AN INTERACTIVE DEBUGGING TOOL FOR C++ BASED ON DYNAMIC SLICING AND DICING PART I: DEFINITIONS AND ALGORITHMS

Winai Wichaipanitch[1] and M. H. Samadzadeh[2]

Abstract

The main objective of this work was to develop an interactive debugging tool for C++ programs. The tool that was developed is called C++Debug and it uses program slicing and dicing techniques. The design started by including simple statements first and then expanded to pointers, structures, functions, and classes. In order for C++Debug to be more powerful, dynamic slicing rather than static slicing was chosen. The work includes new algorithms that handle Class, Function, and Pointer in C++.

The results of this work are reported in two parts:

PART I: Definitions and Algorithms,

PART II: Implementation, Testing, and Evaluation.

This is part I that reports on the definitions and algorithms, how to compute a slice, and the dicing procedure.

1. Introduction

Since the article "Program Slicing" by Mark Weiser was initially published in 1981 [10], program slicing has gained wide recognition in both academic and practical arenas. Several debugging tools have been developed that utilize program slicing. For example, Focus [6] (designed and implemented by Lyle in 1984) was designed to be used with Fortran programs, and C-Sdicer [7][8] (designed and implemented by Nanja and Samadzadeh in 1990) and C-Debug [9] (designed and implemented by Wichaipanitch and Samadzadeh in 1992) were designed to be applicable to C language programs based on dynamic slicing. Program slicing [1][2][10][11][12][13] is one of the debugging methods used to localize errors in a program. The idea of program slicing is to focus on the statements that have something to do with a certain variable of interest (criterion variable), with the unrelated statements being omitted. Using slicing, one obtains a new program of generally smaller size that still maintains all aspects of the original program's behavior with respect to the criterion variable. Dynamic slicing differs from static slicing in that it is defined on the basis of a computation or an execution rather than on all possible computations. Furthermore, it allows one to treat the elements and fields in dynamic records as individual variables [5]. As a result, the slice size computed based on the dynamic slicing technique is generally smaller. Moreover, dynamic slicing allows one to keep track of the run-time

[1] On leave from Computer Engineering Department,Faculty of Engineering, Rajamangala Institute of Technology, Klong 6, pratumthani, Thailand.

[2] Oklahoma State University Computer Science Department 219 MSCS Stillwater, OK 74078, samad@a.cs.okstate.edu

type binding (involving the type of each object) that is unknown at compile time but is determined when the program is executed. Dynamic slicing technique was used in this study. Dicing technique [6][7][8] can then be used to compare two or more slices resulting from the program slicing technique in order to identify the set of statements that are likely to contain an error. The formal model of static/dynamic slicing/dicing is presented. There is a need for debugging tools that are capable of making some deductions regarding the presence and location of errors in programs.

2. Definitions

A number of definitions and algorithms originally introduced by Korel and Laski [3][4][5] were modified, in order to compute slices in classes, objects, arrays, pointers, references, dynamic allocation operators, function overloading, copy constructors, default arguments, operator overloading, inheritance, virtual functions, polymorphism, templates, and exception handling of a C++ program. Those modified definitions plus a number of new definitions and algorithms are introduced in this chapter.

Based on Korel and Laski's work [5], let the flow graph of a program P be a directed graph (N,A,s,e) and C be a *slicing criterion*, where N is a set of nodes, A is a binary relation on N (a subset of N x N) referred to as the set of arcs, $s \in N$ is a unique entry node, and $e \in N$ is a unique exit node.

Each node in N consists of one statement, including a single instruction, a control instruction, and a function instruction. A single instruction can be, for example, an assignment statement or an input or output statement. A control instruction can be such statements as an **if-then-else** statement or a **while** statement, which are also called *test instructions.* A function instruction can be either a called or a calling function instruction.

An arc(n, m) $\in$ A corresponds to a possible transfer of control from instruction n to instruction m.

A path from the entry node s to some node k, $k \in N$, is called a sequence $<n_1, n_2, ..., n_q>$ of instructions, such that $n_1 = s$, $n_q = k$, and $(n_i, n_{i+1}) \in A$, for all n_i, $1 \leq i < q$. If there are input data that cause a path to be traversed during program execution, the path is *feasible*. A feasible path that has actually been executed for some input is called a *trajectory*.

Definition 1. Let X be an instruction in a program and $X \in IN^+$ (the set of non-negative integers). Let P be the set of instruction numbers in a tested C++ program, then P = {1, 2,..., n} represents a program of length n, where n is the size of the program

$$P = \{ X \mid \text{for all } X \text{ with } 1 \leq X \leq n \}$$

where n = length of the program.

Definition 2. Let F_{name} be a function, i.e., a set of instruction X's in the scope of influence of the function name, where all blank lines are ignored. $F_{name} \subseteq P$, and $F_{name} = F_{main}$ if the program has one function.

$$F_{name} = \{ X \mid \text{for all } X \text{ with } i \leq X \leq k \}$$

where (1) i is the starting line number of function name, $i \in P$

(2) k is the ending line number of function name, $k \in P$

Definition 3. Let T be a trajectory, i.e., a feasible path that has actually been executed for some input [5]. A trajectory of length m is denoted by a list $T = <X_1, X_2, ..., X_m>$, where X is an instruction of a tested C++ program.

$T = <$ X | for all X, where X's are in a feasible path executed for some input and $X \in P >$

Definition 4. Let TF_{name} be a function trajectory, i.e., a feasible path of a function name that has actually been executed for some input. TF_{name} is a sublist of T. If a trajectory of length m is denoted by $T = <X_1, X_2, ..., X_m>$, then the function trajectory name is denoted by $TF_{name} = <X_i, X_{i+1}, ..., X_k>$, where $X_i, X_{i+1}, ..., X_k$ are a list of the instruction X's which are in the scope of a given function F_{name}, where i denotes the position of entry node and k denotes the position of ending node of the function name, ($1 \le i < k$, and $i < k \le m$).

$TF_{name} = <$ X | for all X, where X's are in a feasible path executed for some input, $X \in F_{name}$, and $X \in T >$

Definition 5. Let *action* be pair(X,p), i.e., instruction X at position p, which will be replaced by X^p for brevity and ease of understanding [5]. An action X^p is a *test action* if X is a test instruction such as **while** or **for**.

Definition 6. Let M(T) be a set of actions in a given trajectory T, where $M(T) = \{ X^p :$ instruction X at position p in trajectory T $\}$ [5].

Definition 7. Let $M(TF_{name})$ be a set of actions in a given function of a given trajectory TF_{name}, where $M(TF_{name}) = \{ X^p :$ instruction X at position p in trajectory $TF_{name} \}$. $M(TF_{name})$ is a subset of M(T).

Definition 8. Let C be a slicing criterion, which is the specification for a particular behavior of interest. A slicing criterion can be expressed as the values of some set of variables at some set of statements [10]. If we let T be the trajectory of program P on input x, a slicing criterion of program P executed on x can be defined as a triple $C = (x, I^q, V)$, where I^q is an action in T and V is a subset of variables in P [5].

Definition 9. Let $D(X^p)$ be the set of variables that are defined in action X^p, where $X^p \in M(T)$.

Let $DF_{name}(X^p)$ be the set of variables that are defined in action X^p, where $X^p \in M(TF_{name})$.

Definition 10. Let $U(X^p)$ be the set of variables that are used in action X^p, where $X^p \in M(T)$.

Let $UF_{name}(X^p)$ be the set of variables that are used in action X^p, where $X^p \in M(TF_{name})$.

Definition 11. Let $LF_{name}(X^p)$ be a set of variables and C++ preprocessors that are declared as a local declaration in function name.

Definition 12. Let DU be a Definition-Use Relation, a relation in which one action assigns a value to an item of data and the other action uses that value [5]. Instead of using M(T) as Korel and Laski did, $M(TF_{name})$ was used in this work in order to compute a slice from functions or classes.

Let $M(TF_{name})$ be a set of actions in a given trajectory TF_{name}. DUF_{name}, a Definition-Use-$Function_{name}$ Relation, is a binary relation on $M(TF_{name})$ defined as follows:

Let $TF_{name} = <X_i, X_{i+1}, ..., X_t, ..., X_k>$,

$X^p \; DUF_{name} \; Y^t$, $i \le p < t$, iff there exists a variable v

such that (1) $v \in UF_{name}(Y^t)$, and

(2) X^p is the last definition of v at t

where, the last definition X^p of variable v at t is the action which last assigned a value to v when t was reached on trajectory TF_{name}.

Definition 13. Let LDR be a Local-Declaration Relation, a relation in which one action declares a variable and the other action defines or uses that variable.

Let $M(TF_{name})$ be a set of actions in a given trajectory TF_{name}. $LDRF_{name}$, a Local-Declaration$_{name}$ Relation, is a binary relation on $M(TF_{name})$ defined as follows:

Let $TF_{name} = <X_i, X_{i+1}, ..., X_t, ..., X_k>$,

X^p $LDRF_{name}$ Y^t, $i \leq p < t$, iff there exists a variable v

such that (1) $v \in UF_{name}(Y^t) \cup DF_{name}(Y^t)$, and

(2) X^p is the action where variable v was declared in trajectory TF_{name}.

Definition 14. Let TC be a Test-Control Relation, capturing the effect between test actions and actions that have been chosen to execute by these test actions [5]. Instead of using M(T) as Korel and Laski did, $M(TF_{name})$ was used in this work in order to compute a slice from functions or classes. Let $M(TF_{name})$ be a set of actions in a given trajectory TF_{name}. TCF_{name}, a Test-Control-Function$_{name}$ Relation, is a binary relation on $M(TF_{name})$ defined as follows:

Let $TF_{name} = <X_i, X_{i+1}, ..., X_t, ..., X_k>$,

X^p TCF_{name} Y^t, $i \leq p < t$, iff

(1) Y is in the scope of influence of X, and

(2) for all k, $p < k < t$, $T(k) \neq X$

where, the scope of influence is defined as follows.

(1) **if** X **then** B1 **else** B2; Instruction Y is in the scope of influence of X iff Y is in B1 or B2.

(2) **while** X **do** B; Instruction Y is in the scope of influence of X iff Y is in B.

(3) **do** B **while** X; Instruction Y is in the scope of influence of X iff Y is in B.

(4) **case** X **do** B; Instruction Y is in the scope of influence of X iff Y is in B.

(5) **for** X **do** B; Instruction Y is in the scope of influence of X iff Y is in B.

(6) **function** X **do** B; Instruction Y is in the scope of influence of X iff Y is in B.

Definition 15. Let IRF_{name} be an Identity Relation in Function$_{name}$, then X^p IRF_{name} Y^t, iff X = Y is the identity relation IRF_{name} on $M(Front(TF_{name},q))$, where $Front(TF_{name},q)$ is a sublist of TF_{name} consisting of the first q elements of TF_{name}, where $TF_{name} = <X_i, X_{i+1}, ..., X_t, ..., X_q, ..., X_k>$ denotes a function trajectory, q is a position in TF_{name}, $1 \leq i < t$, and $t < q \leq k$.

Definition 16. Figures 1 and 2 present a part of the trajectory of FuncA(int i) and FuncB(int j), where called FuncA(int i) is called by calling FuncA(5) at X^{n+1}, and called FuncB(int j) is called by calling FuncB(2) at X^{l+1}. From Figures, we find that $T = <..., X^{i-2}, X^{i-1}, X^i, X^{i+1}, X^{i+2}, ..., X^j, X^{j+1}, ..., X^k, X^{k+1}, ..., X^l, X^{l+1}, ..., X^m, ..., X^n, X^{n+1}, X^{n+2}, ...>$, where i<j<k, l<m<n and X is any statement in a program P, $TF_{FuncA} = < X^i,$

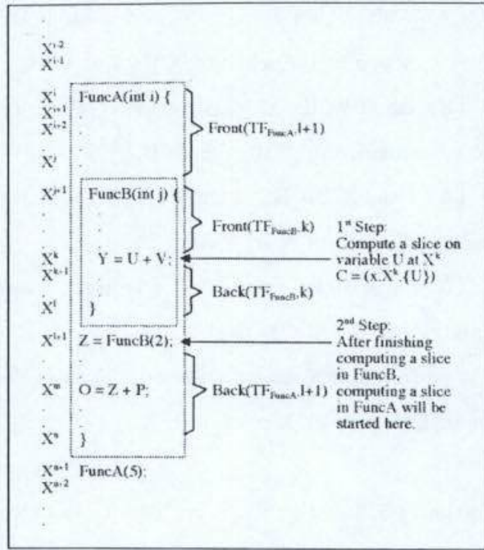

Fig. 1 Illustrate Called-to-Calling

X^{i+1}, X^{i+2}, ..., X^j, X^{l+1}, ..., X^m, ..., X^n>, and TF_{FuncB} = <X^{j+1}, ..., X^k, X^{k+1}, ..., X^l>. Functions FuncA (int i) at X^i and FuncB(int j) at X^{j+1} are called a called function instruction. An action X^p is a *called action* if X is a called function instruction. FuncA(5) at X^{n+1} and FuncB(2) at X^{l+1} are called calling function instructions. An action X^p is a *calling action* if X is a calling function instruction.

Called-to-Calling occurs when a slice is computed

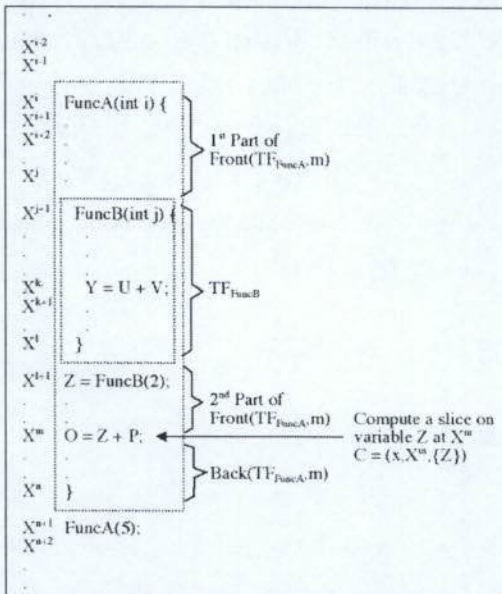

Fig. 2 Illustrate Calling-to-Called

from a called action first and then from a calling action. For example, in Figure 1, suppose one needs to find a slice of variable U at X^k. The process starts from X^k (which is in the scope of influence of called function FuncB(int j), which is called by calling function FuncB(2) at X^{l+1}), and then X^{j+1}, X^{l+1}, respectively. We find that called action X^{j+1} comes before calling action X^{l+1}.

Calling-to-Called occurs when a slice is computed from a calling action first and then from a called action. For example, in Figure 2, suppose that one needs to find a slice of variable Z at X^m. The process starts from X^m, and then X^{l+1} (since Z is last defined at X^{l+1} and used at X^m) and then X^{j+1} (since called FuncB(int j) is called by calling FuncB(2)), respectively. We find that calling action X^{l+1} comes before called action X^{j+1}.

Modified from Korel and Laski's approach [5], let TF_{name} = <X_i, X_{i+1}, X_{i+2}, ..., X_k> be a trajectory of function name, and q be a position in TF_{name}, $i \leq q$ k. Then Front(TF_{name},q) is a sublist <X_i, X_{i+1}, ..., X_q> and Back(TF_{name},q) is a sublist <X_{q+1}, X_{m+2}, ..., X_k> as shown in Figures 1 and 2. All Back (TF_{name},q)'s can be ignored in computing a slice. Just Front (TF_{name},q) must be concentrated on.

Let A and B be two functions, where function A calls function B. Therefore, a slice can be computed in two different ways as follow.

1) Called-to-Calling

$$\text{Total slice}_{AB} = \text{Slice}_B \cup \text{Slice}_A$$

where

(1) $Slice_B$ is a slice computed based on Front(TF_B,k) and slicing criterion C = (x, X^k, V)

(2) $Slice_A$ is a slice computed based on Front(TF_A,l+1) and used variables at calling action X^{l+1}, U(X^{l+1}).

2) Calling-to-Called

Total $slice_{AB}$ = $Slice_A \cup TF_B$

where

(1) $Slice_A$ is a slice computed based on $Front(TF_A, m)$ and slicing criterion $C = (x, X^m, V)$.

(2) TF_B is a function trajectory of function B.

Let $Calling(X^p)$ be a set of calling functions that are used to call a called function in action X^p, where $X^p \in M(T)$.

Let $Called(X^p)$ be a set of called functions that are called by a calling function in action X^p, where $X^p \in M(T)$.

Let EI be a Called-to-Calling Relation between called and calling functions. Let M(T) be a set of actions in a given trajectory T of length m. EI is a binary relation on M(T) defined as follows:

Let $T = \langle X_1, X_2, ..., X_t, ..., X_m \rangle$,

X^p EI Y^t, $t \leq p < m$, iff there exists function f such that

(1) a called function $f \in Called(Y^t)$,

(2) a calling function $f \in Calling(X^p)$, and

(3) X^p is the calling action, where the calling function f at p calls a called function f at t

Let IE be a Calling-to-Called Relation between called and calling functions. Let M(T) be a set of actions in a given trajectory T of length m. IE is a binary relation on M(T) defined as follows:

Let $T = \langle X_1, X_2, ..., X_t, ..., X_m \rangle$,

X^p IE Y^t, $1 \leq p < t$, iff there exists function f such that

(1) a calling function $f \in Calling(Y^t)$,

(2) a called function $f \in Called(X^p)$, and

(3) X^p is the called action where the called function f at p is called by a calling function f at t

Definition 17. To find the slicing set S_c, we first find the set A^0 of all actions that have direct influence on V at q and on action I^q. A^0 is defined as follows [5].

$$A^0 = LD(q,V) \cup LT(I^q) \cup I^q$$

where LD(q,V) is the set of last definitions of variables in V at the execution position q, and $LT(I^q)$ is a set of test actions that have Test-Control influence on I^q.

We will find S_c iteratively, as the limit of a sequence $S^0, S^1, ..., S^n$, $0 \leq n < q$, which is defined as follows.

$$S^0 = A^0 \text{ and } S^{i+1} = S^i \cup A^{i+1}$$

where

$A^{i+1} = \{ X^p \in M(TF_{name}) : 1 \leq p < q,$

(1) $X^p \notin S^i$, and

(2) there exists $Y^t \in S^i$, $t < q$, $X^p\ Z\ Y^t \}$

where $Z = DU \cup TC \cup IR \cup LDR$

Finally, we can get the slice from the following definition.

$$S_c = S^k$$

where S^k is the limit of the sequence $\{S^i\}$.

Definition 18. Let FN(q) be a string of function name such that X^q, X is in the scope of influence.

Definition 19. Let G(X) be a set of variables and preprocessors that are declared as a part of global declaration. G(X) is computed from the source program, not from a trajectory path.

Definition 20. Let VDU(FunctionName) be a set of variables that are used, UF_{name}, and defined, DF_{name}, in a given function name.

Definition 21. In order to find the scope of influence of each instruction, *variable scope*, VS, and *control scope*, CS, are used as defined bellow.

1. Variable scope, VS, gives the information that the variables that used or defined in each instruction were declared at what instructions.

Let X_{DCL} be an instruction that declared variables such as "int I;".

Let X_{DU} be an instruction that used or defined the variables declared by X_{DCL}, where variables that are used or defined are in the scope of influence of the variables that are declared in X_{DCL}. For example, "I=I+1;", which is the first I is defined and the second I is used both are declared by "int I;".

Then we get $VS(X_{DU})$, a variable scope relation at X_{DU}, which is a set of instructions X_{DCL}, where X_{DU} is in the scope of influence of X_{DCL}.

2. Control scope, CS, gives information about instructions that are in the scope of influence of control instructions such as test statements, functions, and classes. For calculation of the scope of influence of each statement, the me_too set is used [6].

Let X be an instruction, the me_too is a set of instructions that are in the scope of influence of instruction X.

Due to the complexity of the C++ language and in order for C++Debug to be applicable to programs containing functions, classes, namespaces, unions, structures, and preprocessors (a separate first step in compilation, e.g., #include, #define, or #if), the me_too set was modified according to the rules shown in Fig. 4

Instruction (X)	Prototype	Called	Calling	D	U	DCL	VS	CS
1: #include <iostream>						include		
2:								
3: class Compute {	Compute(01)							26
4: private:								
5: int Max;						Max(02)		3
6: float Num[4];						Num(03)		3
7:								
8: public:								
9: Compute(int M, float *N) {		Compute(01)				M(04)		3,1
10: Max = M;				Max(02)	M(04)	N(05)	5, 9	4
11: cout<<"allocate mem"<<endl;					cout(07) endl(08)			9
12: for(int I=0; I<Max; ++I)				Num(03)	Max(02) N(05)		5,6,9	9
Num[I] = N[I];				I(09)	I(09)	I(09)		9
13:								9
14: }								
15:								9
16: float Sum(void) {								
17: float Tsum = 0;		Sum(10)						3,2
18: for(int I=0; I<Max; ++I)				Tsum(11	Max(02) Num(03)	Tsum(11)	5,6,17	1
Tsum = Tsum + Num[I];				)	Tsum(11) I(12)	I(12)		16
19:				I(12)				16
20: return Tsum;					Tsum(11)		17	
21: }								
22:								16
23: float Avg(void) {								16
24: return Sum()/(Max + 1);		Avg(13)	Sum(10)		Max(02)		5	
25: }								3,2
26: };								5
27:								23
28: main () {								23
29: int Max = 4;		main(14)				Max(15)		3
30: float Num[4] = {10.0, 20.0, 15.0,5.0};						Num(16)		
31: Compute A(Max, Num);			Compute(01)		Max(15) Num(16)	A (17)	9,29,30	34
32: cout<<A.Sum()<<endl;			Sum(19)		cout(07) endl(08) A(17)		31	28
33: cout<<A.Avg()<<endl;			Avg(20)		cout(07) endl(08) A(17)		31	28
34: }								28
								28
								28
								28
								28

Fig. 3 The Prototype, Called, Calling, D, U, DCL, VS, and CS sets for the program depicted in Fig. 8

1. For any straight-line instruction, the CS set must contain:
   1.1 Instruction of which it is in the scope of influence
2. For any control instruction, the CS set must contain:
   2.1 Instruction of which it is in the scope of influence
   2.2 Instruction representing the beginning of the scope of influence
   2.3 Instruction representing the end of the scope of influence
3. In case of functions, the CS set of that instruction must contain
   3.1 Instruction of which it is in the scope of influence
   3.2 Instruction representing the beginning of t he scope of influence
   3.3 Instruction representing the end of the scope of influence
4. In case of classes, structures, unions, and namespaces, the CS set of that instruction must contain
   4.1 Instruction representing the beginning of the scope of influence
   4.2 Instruction representing the end of the scope of influence

Fig. 4 Rules for computing the CS set

and will still be called the control scope, CS, set.

Based on the rules in Figure 3, Figures 4 shows an example of computing the CS set of a tested program that computes the the sum and average of a set of numbers in Figure 8.

To find the final slicing set F_s with scope, we first find the set S^0 of all instructions that sliced from the tested program P based on slicing criterion $C(x,I^q,V)$. S^0 is defined as follows.

$S^0 = S_c$

where S_c is a slicing set defined in Definition 17.

We will find F_s iteratively, as the limit of a sequence $F^0, F^1, ..., F^n$, $0 \leq i < n$, n = length of program P, which is defined as follows.

$F^0 = S^0$ and $F^{i+1} = F^i \cup S^{i+1}$

where

S^{i+1} = { $X \in P : 1 \leq X < n$, n = length of programP,

(1) $X \notin F^i$, and

(2) there exists $Y \in F^i$, $X \in Z(Y)$ }

where $Z = VS \in CS$

Finally, we can get the final slice with scope from the following definition.

$F_s = F^k$

where F^k is the limit of the sequence $\{F^i\}$.

3. Algorithms

Figure 5 presents the algorithm designed and implemented for C++Debug. The algorithm is separated into 4 parts: *Datastructures, Initialize, PASS I, and PASS II.* The Initialize part is used to initialize variables, files, etc., when the program starts.

The objectives of PASS I are to create databases and to create a trajectory T. All computations in PASS I are determined based on a source code program. The databases are used to collect the necessary information used in PASS II such as *Symbol Table, List of Reserved Words, List of Basic Types, Types, Identifiers Information, Scope of Influent,* etc. The trajectory T is created by a tool named cpptrace.

PASS II uses the information in each database and the trajectory T from PASS I to compute a set of slices. First, a slicing criterion comprising of a set of variables V and position q is entered. After that, each slice of each variable in set V at position q is computed one by one. The process starts with finding a slice inside the function where position q is at, until finished. Then the algorithm goes to its calling function and starts to find a slice in this calling function again. The process is repeated until the final slice of the calling function named main() is computed. Clearly, the slice of each variable in the set V is computed based on all functions that related to each variable in the set V starting from the function where position q is at, its calling function, ..., and end at

```
Datastructures
Begin
 Initialize():                  // initialize files, variables, etc.;
  // PASS I
  // compute from source code program P
 Create_Information_Database(P);
                                // compute trajectory code T
                                // see Definition 3
                                // by using tool named cpptrace
 T = gen_T(P);
  // PASS II
  // compute slices from trajectory T
 I = 1;
                                // slicing criterion at position q
                                // on a set of variables V
 C = Read_Criterion();          // see Definition 8

 while (C.V ≠ "Exit" ) {                 // to check not exit the program
  S[0] = {};                    //  clear temporary slice storage
  while (C.q ≥ 1 and C.q ≤ MaxTraj) {   // Is C.q a valid number
                                        // in the trajectory T ?
STEP I:                         // compute slice in called function
   S[0] = S[0] ∪ Compute_Slice_in_Function_Name(C);
STEP II:
   if (FN(C.q)) == "main" )         // check called-to-calling function
   then                         // finish computing a slice for each variable
    break;                      // then break the loop
   else
    X^p EI Y^t; Y^t ∈ S[0]      // get a new position of its calling function
         C.q = X^p              // see Definition 16
  }
 STEP III:                      // add scope of influent to complete each
slice
      Slice[I] = Add_Scope_of_Influent(S[0])

  I++
  C = Read_Criterion();         // get a new slicing criterion at
                                // position q on a new variables V
 }
     // finally we get each Slice[I] for each variable V[I]
     // at a specific position q's
end
```

Fig. 5 Algorithm to compute a set of slices

```
Step 1.1:  // function to compute a slice without its scope of
influence

  Compute_Slice_in_Function_Name(SliceCriterion C) {
          // get function name, see Definition 18
    name  = FN(C.q);
          // compute a sublist function trajectory, see Definition 4
    TFname      = SubT(LF(C));
          // compute defined var., see Definition 9
    DFname      = ComputeDFname(TFname);
          // compute used var., see Definition 10
    UFname      = ComputeUFname(TFname);
          // compute defined used rel., see Def. 12
    DUFname   = ComputeDUFname(TFname);
          // compute test control rel., see Def. 14
    TCFname   = ComputeTCFname(TFname);
          // compute identity rel., see Definition 15
    IRFname    = ComputeIRFname(TFname);
          // compute local declaration relation see Definition 13
    LDRFname = ComputeLDRFname(TFname);

Step 1.2:   // compute a slice in a function name, see Definition 17
    S = ComputeSlice(DUFname, TCFname, IRFname,
          LDRFname, C);

Step 1.3:                     // see Definition 16
    if(X^p IE Y^t; Y^t ∈ S)                              // check Calling-
to-Called function
       name = FN(p)     // get calling function name
       S = S ∪ TF_name  // where IE, a Calling-to-Called function,
                        // is an element of S
    return (S);
  }
```

Fig. 6 Algorithm to compute a slice of each function

function main(). Compute_scope_of_influence

```
                  // function to compute the scope of influence of a slice
 Add_Scope_of_Influent(array[1..n] of set[action] S) {
                                        // Add scope of influence to a slice,
  S = Var_Control_Scope (S);  // see Definition 21
  return S;
 }
```

Fig. 7 Function to compute the scope of influence of a slice

(C) makes the final slice completed by adding some statements that may govern each statement in the slice.

4. Examples: How to Compute a Slice

The program in Figure 8 computes the sum and average of integers. In this example, variable Max is 4 and the array called Num contains 10.0, 20.0, 15.0, and 5.0. Upon completion of program execution, the program should yield one results as 12.5. However, this program contains an error in line 24. Rather than return Sum()/Max, the program computes return Sum()/(Max+1), thus yielding an error (Avg = 10.0 instead of 12.5). To localize such an error, program slicing and dicing techniques can be used. The trajectory of the program in Figure 8 is shown in Figure 9.

Example 1. Consider trajectory T in Figure 9. Using the criterion C = (x, 33^{30}, {Avg}), we have x = (Max,Num) = (3, (10.0, 20.0, 15.0, 5.0)).

The step-by-step trace of the algorithm in Figure 5

```
1: #include <iostream>
2:
3: class Compute {
4:   private:
5:     int Max;
6:     float Num[4];
7:
8:   public:
9:     Compute(int M, float *N) {
10:      Max = M;
11:      cout<<"allocate mem"<<endl;
12:      for(int I=0; I<Max; ++I)
         Num[I] = N[I];
13:
14:    }
15:
16:    float Sum(void) {
17:      float Tsum = 0;
18:      for(int I=0; I<Max; ++I)
         Tsum = Tsum + Num[I];
```

```
19:
20:     return Tsum;
21:   }
22:
23:   float Avg(void) {
24:     return Sum()/(Max + 1);
25:   }
26: };
27:
28: main () {
29:   int Max = 4;
30:   float Num[4] = {10.0, 20.0, 15.0, 5.0};
31:   Compute A(Max, Num);
32:   cout<<A.Sum()<<endl;
33:   cout<<A.Avg()<<endl;
34: }
```

Fig. 8 A program for calculating the sum and average of a set of numbers

```
28^1  main() {
29^2    int Max = 4;
30^3    float Num[4] = {10.0, 20.0, 15.0, 5.0};

9^4   Compute (int M, float *N) {
10^5      Max = M; allocate mem
11^6      cout<<"allocate mem"<<endl;
12^7      for(int I=0; I<Max; ++I)
            Num[0] = N[0];
12^8      for(int I=0; I<Max; ++I)
            Num[1] = N[1];
12^9      for(int I=0; I<Max; ++I)
            Num[2] = N[2];
12^10     for(int I=0; I<Max; ++I)
            Num[3] = N[3];
14^11   }

31^12   Compute A(Max, Num);

16^13   float Sum(void) {
17^14     float Tsum = 0;
18^15     for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[0];
18^16     for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[1];
18^17     for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[2];
18^18     for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[3];
20^19     return Tsum; 50

32^20   cout<<A.Sum()<<endl;

23^21   float Avg(void) {
24^22     return Sum()/(Max + 1);

16^23   float Sum(void) {
17^24     float Tsum = 0;
18^25     for(int I=0; I<Max; ++I)
          Tsum = Tsum + Num[0];
18^26     for(int I=0; I<Max; ++I)
          Tsum = Tsum + Num[1];
18^27     for(int I=0; I<Max; ++I)
          Tsum = Tsum + Num[2];
18^28     for(int I=0; I<Max; ++I)
          Tsum = Tsum + Num[3];
20^29     return Tsum; 10

33^30   cout<<A.Avg()<<endl;
34^31 }
```

$$T = \langle 28^{1},29^{2},30^{3},9^{4},10^{5},11^{6},12^{7},12^{8},12^{9},12^{10},14^{11},31^{12},16^{13},17^{14},18^{15},18^{16},18^{17},18^{18},20^{19},32^{20},23^{21},24^{22},16^{23},17^{24},18^{25},18^{26},18^{27},18^{28},20^{29},33^{30},34^{31}\rangle$$

$$TF_{Main} = \langle 28^{1},29^{2},30^{3},31^{12},32^{20},33^{30},34^{31}\rangle$$

$$TF_{Compute} = \langle 9^{4},10^{5},11^{6},12^{7},12^{8},12^{9},12^{10},14^{11}\rangle$$

$$TF_{Sum(1)} = \langle 16^{13},17^{14},18^{15},18^{16},18^{17},18^{18},20^{19}\rangle$$

$$TF_{Sum(2)} = \langle 16^{23},17^{24},18^{25},18^{26},18^{27},18^{28},20^{29}\rangle$$

$$TF_{Avg} = \langle 23^{21},24^{22}\rangle$$

Fig. 9 The trajectory of the program from Figure 8 on input data Max = 4, Num = (10.0, 20.0, 15.0, 5.0)

follows.

Step 1:

S[0] = S [0] ∪

Compute_Slice_in_Function_Name(C)

Step 1.1:

FN(C.q) = FN (30) = "main"

// therefore compute slice in function "main"

compute TF_{main} // as shown in Figure 9

compute LDF_{main} // as shown in Figure 10

compute DUF_{main} // as shown in Figure 10

compute TCF_{main} // as shown in Figure 10

$DUF_{Main} = \{\}$ $\quad$ $LDRF_{Main}(29^{2}) = \{31^{12}\}$

$TCF_{Main} = \{\}$ $\quad$ $LDRF_{Main}(30^{3}) = \{31^{12}\}$

$IRF_{Main} = \{\}$

Fig. 10 The DUF_{Main}, TCF_{Main}, LDF_{Main}, and IRF_{Main} relations that are called by 32^{20} for the trajectory depicted in Figure 9

$DUF_{Compute}(10^{5}) = \{12^{7}\}$ $\quad$ $IRF_{Compute}(12^{7}) = \{12^{8},12^{9},12^{10}\}$

$DUF_{Compute}(12^{7}) = \{\}$ $\quad$ $IRF_{Compute}(12^{8})$

$= \{12^{7},12^{9},12^{10}\}$

$IRF_{Compute}(12^{9}) = \{12^{7},12^{8},12^{10}\}$

$TCF_{Compute} = \{\}$ $\quad$ $IRF_{Compute}(12^{10})$

$= \{12^{7},12^{8},12^{9}\}$

$LDRF_{Compute}(9^{4}) = \{10^{5},12^{7},12^{8},12^{9},12^{10}\}$

Fig. 11 The $DUF_{Compute}$, $TCF_{Compute}$, $LDF_{Compute}$, and $IRF_{Compute}$ relations that are called by 32^{20} for the trajectory depicted in Figure 9

$DUF_{Sum}(18^{15}) = \{18^{16}\}$ $\quad$ $IRF_{Sum}(18^{15}) = \{18^{16},18^{17},18^{18}\}$

$DUF_{Sum}(18^{16}) = \{18^{17}\}$ $\quad$ $IRF_{Sum}(18^{16}) = \{18^{15},18^{17},18^{18}\}$

$DUF_{Sum}(18^{17}) = \{18^{18}\}$ $\quad$ $IRF_{Sum}(18^{17}) = \{18^{15},18^{16},18^{18}\}$

$DUF_{Sum}(18^{18}) = \{20^{19}\}$ $\quad$ $IRF_{Sum}(18^{18}) = \{18^{15},18^{16},18^{17}\}$

$TCF_{Sum} = \{\}$

$LDRF_{Sum}(17^{14}) = \{18^{15},18^{16},18^{17},18^{18},20^{19}\}$

Fig. 12 The DUF_{Sum}, TCF_{Sum}, LDF_{Sum}, and IRF_{Sum} relations that are called by 32^{20} for the trajectory depicted in Figure 9

$DUF_{Sum}(18^{25}) = \{18^{26}\}$ $\quad$ $IRF_{Sum}(18^{25}) = \{18^{26},18^{27},18^{28}\}$

$DUF_{Sum}(18^{26}) = \{18^{27}\}$ $\quad$ $IRF_{Sum}(18^{26}) = \{18^{25},18^{27},18^{28}\}$

$DUF_{Sum}(18^{27}) = \{18^{28}\}$ $\quad$ $IRF_{Sum}(18^{27}) = \{18^{25},18^{26},18^{28}\}$

$DUF_{Sum}(18^{28}) = \{20^{29}\}$ $\quad$ $IRF_{Sum}(18^{28}) = \{18^{25},18^{26},18^{27}\}$

$TCF_{Sum} = \{\}$

$LDRF_{Sum}(17^{24}) = \{18^{25},18^{26},18^{27},18^{28},20^{29}\}$

Fig. 13 The DUF_{Sum}, TCF_{Sum}, LDF_{Sum}, and IRF_{Sum} relations that are called by 24^{22} for the trajectory depicted in Figure 9

$DUF_{Avg} = \{\}$ $\quad$ $TCF_{Avg} = \{\}$ $\quad$ $IRF_{Avg} = \{\}$

Fig. 14 The DUF_{Avg}, TCF_{Avg}, and IRF_{Avg} relations for the trajectory depicted in Figure 9

compute IRF_{main} // as shown in Figure 10

compute $LDRF_{main}$ // as shown in Figure 10

Step 1.2:

S = ComputeSlice(DUF_{main}, TCF_{main}, IRF_{main}, $LDRF_{main}$, C)

Since C = (x, 33^{30}, {Avg}) // given

LD(30, {Avg}) = {}, LT(33^{30}) = {28^1}, I^q = 33^{30}

$A^0 = \{28^1, 33^{30}\}$, $S^0 = \{28^1, 33^{30}\}$,

$A^1 = \{31^{12}\}$, $S^1 = \{28^1, 31^{12}, 33^{30}\}$,

$A^2 = \{29^2, 30^3\}$, $S^2 = \{28^1, 29^2, 30^3, 31^{12}, 33^{30}\}$,

$A^3 = \{\}$, $S^3 = \{28^1, 29^2, 30^3, 31^{12}, 33^{30}\}$,

$S_c = S^3 = \{28^1, 29^2, 30^3, 31^{12}, 33^{30}\}$.

Step 1.3: Check Calling-to-Called functions

Yes, since {23^{21}} IE {33^{30}}, and {9^4} IE {31^{12}},

FN(4) = "Compute", and FN(21) = "Avg" ,

$S_c = S_c\ TF_{Compute} \cup TF_{Avg}$,

$TF_{Compute} = < 9^4, 10^5, 11^6, 12^7, 12^8, 12^9, 12^{10}, 14^{11} >$,

$TF_{Avg} = < 23^{21}, 24^{22} >$,

$S_c = \{28^1, 29^2, 30^3, 9^4, 10^5, 11^6, 12^7, 12^8, 12^9, 12^{10}, 14^{11}, 31^{12}, 23^{21}, 24^{22}, 33^{30}\}$,

since {16^{23}} IE {24^{22}},

FN(23) = "Sum",

$S_c = S_c \cup TF_{Sum}$,

$TF_{Sum} = < 16^{23}, 17^{24}, 18^{25}, 18^{26}, 18^{27}, 18^{28}, 20^{29} >$,

Finally, we get $S[0] = S[0] \cup S_c$

$= \{28^1, 29^2, 30^3, 9^4, 10^5, 11^6, 12^7, 12^8, 12^9, 12^{10}, 14^{11}, 31^{12}, 23^{21}, 24^{22}, 16^{23}, 17^{24}, 18^{25}, 18^{26}, 18^{27}, 18^{28}, 20^{29}, 33^{30}\}$.

Step 2: Check for more Called-to-Calling functions since FN(30) = "main" then no more calling functions and break.

Step 3: Add scope of influence

Slice[1] = Add_Scope_of_Influence(S[0])

Let $F^0 = S_0 = S[0]$

F^0 = {9, 10, 11, 12, 14, 16, 17, 18, 20, 23, 24, 28, 29, 30, 31, 33}

S^0 = {9, 10, 11, 12, 14, 16, 17, 18, 20, 23, 24, 28, 29, 30, 31, 33},

F^1 = {1, 3, 5, 6, 21, 25, 34},

S^1 = {1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 23, 24, 25, 28, 29, 30, 31, 33, 34},

F^2 = {26},

S^2 = {1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 23, 24, 25, 26, 28, 29, 30, 31, 33, 34},

F^3 = {},

S^3 = {1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 23, 24, 25, 26, 28, 29, 30, 31, 33, 34},

Slice[1] = S^3 = {1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 23, 24, 25, 26, 28, 29, 30, 31, 33, 34}.

And finally, the dynamic slice is shown in Figure 15.

Example 2. Consider trajectory T in Figure 9. Using the criterion C = (x, 32^{20}, {Sum}), we have x = (Max,Num) = (3, (10.0, 20.0, 15.0, 5.0)).

The step-by-step trace of the algorithm in Figure 5 follows.

Step 1:

S[0] = S [0] $\cup$ Compute_Slice_in_Function_Name(C)

Step 1.1:

FN(C.q) = FN (20) = "main"

// therefore compute slice in function "main"

compute TF_{main} // as shown in Figure 9

compute LDF_{main} // as shown in Figure 10

compute DUF_{main} // as shown in Figure 10

compute TCF_{main} // as shown in Figure 10

compute IRF_{main} // as shown in Figure 10

compute $LDRF_{main}$ // as shown in Figure 10

Step 1.2:

Compute S = ComputeSlice(DUF_{main}, TCF_{main}, IRF_{main}, $LDRF_{main}$, C) .

Since $C = (x, 32^{20}, \{Sum\})$ // given

$LD(20, \{Avg\}) = \{\}$, $LT(32^{20}) = \{28^1\}$, $I^q = 32^{20}$

$A^0 = \{28^1, 32^{20}\}$, $S^0 = \{28^1, 32^{20}\}$,

$A^1 = \{31^{12}\}$, $S^1 = \{28^1, 31^{12}, 16^{13}, 32^{20}\}$,

$A^2 = \{29^2, 30^3\}$, $S^2 = \{28^1, 29^2, 30^3, 31^{12}, 32^{20}\}$,

$A^3 = \{\}$, $S^3 = \{28^1, 29^2, 30^3, 31^{12}, 32^{20}\}$,

$S_c = S^3 = \{28^1, 29^2, 30^3, 31^{12}, 32^{20}\}$.

Step 1.3: Check Calling-to-Called functions

Yes, since $\{9^4\}$ IE $\{31^{12}\}$, and $\{16^{13}\}$ IE $\{32^{20}\}$,

FN(4) = "Compute", and FN(13) = "Sum",

$S_c = S_c \cup TF_{Compute} \cup TF_{Sum}$,

$TF_{Compute} = < 9^4, 10^5, 11^6, 12^7, 12^8, 12^9, 12^{10}, 14^{11} >$,

$TF_{Sum} = < 16^{23}, 17^{24}, 18^{25}, 18^{26}, 18^{27}, 18^{28}, 20^{29} >$,

$S_c = \{28^1, 29^2, 30^3, 9^4, 10^5, 11^6, 12^7, 12^8, 12^9, 12^{10}, 14^{11}, 31^{12}, 16^{23}, 17^{24}, 18^{25}, 18^{26}, 18^{27}, 18^{28}, 20^{29}, 32^{20}\}$,

Finally, we get $S[0] = S[0] \cup S_c$

$= \{28^1, 29^2, 30^3, 9^4, 10^5, 11^6, 12^7, 12^8, 12^9, 12^{10}, 14^{11}, 31^{12}, 16^{23}, 17^{24}, 18^{25}, 18^{26}, 18^{27}, 18^{28}, 20^{29}, 32^{20}\}$.

Step 2: Check for more Called-to-Calling functions

since FN(20) = "main" then no more calling function and break

Step 3: Add scope of influence

Slice[1] = Add_Scope_of_Influence(S[0])

Let $F^0 = S_0 = S[0]$

$F^0 = \{9, 10, 11, 12, 14, 16, 17, 18, 20, 28, 29, 30, 31, 32\}$

$S^0 = \{9, 10, 11, 12, 14, 16, 17, 18, 20, 28, 29, 30, 31, 32\}$,

$F^1 = \{1, 3, 5, 6, 21, 34\}$,

$S^1 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 28, 29, 30, 31, 32, 34\}$,

$F^2 = \{26\}$,

$S^2 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 26, 28, 29, 30, 31, 32, 34\}$,

$F^3 = \{\}$,

$S^3 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 26, 28, 29, 30, 31, 32, 34\}$,

$Slice[1] = S^3 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 26, 28, 29, 30, 31, 32, 34\}$.

```
1: #include <iostream>
3: class Compute {
4:   private:
5:     int Max;
6:     float Num[4];
8:   public:
9:     Compute(int M, float *N) {
10:      Max = M;
11:      cout<<"allocate mem"<<endl;
12:      for(int I=0; I<Max; ++I)
         Num[I] = N[I];
14:    }
16:    float Sum(void) {
17:      float Tsum = 0;
18:      for(int I=0; I<Max; ++I)
         Tsum = Tsum + Num[I];
20:      return Tsum;
21:    }
23:    float Avg(void) {
24:      return Sum()/(Max + 1);
25:    }
26: };
28: main () {
29:   int Max = 4;
30:   float Num[4] = {10.0, 20.0, 15.0, 5.0};
31:   Compute A(Max, Num);
33:   cout<<A.Avg()<<endl;
34: }
```

Fig. 15 A dynamic program slice computed based on variable Avg in line 33 of the program in Figure 8

```
1: #include <iostream>
3: class Compute {
4:   private:
5:     int Max;
6:     float Num[4];
8:   public:
9:     Compute(int M, float *N) {
10:      Max = M;
11:      cout<<"allocate mem"<<endl;
12:      for(int I=0; I<Max; ++I)
         Num[I] = N[I];
14:    }
16:    float Sum(void) {
17:      float Tsum = 0;
18:      for(int I=0; I<Max; ++I)
         Tsum = Tsum + Num[I];
20:      return Tsum;
21:    }
26: };
28: main () {
29:   int Max = 4;
30:   float Num[4] = {10.0, 20.0, 15.0, 5.0};
31:   Compute A(Max, Num);
32:   cout<<A.Sum()<<endl;
34: }
```

Fig. 16 A dynamic program slice computed based on variable Sum in line 32 of the program in Figure 8

```
23:    float Avg(void) {
24:      return Sum()/(Max + 1);
25:    }
```

Fig. 17 The final program segment after slicing and dicing

And finally, the dynamic slice is shown in Figure 16.

5. Dicing Procedures

Dicing [6][7][8] is the process of identifying a set of statements likely to contain an error. A dice is determined as follows:

1. Compute the slice (Si) for the incorrectly valued output variable(s), which is a subset of KBI (known to be incorrect).
2. Compute the slice (Sc) for the correctly valued output variables(s), which is a subset of CSF (correct so far).
3. Compute (Si - Sc), which makes up the dice.

Example 3. Observe that a dynamic program slice in Example 1 is a subset of KBI, while a dynamic program slice in Example 2 is a subset of CSF. Consequently, using the definition of dicing, a dice program can be shown as follows
Once the procedure is finished, line 24 will be shown as the incorrect line.

6. Conclusions

In this paper, we presented the definitions, the algorithms, and the approaches used to compute a program slice and a program segment after dicing. Some examples were shown as well.

In this work, a number of definitions and algorithms originally introduced by Korel and Laski [5] were modified in order to compute slices in classes, objects, arrays, pointers, references, dynamic allocation operators, function overloading, copy constructors, default arguments, operator overloading, inheritance, virtual functions, polymorphism, templates, and exception handling of a C++ program. These definitions and algorithms were used to implement a tool named C++Debug.

C++Debug was designed to allow ease and convenience on the part of the user. Using C++Debug, a user can interact directly with the computer in locating errors in a program. For convenience, the program provides menus to allow the user to select any one of the functions contained therein. Based on the results of the experimentation, C++Debug could generate a new slicing program that is of smaller size than the original source program. The new slicing program still preserves part of the program's original behavior for a specific input. In addition, C++Debug can be used as a tool like ctrace under UNIX [14]. C++Debug can work on both C and C++.

7. References

[1] Keith Brian Gallagher, *Using Program Slicing in Software Maintenance*, Ph.D. Dissertation, Computer Science Department, University of Maryland, Baltimore County, MD, 1990.

[2] Keith Brian Gallagher and James R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, pp. 751-761, August 1991.

[3] Bogdan Korel, "PELAS-Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253-1260, September 1988.

[4] Bogdan Korel and Janusz Laski, "Dynamic Program Slicing," *Information Processing Letters*, Vol. 29, No. 3, pp. 155-163, October 1988.

[5] Bogdan Korel and Janusz Laski, "Dynamic Slicing of Computer Programs," *Journal of Systems and Software*, Vol. 13, No. 3, pp. 187-195, November 1990.

[6] James R. Lyle, *Evaluating Variations on Program Slicing for Debugging*, Ph.D. Dissertation, Computer Science Department, University of Maryland, College Park, MD, 1984.

[7] Sekaran Nanja, *An Interactive Debugging Tool for C Based on Program Slicing and Dicing*, Master of Science Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, May 1990.

[8] Sekaran Nanja and Mansur H. Samadzadeh, "A Slicing/Dicing-Based Debugger for C," *The 8th Annual Pacific Northeast Software Quality Conference*, Portland, OR, pp. 204-212, October 1990.

[9] Mansur H. Samadzadeh and Winai Wichaipanitch, "An Interactive Debugging tool for C based on Dynamic Slicing", *Proceedings of the 1993 ACM Computer Science Conference*, Indianapolis, IN, pp. 30-37, February 1993.

[10] Mark Weiser, "Program Slicing," *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, pp. 439-449, March 1981.

[11] Mark Weiser, "Programmers Use Slices When Debugging," *Communications of the ACM*, Vol. 25, No. 7, pp. 446-452, July 1982.

[12] Mark Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 352-357, July 1984.

[13] Mark Weiser and James R. Lyle, "Experiments on Slicing-Based Debugging Aids," *a paper presented at The First Workshop on Empirical Studies of Programmers*, (Soloway, E. and Iyengar, S., Editors), Ablex Publishing Corporation, Norwood, NJ, pp. 187-197, 1986.

[14] "UNIX IN A NUTSHELL," http:/www.oreilly.com/ catalog/unixcd/chapter/c02_043.htm, Last Update: November 1998, Last Access: May 19, 2003.