program is the pre-condition to slice an OO program. Our algorithm supports inter-procedure slicing by resolving *function-call dependence*, *parameter-in dependency*, and *parameter-out* dependency. Our algorithm sort function calls in one calling site (statement) to ensure the exact mapping to functions.

- The presented dynamic forward slicing algorithm supports slicing OO programs and all of their language constructs, such as objects and user defined data types. Our algorithm also handles *inheritance* and *polymorphism* more precisely than static slicing algorithms. Although the *inheritance* information is derived from static source code parsing, the information is refined through run-time information. The inheritance is modeled by a *CDClass* data structure.

- Improved the performance of the slicing algorithm by providing direct access of already computed program components. This storage is mainly reserved for static program structures and program features such as location of namespace, class, function/method, statement, and type.

## 3.8 Optimization of Dynamic Forward Slicing for OO Program Algorithm

One of the major challenges of slicing OO programs is the support for unstructured programming language constructs, e.g. *goto/label*, *break*, and *exceptional handling*. In particular *exception handling* exhibits a major feature of OO programs; most existing slicing algorithms do not support slice computation in the presents of exception handling. In an optimized version of our dynamic forward slicing algorithm we provide an explicit support for *exception handling*. Prior to the detailed description of this optimized algorithm, we introduce notions of *exception handling* and relevant dependencies associated with it.

### 3.8.1 Exception Handling

Exception handling is a mechanism used in OO programs for detecting, delivering, and handling run-time errors. Exception handling provides an approach to separate error-handling code from "real" code, making programs more readable and maintainable.

The exception object is a carrier that contains information from the exception occurrence to the handler. It may have attributes or act merely as a signal. If an exception occurrence is identified, the exception mechanism offers the possibility of throwing an exception object. When an exception object is thrown, the intuition is that the exception object 'travels' back the dynamic call-chain until an exception handler for this exception object type has been located. An exception handler is a syntactic handler dealing with one

particular exception. [Jor00]

For the remainder of the thesis, we restrict our discussion to Java exception- handling constructs; C++ programs can be constructed and analyzed in a similar fashion.

In Java, exceptions are categorized into *synchronous* exceptions or *asynchronous* exceptions. *Synchronous* exception may be raised by an expression evaluation (including function call), a statement execution, or simply a throw statement. There are two kinds of synchronous exceptions further clarified in Java: *checked* exception and *unchecked* exception. For *checked* exceptions, the compiler must find a handler or a signature declaration for the method that raises the exception; for *unchecked* exceptions, the compiler does not attempt to find such an exceptional handler or a method signature declaration. *Asynchronous* exceptions occur when either the Java Virtual Machine raises an instance of *InternalError* (due to the faults in the virtual-machine software, the host system software, or the hardware), or a thread invokes the *stop*() method that raise an instance of *ThreadDeath* in another thread.

In Java, a *try* statement represents the exception-handling construct. The legal constructs of a *try* statement are *try-catch*, *try-finally*, and *try-catch-finally*. A *try* block (called guarded section) is the set of statements whose execution must be monitored for the possible exception occurrence. A *catch* block, which must be associated with a *try* block, is a sequence of clauses that specifies the exception handler. Each *catch* clause defines the type of exception it handles, and contains a set of codes that are to be executed when an

exception of the type is caught. A *finally* block can be associated with a *try* block. Nested cleaning up code may always be executed, regardless of the way in which control flows out of the *try* block. A *finally* block can raise an exception, which masks out the previous exception.

The sample program (in Java) in figure 14 is used to illustrate the exception handling. In this example, executing the function *connectDB()* may generate and throw two types of exceptions that are implemented. The exception may be an object of *ClassNotFoundException*, or an *SQLException*. The exception handling is deferred until the execution leaves the caller *connectDB()* in line 10 and arrives at a corresponding *catch*, within function *main()*. Depending on the program state, it becomes the task of main functions to accept or forward the received exception. During the execution of a program, only the active handler most recently encountered by the control flow will be invoked, others are ignored. If a function throws an exception for which there is no exception the program will terminate.

Our extended version of the dynamic forward slicing algorithm handles exception handling by identifying and including the relevant parts of the *throw, try-catch-finally* statements that are relevant to a particular variable of interest.

```
0 package datatransfer;
1 class DataTransfer extends Serializable {
2    public void openInputStream() throws IOException {};
3    public Connection connectDB() throws ClassNotFoundException, SQLException {};
4    public String readFromFile() throws
                     OptionalDataException,
                     ClassCastException,
                     IOException {
                          throw new OptionalDataException() };
5 class Connection {
     public static void writeToDB( String word ) throws
                     SQLException, Exception {} }
6    public static void main(String[] argv) {
7         DataTransfer sam = new DataTransfer();
8         try{
9              sam.openInputStream();
10             Connection con = sam.connectDB();
11             String word = sam.readFromFile();
12             while( !word.equals("END") ){
13                  con.writeToDB(word);
14                  word = sam.readFromFile();
               }
15        }catch(OptionalDataException e){ //...
16        }catch(ClassNotFoundException e){ //...
17        }catch(ClassCastException e){ //...
18        }catch(SQLException e){ //...
19        }catch(IOException e){  //...
          }
     }
  }
```

**Figure 14** a simplified Java style program illustrating the application of exception handling

### 3.8.2 Additional Dependencies

Beside the dependencies examined in chapter 3, we need to analyze the static dependences existing within *try*, *catch*, and *finally* block and dynamic dependences between throwing

an exception and its associated *catch* block that eventually catches the exception. It is obvious that only handlers that have been executed are possible to be included in the slice. The following are some of the additional dependencies introduced to support the slice computation in the presents of exception handling.

Static dependencies:

- Neither a catch block nor a finally block can be stand-alone. Their existence is dependent on the existence of a try block.
- Since *checked* exceptions must be visible for the exception handlers (catch blocks) during compilation, these handlers are regarded as not removable in order to guarantee an executable slice. Namely, if a *checked* exception signature depends on the existence of a *catch*.

Dynamic dependencies:

- During the execution of a Java program, the activated exception handler is directly dependent on the last executed statement, which may be an explicit *throw* statement, a function call (Java virtual machine defined), or an evaluation of an expression.

### 3.8.3 Optimized Algorithm

The optimized algorithm is shown in figure 15, including two new functions (in bold) that correspond to the extension of the optimized algorithm dealing with exception handling. Figure 15 is based on Figure 12, and redundant contents in Figure 12 are not shown in Figure 15 again. Details are shown in following figure 16.

**Program dynamicForwardSlicingAlgorithmofOOPWithExceptionHandling**

```
:
:
:
7        functionProcessDynamicForwardSlicingAlgorithmofOOP(f); See procedure EHA;
:
:
:
13       s = createStatementDynamicForwardSlicingAlgorithmofOOP(); See procedure EHB;
:
:
:
32
```

**Figure 15** dynamic forward slicing algorithm of OOP with exception handling

```
procedure EHA(){
      figure out all exceptions from signature;
      for each above exceptions e type of checked {
            figure out the matched exception handler eh in latest try statement from ST;
            build static dependence from f to eh, i.e. f dependent on eh;
      }
}

procedure EHB(){
      if( statement s is type of try ){
            figure out and create corresponding exception handlings (catch) statements;
            for each above statements cs{
                  register cs with s;  //use adaptive space
                  set cs.parent-statement = s;
            }
      }

      if( statement s is type of catch ){
            build dependence on the top of ST for s;
      }

      if( statement s is type of finally ){
            set cs.parent-statement = latest try statement in ST;
      }
}
```

**Figure 16** procedures to deal with exception handling

Changes made to the original algorithm include: modifications to the data structure *ST* (statement) to support the notion of *try* statement and the associated exception handlers. Other changes include a change in the procedure EHA, the static dependencies to the relevant exception handlers are identified. Because a *catch* statement is directly dependent on a *throw* statement or indirectly dependent on an expression that throws an exception, the last executed statement on the stack is used by the *catch* statement before stack *ST* is popped (see procedure EHB). Finally, either a *catch* statement or a *finally* statement is associated to a *try* statement, that can be seen as its parent statement (direct control dependence).

### 3.8.4 Sample Computation

Using the sample program from figure 14, the program is executed up to statement 11, where an exception *OptionalDataException* is thrown. The dynamic forward slice computation in the presents of exception handling is shown, along with associated execution trace $\{6^1,7^2,8^3,9^4,2^5,10^6,3^7,11^8,4^9,15^{10}\}$ in table 5.

For example, variable *sam* is defined in statement 7 that corresponds to the execution position 2 in the trace. When the execution arrives at statement 8, the collected dependence for this *try* statement is {8,6,1,0}, representing the normal control dependence and

ownership dependence. The slice for the catch blocks at statement 15,16,18,19 is temporally collected by the *try* statement, using static parsing information. Block 17 is excluded because it handles an unchecked runtime exception. After functions of *sam.openInputStream*(), *sam.connectDB*(), and *sam.readFromFile*() are invoked respectively, 15,16,18, and 19 in slice of *try* are marked non-removable. When the execution exits from the *try* block, the slice for *try* will be re-computed that will result in the new slice for the try block {8,6,1,0,15,16,18,19}. Variable *con* is notified about the change in the slice because *con* is defined within try's scope.

In this particular example, statement 15 actually catches the exception, where the exception object was defined. This *catch* clause inherits the slice of the *throw* statement inside the function *readFromFile()*.

**Table 5** processing of internal storage when slicing sample program in Figure 14, given assumption that function *readFromFile()* is eventually executed

| trace | *NS* | *CL* | *FUNC* | *FUN* | *ST* | *VAR* | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Sam | Con | word | e |
| $6^1$ | datatransfer | DataTransfer | Main | Main | | | | | |
| $7^2$ | ... | ... | DataTransfer<br>Main | DataTransfer<br>Main | 7 | {7,6,1,0} | | | |
| $8^3$ | ... | ... | | ... | 8 | ... | | | |
| $9^4$ | ... | ... | openInputStream<br>main | ... | 8,9 | ... | | | |
| $2^5$ | ... | ... | ... | DataTransfer<br>openInputStream<br>main | 8,9 | ... | | | |
| $10^6$ | ... | ... | connectDB<br>main | ... | 8,10 | ... | {10,8,7,6,1,0,3} | | |
| $3^7$ | ... | ... | ... | DataTransfer<br>openInputStream<br>connectDB<br>main | 8,10 | ... | ... | | |
| $11^8$ | ... | ... | readFromFile<br>main | ... | 8,11 | ... | ... | {} | |

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 4[9] | … | … | … | DataTransfer openInputStream connectDB readFromFile main | 8,11 | … | … | {} |  |  |
|  |  |  |  |  |  |  | {10,8,7,6,1,0,3, 15,16,18,19} |  |  |  |
| 15[10] | … | … | main | … | 8,15 | … | … | {} | {7,6,1,0,15,16,1 8,19,11,4} |  |
| end | … | … |  | … |  | … | … | {} | … |  |

# 4 Experimental Analysis

As part of this research we implemented the version 1.0 of the dynamic forward slicing algorithm within the CONCEPT project. Beside the dynamic run-time information, our algorithm also utilizes static information for the parsing of source code that is provided by the CONCEPT project. In this section, we will discuss implementation issues and provide some initial experimental results from the use of our algorithm.

The ***CONCEPT*** (Comprehension Of Net-Centred Programs and Techniques) is a software analysis and comprehension framework with the major goal to address current and future challenges in the comprehension of large and distributed systems. The motivation is to provide programmers with novel comprehension techniques, supporting the understanding of large and distributed systems. These techniques are based on a variety of source code analysis, visualization, and application approaches. The presented dynamic forward slicing algorithm is one of the analysis techniques to support this comprehension process. The dynamic forward algorithm was integrated as shown in Figure 17 in the overall CONCEPT system architecture.

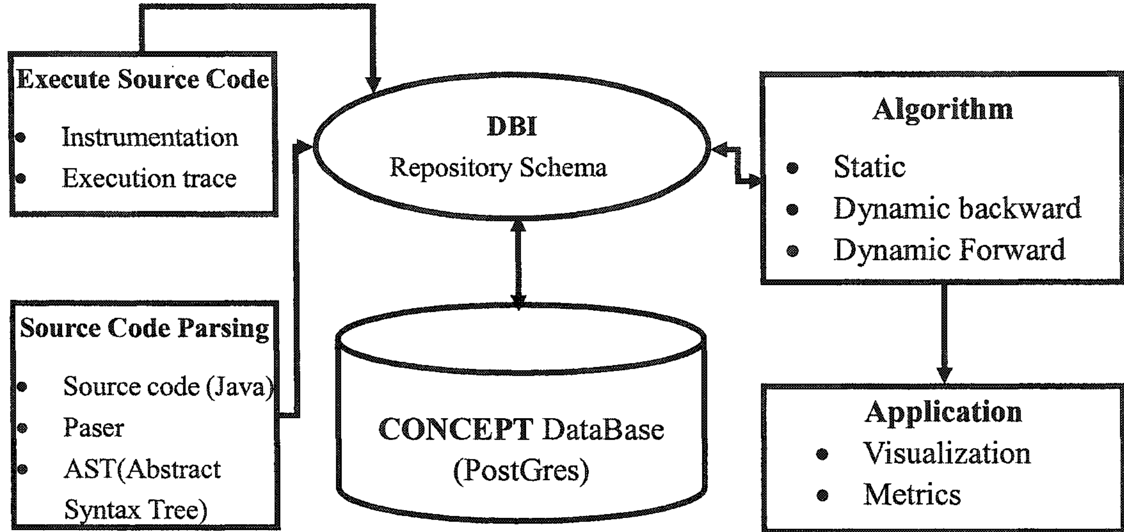



**Figure 17** CONCEPT system architecture

The source code analysis process with the CONCEPT project is supported by two different approaches. The first one, a static approach that relies solely on static information from parsing the source code. The static information is represented in the form of an abstract Syntax Tree (AST) that is stored in a knowledge base. The AST provides detailed static information about structural and logical dependencies in the source code.

The second approach is to utilize dynamic information that is derived by instrumenting the source code. This dynamic information is the main information source for the dynamic forward algorithm.

## 4.1 Implementation

We have implemented the algorithm introduced in chapter 3 using Java version 1.4. *CDFunctioncallFactory*, *CDStatementFactory*, and *CDFunctionFactory* function as stacks simulating the control flow. Figure 18 shows the high-level design of the algorithm implementation.

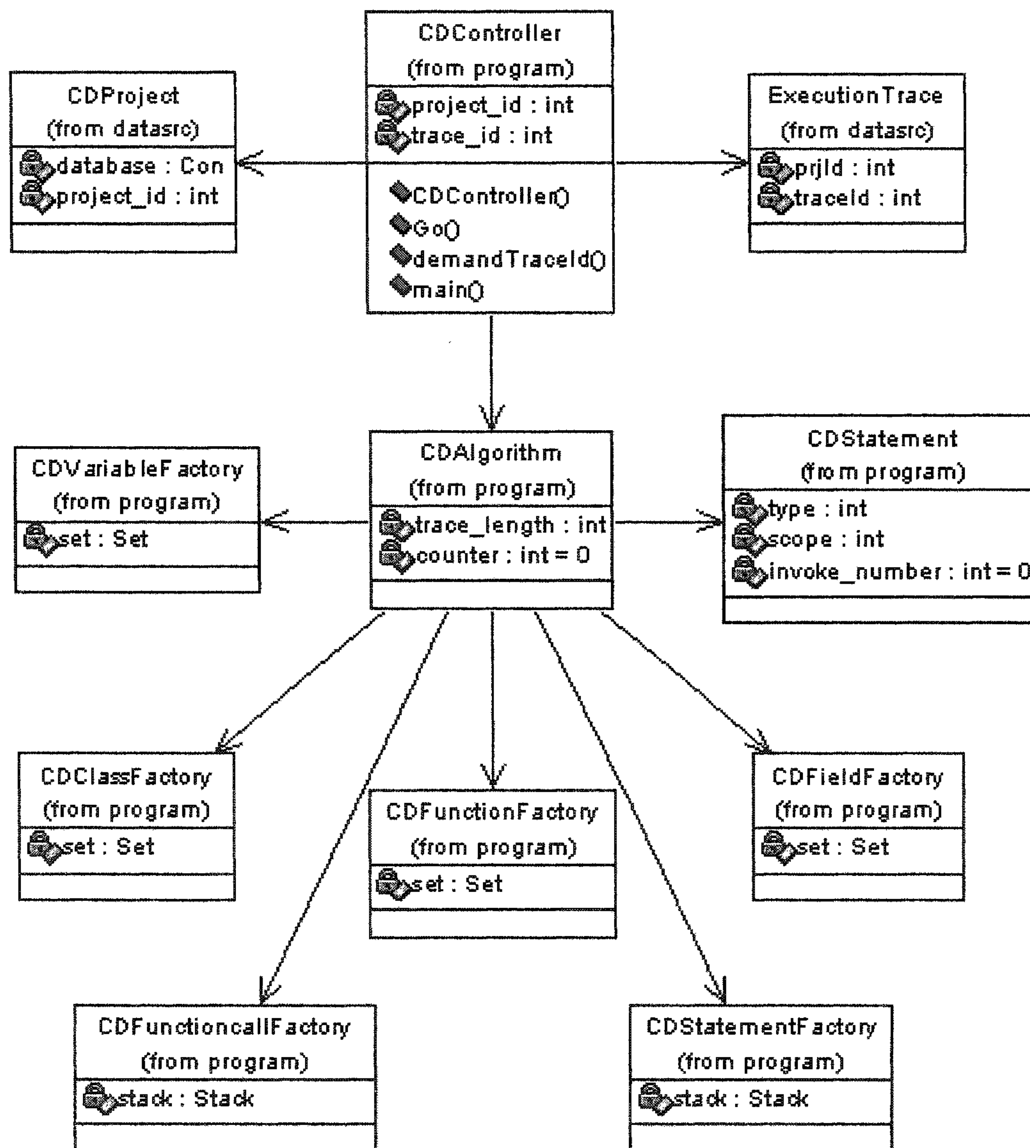



**Figure 18** High-level view of the algorithm implementation

The entry point of this implementation starts with the accessing dynamic data from database by *Execution Trace*. Parsed information of the source code will be accessed by *CDProject*. *CDController* schedules above two procedures, categorizes and forwards obtained information to the factories.

Static information about program components such as namespace, class, function (method), and statement are identified by unique *SourceElement*(s). We illustrated this feature in figure 19.

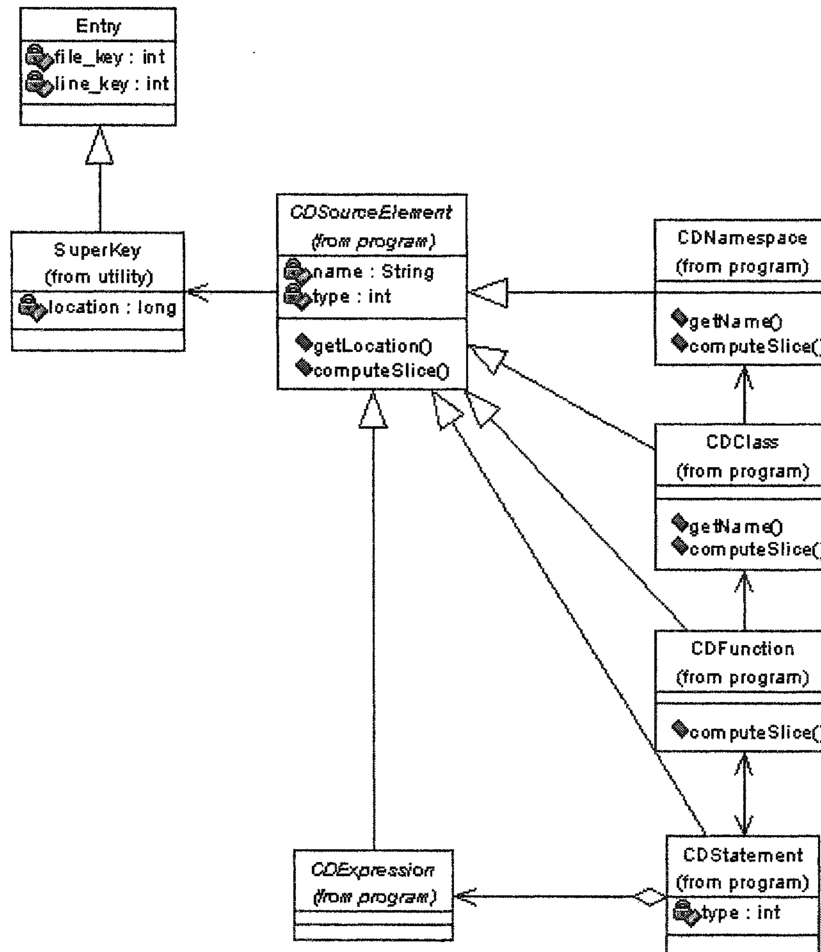



**Figure 19** Class diagram for modeling the static information

Expression represents a key element among program components. An expression is composed of constant, variables, function calls and their combinatory operators. If a slice is expression oriented, it may be a union of slice of involved variables and function calls. Following class diagram shown in Figure 20 illustrates this feature by presenting all types of variables, function call, and necessary composition.

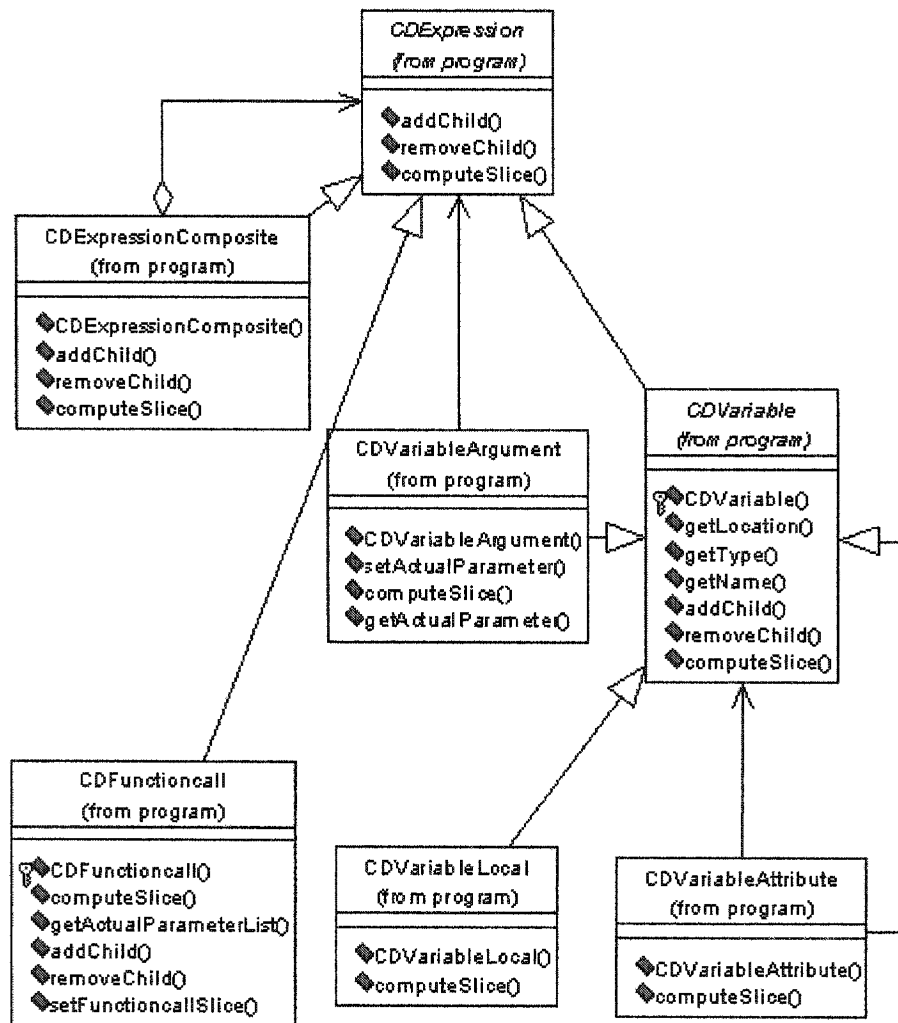



**Figure 20** Class model for representing the Expression structure

In our algorithm the registration strategy is used to bind one program component with

another to guarantee that the "observer" obtaining a notification about any change to the slice. For instance, a defined variable is registered with an executed stored parent statement. In our implementation, any declared or defined variable is stored in a set that is an attribute of *CDStatementFactory*.

## 4.2 Experiment settings

In order to evaluate the quality of our algorithm, we performed an initial experimental study to investigate the memory usage, execution performance, and precision of the slicing algorithm. For the experimental setting, we used two different software systems: one system is the implementation of an elevator scheduling, named ELEVATOR; and the other is a subsystem of CONCEPT used for design pattern recovering, named *concept.java*.

The experiments are performed on a Pentium IV 2.8G CPU, 1G memory, and 80G Ultra IDE hard disk. The operating system is Windows 2000 SP3. The database system, PostgreSQL 7.3, is installed on a server running the Redhat Linux 9 operating system. The workstation is connected through 100MBits Ethernet network to the server.

As part of this initial experiment we monitored the memory usage through the windows task manager. An internal system time recorder was used. We embed a time consumption recorder in our implementation that displays the start and end time for the major processes involved in the slice computation. We excluded the time for loading the data model from the database. We use the factor of *precision* to measure the accuracy of slices obtained by our algorithm. The precision compares the size of the manually computed slice with the

slice size obtained by our algorithm for the same variables and attributes.

The two Programs used in our experimental settings are:

- The ***elevator*** package is a simple sample program simulating an elevator. Its main purpose is to provide a testing environment for some of the major OO program language constructs. The *elevator* project utilizes inheritance, polymorphism, and nested classes. The elevator package contains of 3 files and 258 lines of source code (LOC).
- The ***concept.java*** package version 1.0 is part of CONCEPT project implementing the design pattern recovering. It includes several nested sub packages, such as language parser, storage, language analysis, program model representation, matching, and a Java de-compiler. This package contains 196 files, 14,768 lines of code (LOC), and 11,658 lines of code without comments.

## 4.3 Results from the experimental analysis

### 4.3.1 Memory usage

Table 6 shows the information concerning memory usage during the slice computation using our dynamic forward slicing algorithm.

**Table 6** memory usage when slicing various execution trace of two program

| | **elevator** | **concept.java** |
|---|---|---|
| Lines Of Code (LOC) | 258 | 14,768 |
| LOC without comment | 258 | 11,658 |
| Number of classes | 4 | 201 |
| Number of methods | 16 | 2,621 |
| Lines of execution trace (certain input) | 756 | 42,843 |
| Number of computed attributes | 17 | 1057 |
| Number of computed local variables | 20 | 2781 |
| Peak of memory usage | **968K** | **33,205K** |
| Lines of execution trace (certain input) | 6,792 | 129,501 |
| Number of computed attributes | 27 | 3,309 |
| Number of computed local variables | 25 | 1,340 |
| Peak of memory usage | **1,320K** | **34,106K** |
| Lines of execution trace (certain input) | 38,424 | |
| Number of computed attributes | 29 | |
| Number of computed local variables | 23 | |
| Peak of memory usage | **1,602K** | |

From the results shown in Figure 6, one can conclude that the space consumption is mainly related to the amount of variables defined/modified during a program execution. Meanwhile, the length of execution trace has only an indirect effect when memory consumption increases. If slicing a program with heavier complexity and coupling, such as concept.java, the space consumption in storing slice increases too.

### 4.3.2 Execution time

The following table shows the computation time for the same program executions as shown in Table 6

**Table 7** Execution time of pattern recovering process

| | **elevator** | **concept.java** |
|---|---|---|
| Number of executed statements | 756 | 42,843 |
| Number of computed attributes | 17 | 1057 |
| Number of computed local variables | 20 | 2781 |
| Time consumed after data loading from DB | **1.04sec** | **38.66sec** |
| Number of executed statements | 6,792 | 129,501 |
| Number of computed attributes | 27 | 3,309 |
| Number of computed local variables | 25 | 1,340 |
| Time consumed after data loading from DB | **2.63sec** | **1,44.49sec** |
| Number of executed statements | 38,424 | |
| Number of computed attributes | 29 | |
| Number of computed local variables | 23 | |
| Time consumed after data loading from DB | **15.20sec** | |

From the results in table 7, the time complexity is directly related to the number of executed statements and the number of variables declared/modified during a particular program execution. For a simple system, with less variables and attributes involved, while execution trace increases within 40,000 lines, time consumed increases in a linear manner. For a large system, having between 2,000 to 4,500 variables and attributes involved and the number of executed statement being larger than 100,000 lines the algorithm shows an exponential time complexity.

### 4.3.3 Precision

Table 8 shows the precision of slicing algorithm when computing slices for the *elevator* and *concept.java* project.

**Table 8** accuracy of computed slices

| | # Of executed statements for elevator | | | # Of executed statements for concept.java | |
|---|---|---|---|---|---|
| | **756** | **6,792** | **38,424** | **42,843** | **129,501** |
| N°: computed variables | 20 | 25 | 23 | 2781 | 1,340 |
| N°: randomly selected variables | 10 | 10 | 5 | 2 | 2 |
| N°: Average lines of computed slices | 31 | 49 | 175 | 293 | 350 |
| N°: Average lines of manually computed slices | 30 | 45 | 149 | 280 | 321 |
| Accuracy ratio | **0.96** | **0.91** | **0.85** | **0.95** | **0.92** |
| N°: computed local attributes | 17 | 27 | 29 | 1,057 | 3,309 |
| N°: randomly selected variables | 10 | 10 | 10 | 2 | 2 |
| N°: Average lines of computed slices | 9 | 10 | 25 | 38 | 57 |
| N°: Average lines of manually computed slices | 9 | 10 | 25 | 37 | 55 |
| Accuracy ratio | **1** | **1** | **1** | **0.94** | **0.96** |

In order to check the precision of the slices computed by our algorithm, we randomly chose variables and attributes and computed manually the slice for these variables. From the data summarized in above table 8, we may conclude that the general accuracy is averagely above 90%. Accuracy in computing slice of attributes is relatively higher than slicing local variables. However, due to the lack of dynamic information the obtained slice may not be directly executable. Manually modifications to the slice have to be performed, to make the slice executable. After performing these necessary syntactical modifications, the obtained slice showed the same behavior with respect to the slicing criterion as the original program.

# 5 Conclusions

Different program slicing methods are used for maintenance, reverse engineering, testing and debugging. Slicing algorithms can be classified into static slicing and dynamic slicing methods. In several applications, such as debugging, dynamic slices are preferred, since they often produce more precise results. Experimental results from [Lan92] show that the dynamic slice of a program can be expected less than 50% of the executed statements and be well within 20% of the entire program. Several methods for the computation of dynamic slices exist. Most of the traditional algorithms apply backward computation based on a record execution trace. The recording of the execution traces leads to an unbound space complexity for recording the trace.

In this research, we introduced a new dynamic forward algorithm to compute slices for OO programs. The dynamic slices are computed at run-time without requiring any major recording of the program execution. We have also proposed an optimized algorithm to address issues related to exceptional handling. Our algorithm has three specific features: eliminating any overhead copies of TopSlice (conventionally introduced in dynamic forward algorithm); instead, registering and notifying strategies are applied to compute precise and executable slice; static information are reusable in reducing time complexity; identifying the dependencies "hidden" behind of OO program feature. Within the CONCEPT project, we have implemented our algorithm, which is capable of slicing rather larger Java programs. The feasibility of our algorithm is proved concerning the preliminary

design and experimental result.

The next goal is to adapt our algorithm to compute dynamic slices for C++ programs. In addition, to be able to slice recursive program is another plan in the near future. Current inter-procedural analysis assumes no cycles existed between a caller and the callee, presence of cycles may require analysis of one function call multiple times along with the recursion until "summary" information settles. Another future work concerns how to extend our algorithm to applications in software coupling metrics analysis. Moreover, being able to support software architecture recovery is one of next researches. Software architecture is a high-level system abstraction. Its complexity exploits the generic description of a system into infinite behaviors and non-trivial features at the architecture level. In order to understand a system architecture design, we will build dynamic software architecture. Dynamic software architecture represents the run-time behavior of those parts of software architecture that are selected according to a particular slicing.

# References

[Agr90] Hiralal Agrawal and Joseph R. Horgan. *Dynamic Program Slicing*. ACM SIGPLAN'90 conference on programming Language Design and Implementation, 25(6):246-256, June 1990.

[Agr91] Agraval, H., DeMillo R., and Spafford, E. *Dynamic Slicing in the Presence of Unconstrained Pointers*. In Proceedings of ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4) 1991, pp. 60-73. Also Purdue University technique reports SERC-TR-93-P.

[Agr94] H. Agrawal. On *Slicing Programs with Jump Statements*. In proceedings of ACM SIGPLAN '94 Conference on programming language design and implementation. SIGPLAN Notice, pages 302-12, June 1994.

[Arp01] Arpad Beszedes, Tamas Gergely, Zsolt Mihaly Szabo, Janos Csirik and Tibor Gyimothy Research Group on Artificial Intelligence. *Dynamic Slicing Method for Maintenance of Large C Programs*. Proceeding of Fifth European Conference on Software Maintenance and Reengineering, P. 105. 2001.

[Bat99] Don Batory, Rich Cardone, & Yannis Smaragdakis. *Object-Oriented Frameworks and Product-Lines*. 1st Software Product-Line Conference, Denver, Colorado, August 1999. http://www.cs.utexas.edu/users/richcar/splc1.pdf.

[Csa02] Csaba Faragô, Tamàs Gergely. *Handling Pointers and Unstructured Statements in the Forward Computed Dynamic Slice Algorithm*. Acta Cybernetica **15** (2002) 489-508, http://www.inf.u-szeged.hu/informatika/local/acta/vol15n4/cikk2.html.

[Dar01] Darren C. Atkinson, Markus Mock, Craig Chambers, Susan J. Eggers. *Program Slicing Using Dynamic Points-To Data*. http://citeseer.nj.nec.com/atkinson02program.html.

[Harm01] Mark Harman and Robert M. Hierons, *An Overview of Program Slicing*. http://www.brunel.ac.uk/~csstmmh2/sf.html. Software Focus 2, 3 (2001), 85-92. 2001.

[Har98] Mary Jean Harrold, Ning Ci. *Reuse-Driven Interprocedural Slicing*. Proceedings of the international Conference On Software Enginering, Kyoto, Japan, April 1998. IEEE Copyright.

[Hor90] S. Horwitz et al. *Interprocedural Slicing Using Dependency Graphs*. ACM Trans.

Programming Languages and Systems, 12(1): 35-46, 1990.

[Jeff01] Jeff Russell. *Program Slicing Literature Survey*. December 2001. Seminar and Draft Materials, University of Texas at Austin, October 2001.

[Jfe87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. *The program dependence graph and its use in optimization*. ACM Trans-actions on Programming Languages and Systems, 9(3):319–349, July 1987.

[Jor00] Jørgen Lindskov Knudsen, *Exception Handling versus Fault Tolerance*. 2000.

[Kam93a] Kamkar, M., Fritzson, P., and Shahmehri, N. *Three Approaches to Interprocedural Dynamic Slicing*. Microprocessing and Microprogramming 38 (1993), 625-636.

[Kam93b] M. Kamkar, "*Interprocedural Dynamic Slicing with Applications to Debugging and Testing*," *PhD Thesis*, Linkoping University, 1993.

[Kor88] B. Korel and J. Laski, *Dynamic Program Slicing*, Information Processing Letters, vol. 29, no. 3, 1988, pp. 155-163.

[Kor90] B. Korel and J. Laski, *Dynamic Slicing of Computer Programs*, The Journal of Systems and software, vol. 13, no. 3, 1990, pp. 187-195.

[Kor94] Bogdan Korel and Satish Yalamanchili. *Forward Computation of Dynamic Forward Slices*. 1994 ACM 0-89791-683-2/94/0008.

[Kor95] Korel,B., "*Computation of Dynamic Slices for Programs with Arbitrary Control-flow*", The 2nd International Workshop on Automated and Algorithmic Debugging, pp. 1-41, St. Malo, France, 1995.

[Kor97] Korel, B., "*Computation of Dynamic Slices for Unstructured Programs*", IEEE Transactions on Software Engineering, 23(1), pp. 17-34, 1997.

[Lan92] W.Landi and B. G. Ryder. *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing.* In Proceeding of SIGPLAN '92 Conference on Program Language Design and Implementation, P235-248, June 1992.

[Lar96] L. D. Larsen and M. J. Harrold, *Slicing object-oriented software*. 18th International Conference on Software Engineering (ICSE), pages 495-505, Berlin, March 1996.

[Li01] Bixin Li, *A Hierarchical Slice-Based Framework for Object-Oriented Coupling Measurement*, TUCS Technical Reports, Turku Centre for Computer Science, NO415, 2001.

[Lia98] Donglin Liang, et al. *Slicing Objects Using System Dependence Graphs*. International Conference on Software Maintenance, pp. 358-367, November 1998.

[Ott84] Ottenstein, K., and Ottenstein, L., "*The Program Dependence Graph in a Software Development Environment*", In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices 19(5), pp. 177-184, 1984.

[Ril98] J. Rilling. *Program Slicing Algorithm*. PhD thesis. Illinois Institute of Technology, 1998.

[Ste99] Christoph Steindl. *Program Slicing for Object-Oriented Program Language*. A dissertation submitted to JOHANNES KEPLER UNIVERSITY LINZ, 1999.

[Tip95] F. Tip. *A survey of program slicing techniques*. Journal of Programming Languages, 3(3):121–189, Sept. 1995.

[Wan96] Yamin Wang, Wei-Tek Tsai, Xiaoping Chen, Sanjai Rayadurgam. *The Role of Program Slicing in Ripple Effect Analysis*. University of Minnesota Computer Science & Engineering Technical Report, TR number: TR 96-014.

[Wei82] Weiser, M., *Program Slicing*, IEEE Transactions on Software Engineering 10(4), pp. 352-357, 1984.

[Wis00] *The Wisconsin Program Slicing System*. http://www.cs.wisc.edu/wpis/html/

[Zha01] Jianjun Zhao. *Slicing Aspect-Oriented Software*. IPSJ SIGNotes Software Engineering Abstract No.135 – 007.

[Zha02] JianJun Zhao. *Applying Program Dependence Analysis to Java Software*. Proc. Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium, pp.162-169, Tainan, TAIWAN, December 1998. http://citeseer.nj.nec.com/zhao98applying.html.

[Zhan03] Xiangyu Zhang, Rajiv Gupta, Youtao Zhang. *Precise Dynamic Slicing Algorithms*. The University of Arizona Dept. of Computer Science Tucson, Arizona 85721. The Univ. of Texas at Dallas Dept. of Computer Science Richardson, TX 75083.

# Appendix: Additional Procedures of DFS

```
☐ procedure functionProcess(Function f)
BEGIN
   compute out all argument variables of currently executed function;

   function-call fc = top of FUNC;
   compute dependence including actual parameters' slices and the caller for fc;
   dynamic-slices of f = slices of fc;

   for each argument variable v in current function {
      declare-slice( v ) = slice/dependence of f;
      put v into VAR;

      if(corresponding actual parameter v' of v is a parameter-out variable){
         register v' with v.
      }
   }
   return;
END

☐ procedure Statement createStatement( )
BEGIN
   new a statement s from template;
   s.parent-function = peek of top of FUNC;
   compute out the type of s;
   s.parent-statement = peek of top of ST if they have same parent-function;
   //excluding recursive functions

   find out all defined variables for s;
   for each defined variable v with strict sequence {
      createExpressionDynamicForwardSlicingAlgorithmofOOP(root for v);
   }
   return s;
END

☐ Procedure computeStatementSlice( Statement s )
BEGIN
   quote slice/dependence from parent function;  //ownership dependence
   quote slice/dependence from parent statement;  //control dependence
   for each expression ex in statement s{
      for each element e in ex {
         case type of constant:
```

```
      case type of variable:
        slice( s ) ∪= slice of declare-slice of e;
        slice( s ) ∪= slice of dynamic-slice of e;
        if( e is type of attribute variable of an object o ){
          slice( s ) ∪= slice of o; //both declare-slice and dynamic-slice;
        }
      case type of function call:
        slice( s ) ∪= slice/dependence of e;  //including dependence on return
    }
  } //data dependence
  return;
END
```

```
procedure createExpression(AST root )
BEGIN
  if( type of root is a constant c ){
    create c;
    add c into the set of current expression;
  }

  if( type of root is a variable v ){
    get v from VAR;
    add v into the set of current expression;
  }

  if( type of root is a function call ){
    function-call=create this function call including all its parameter-in/out;
    if( function-call is user defined ){
      push function-call into FUNC;
    }
    add function-call into the set of current expression;
  }

  createExpressionDynamicForwardSlicingAlgorithmofOOP(root.right);
  createExpressionDynamicForwardSlicingAlgorithmofOOP(root.left);
  return;
END
```

```
procedure variableComputing( Variable v, Statement s )
BEGIN
  switch( v ){
    case v is object variable:
      if( v is declared )
        declare-slice( v ) = slice( s );
      if( v is newed ){ //instantiation
```

```
            dynamic-slice( v ) = slice( s );
            create data member variables and reset their declare-slice;
            if( base class of v's class existed ){
               create data member variables of the base class;
               reset declare-slices for above created variables;
            }
            put all these data member variables into VAR;
            register v with every these data member variables; //yes
            register each data member variable with v;
         }
         if( v is defined/such as assigned by another object ){
            dynamic-slice( v ) = slice( s );
            for( each data member variable v' of v ){
               dynamic-slice( v' ) ∪= dynamic-slice( v );
               dynamic-slice( v' ) ∪= slice( corresponding data member of assigner of v );
            }
         }
         break;

      case v is a local variable or a global variable as primary type:
         if( v is defined )
            dynamic-slice( v ) = slice( s );
         else
            declare-slice( v ) = slice( s );
         break;

      case v is a defined object attribute variable:
         dynamic-slice( v ) = slice( s );
         break;

      case v is a defined function argument variable:
         update/notify dynamic-slice of corresponding parameter-out variable;
         break;
   }
   return;
END

procedure getDegree( )
BEGIN
   if( parent-statement of this-statement existed ){
      this-statement.depth-degree=1+ parent-statement.depth-degree;
   }else
      this-statement.depth-degree=1;
   return;
END
```