

AN INTERACTIVE DEBUGGING TOOL FOR C++  
BASED ON DYNAMIC SLICING AND DICING

By

WINAI WICHAIPANITCH

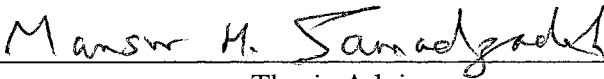
Bachelor of Science (Electrical Engineering)  
Rajamangala Institute of Technology  
Bangkok, Thailand  
1984

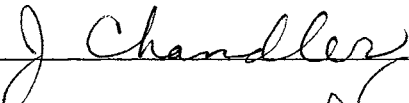

Master of Science (Computer Science)  
Oklahoma State University  
Stillwater, Oklahoma  
1992

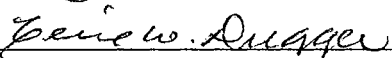
Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the Degree of  
DOCTOR OF PHILOSOPHY  
August 2003

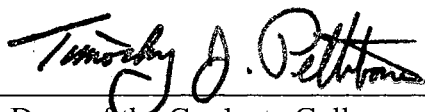
AN INTERACTIVE DEBUGGING TOOL FOR C++  
BASED ON DYNAMIC SLICING AND DICING

Thesis Approved:

  
\_\_\_\_\_  
Thesis Adviser

  
\_\_\_\_\_  
  
\_\_\_\_\_

  
\_\_\_\_\_

  
\_\_\_\_\_  
Dean of the Graduate College

## PREFACE

Since the article "Program Slicing" by Mark Weiser was initially published in 1981 [Weiser 81], 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 (designed and implemented by Lyle in 1984) was designed to be used with Fortran programs, and C-Sdicer (designed and implemented by Nanja and Samadzadeh in 1990) and C-Debug (designed and implemented by Wichaipanitch and Samadzadeh in 1992) were designed to be applicable to C language programs based on dynamic slicing.

Program slicing [Weiser 81, 82, and 84] 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 [Korel and Laski 90]. 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 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 [Lyle 84] [Nanja 90] [Nanja and Samadzadeh 90] 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.

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++.

## ACKNOWLEDGMENTS

I owe a great deal of gratitude and appreciation to my major adviser Dr. Mansur H. Samadzadeh for his guidance, motivation, dedication, and valuable instruction during my dissertation work. Dr. Samadzadeh continued to spend endless hours reviewing my work and offering suggestions for further refinement.

I wish to thank my other committee members Drs. Blayne E. Mayfield, John P. Chandler, and Cecil Dugger. Their time and efforts are greatly appreciated.

Many thanks are due to my wife Cholada for her moral support.

Finally, but certainly not least, I wish to thank my parents, Mr. Arun and Mrs. Tonghaw. It was their examples of hard work over many years that gave me the inspiration and motivation to complete my graduate studies. I will be forever indebted to them for this.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION .....	1
1.1 Introduction .....	1
1.2 Purpose of the Study .....	2
1.3 Organization of the Report .....	3
II. LITERATURE REVIEW .....	5
2.1 Introduction .....	5
2.2 Program Slicing .....	7
2.2.1 Static Slicing .....	8
2.2.2 Dynamic Slicing .....	8
2.3 Dicing .....	9
2.4 Examples .....	9
2.5 Dynamic Slicing Procedures .....	14
2.5.1 Background .....	14
2.5.2 Slicing Criterion .....	16
2.5.3 Steps Needed to Obtain a Dynamic Program Slice .....	17
2.6 Dicing Procedures .....	24
2.7 Problems with Slices .....	25
III. C++ DYNAMIC SLICING AND DICING PROCEDURES .....	26
3.1 Introduction .....	26
3.2 Definitions .....	26
3.3 Algorithms .....	48
3.4 Examples: How to Compute a Slice of a Program Containing Functions .....	52
3.5 A Slice with Classes, Structures, and Unions .....	59
3.6 Problems and Situations in C++ That Were Taken into Account in the Design .....	70
3.7 Dicing Procedure .....	72

Chapter	Page
IV. C++DEBUG .....	73
4.1 Introduction .....	73
4.2 Software specification .....	74
4.3 Software Design and Implementation .....	74
4.3.1 C++Debug Block Diagram .....	74
4.3.2 Datastructures .....	76
4.3.3 Symbol Tables .....	76
4.4 Testing and Evaluation .....	77
4.4.1 Introduction .....	77
4.4.2 Testing .....	77
4.4.3 Evaluation .....	77
4.5 Limitations .....	77
4.6 Program Documentation .....	78
4.7 System Evolution .....	78
4.8 Slicing-Based Metrics .....	79
V. SUMMARY, CONCLUSIONS, AND FUTURE WORK .....	82
5.1 Summary .....	82
5.2 Conclusions .....	83
5.3 Future Work .....	83
5.3.1 Improvements .....	83
5.3.2 Additions .....	84
5.3.3 Future Work .....	84
REFERENCES .....	86
APPENDICES .....	88
APPENDIX A – GLOSSARY .....	89
APPENDIX B – USER’S MANUAL FOR C++DEBUG .....	94
APPENDIX C – DATASTRUCTURES DESIGN FOR C++DEBUG BASED ON DYNAMIC PROGRAM SLICING AND DICING .....	104
APPENDIX D – SOFTWARE SPECIFICATIONS .....	120

Chapter	Page
APPENDIX E – TESTING AND EVALUATION .....	126
APPENDIX F – SAMPLE PROGRAMS USED FOR THE COMPUTATION OF SLICING-BASED METRICS .....	139
APPENDIX G – SAMPLE C++DEBUG SOURCE CODE LISTING .....	146



## LIST OF FIGURES

Figure	Page
1. The basic idea of program slicing and dicing .....	6
2. A program for counting occurrences and calculating the sum and average of a set of numbers .....	10
3. The output data of the program in Figure 2 .....	11
4. A static program slice computed based on variable <code>CountNumber</code> in line 19 of the program in Figure 2 .....	11
5. A dynamic program slice computed based on variable <code>CountNumber [ 1 ]</code> in line 19 of the program in Figure 2 .....	12
6. A dynamic program slice computed based on variable <code>CountNumber [ 2 ]</code> in line 19 of the program in Figure 2 .....	12
7. A program slice computed based on variable <code>Sum</code> in line 19 of the program in Figure 2 .....	13
8. A program slice computed based on variable <code>Avg</code> in line 19 of the program in Figure 2 .....	13
9. A final program segment after using dicing .....	13
10. A trajectory of the program from Figure 2 on input data <code>MaxData = 2, Data = (3,5)</code> .....	15
11. The sets $D(x^p)$ and $U(x^p)$ , definition and use, for the trajectory in Figure 10 .....	18
12. The DU (definition-use) relation for the trajectory depicted in Figure 10 .....	19

Figure	Page
13. The TC (test-control) relation for the trajectory depicted in Figure 10 .....	20
14. The IR (identity relation) relation for the trajectory depicted in Figure 10 .....	20
15. A dynamic program slice computed based on variable CountNumber[1] in line 19 of the program in Figure 2 .....	22
16. A dynamic program slice computed based on variable CountNumber[5] in line 19 of the program in Figure 2 .....	22
17. A dynamic program slice computed based on variable Sum in line 19 of the program in Figure 2 .....	23
18. A dynamic program slice computed based on variable Avg in line 19 of the program in Figure 2 .....	24
19. The final program segment after slicing and dicing .....	25
20. A program for computing the factorial of a number .....	28
21. A trajectory of the program in Figure 20 on input data Num = 3 .....	29
22. The sets $M(TF_{Fac})$ , $DF_{Fac}(X^p)$ , $UF_{Fac}(X^p)$ , and $LF_{Fac}(X^p)$ for the trajectory in Figure 21 .....	33
23. The sets $M(TF_{main})$ , $DF_{main}(X^p)$ , $UF_{main}(X^p)$ , and $LF_{main}(X^p)$ for the trajectory in Figure 21 .....	33
24. The $DUF_{Fac}$ relation for the trajectory depicted in Figure 21 .....	35
25. The $DUF_{main}$ relation for the trajectory depicted in Figure 21 .....	35
26. The $LDRF_{Fac}$ relation for the trajectory depicted in Figure 21 .....	35
27. The $LDRF_{main}$ relation for the trajectory depicted in Figure 21 .....	35
28. The $TCF_{Fac}$ relation for the trajectory depicted in Figure 21 .....	36

Figure	Page
29. The $IRF_{Fac}$ relation for the trajectory depicted in Figure 21 .....	37
30. A trajectory of functions A and B where function A calls function B.....	38
31. Illustrate Called-to-Calling .....	39
32. Illustrate Calling-to-Called .....	40
33. The EI relation for the trajectory depicted in Figure 21 .....	42
34. The IE relation for the trajectory depicted in Figure 21 .....	42
35. Rules for computing the CS (control scope) set .....	46
36. The Prototype, Called, Calling, D, U, DCL, VS, and CS sets for the program depicted in Figure 20 .....	47
37. Algorithm to compute a set of slices .....	49
38. Slicing data structures .....	50
39. Algorithm to compute a slice of each function .....	51
40. Function to compute the scope of influences of a slice .....	51
41. A dynamic program slice computed based on variable Num in line 23 of the program in Figure 20 .....	54
42. A dynamic program slice computed based on variable Fac in line 22 of the program in Figure 20 .....	56
43. A dynamic program slice computed based on variable I in line 8 of the program in Figure 20 .....	59
44. A program for calculating the sum and average of a set of numbers .....	60
45. The trajectory of the program from Figure 44 on input data Max = 4, Num = (10.0, 20.0, 15.0, 5.0) .....	61
46. The Prototype, Called, Calling, D, U, DCL, VS, and CS sets for the program depicted in Figure 20 .....	62

Figure	Page
47. The $DUF_{Main}$ , $TCF_{Main}$ $LDF_{Main}$ , and $IRF_{Main}$ relations that are called by 32 <sup>20</sup> for the trajectory depicted in Figure 45 .....	63
48. The $DUF_{Compute}$ , $TCF_{Compute}$ $LDF_{Compute}$ , and $IRF_{Compute}$ relations that are called by 32 <sup>20</sup> for the trajectory depicted in Figure 45 .....	63
49. The $DUF_{Sum}$ , $TCF_{Sum}$ , $LDF_{Sum}$ , and $IRF_{Sum}$ relations that are called by 32 <sup>20</sup> for the trajectory depicted in Figure 45 .....	63
50. The $DUF_{Sum}$ , $TCF_{Sum}$ $LDF_{Sum}$ , and $IRF_{Sum}$ relations that are called by 24 <sup>22</sup> for the trajectory depicted in Figure 45 .....	63
51. The $DUF_{Avg}$ , $TCF_{Avg}$ , and $IRF_{Avg}$ relations for the trajectory depicted in Figure 45 .....	64
52. A dynamic program slice computed based on variable Avg in line 33 of the program in Figure 44 .....	67
53. A dynamic program slice computed based on variable Sum in line 32 of the program in Figure 44 .....	69
54. The final program segment after slicing and dicing .....	72
55. Block diagram of C++Debug .....	75
56. Help menu and prompt .....	95
57. The trajectory path .....	99
58. Data structure of Types .....	105
59. A C++ program that uses iterators .....	105
60. Show the database of Types used in C++ .....	106
61. Show how the database stores Types of the program in Figure 59 .....	106
62. Data structure of Declarations .....	107
63. Show how the database stores Declarations of the program in Figure 59 .....	107

Figure	Page
64. Scopes of variable <code>x</code> as a global, local, and second local .....	108
65. Data structure of <code>Typedef</code> .....	109
66. A program segment that uses <b><code>typedef</code></b> .....	109
67. Show how the database stores <code>Typedef</code> defined by <b><code>typedef</code></b> in Figure 66 .....	109
68. Data structure of <code>Pointers</code> .....	110
69. A program segment that uses pointers .....	110
70. Show how the database stores <code>Pointers</code> of the part of the program in Figure 69 .....	111
71. Show how the database stores <code>Def(n)</code> and <code>Ref(n)</code> of the part of the program in Figure 69 .....	111
72. Data structure of <code>Arrays</code> .....	111
73. A program segment that uses arrays .....	112
74. Show the data base of <code>Arrays</code> used by the part of the program in Figure 73 .....	112
75. A program segment that uses pointers into arrays .....	112
76. Show how the database uses function <code>InsertPointerName(ID)</code> in Section C.3.1 to store variables of the part of the program in Figure 75 .....	112
77. Show how the database uses function <code>InsertArrayName(ID)</code> in Section C.3.2 to store variables of the part of the program in Figure 75 .....	113
78. Show how the database stores <code>Def(n)</code> and <code>Ref(n)</code> of the part of the program in Figure 75 .....	113
79. A program segment that uses <b><code>const</code></b> .....	113
80. Show how the database stores constant declared in Figure 79 .....	114

Figure	Page
81. Show how the database uses function <code>InsertPointerName(ID)</code> in Section C.3.1 to variables of the part of the program in Figure 79 .....	114
82. Show how the database stores <code>Def(n)</code> and <code>Ref(n)</code> of the part of the program in Figure 79 .....	114
83. Data structure of <code>References</code> .....	115
84. A program segment that uses references .....	115
85. Show how the database uses function <code>InsertReferenceName(ID)</code> to store variables of the part of the program in Figure 84 .....	115
86. A program segment that uses pointer to <b>void</b> .....	116
87. Show how the database uses function <code>InsertPointerName(ID)</code> in Section C.3.1 to store variables of the part of the program in Figure 86 .....	116
88. Data structure of <code>Structures</code> .....	117
89. A program segment that uses structures .....	117
90. Show how the database stores <code>Structures</code> of the part of the program in Figure 89 .....	118
91. Show how to determine the set of variables by using functions <code>Def(n)</code> and <code>Def(n)</code> .....	119
92. Part of function <code>UsedVariable</code> and its path .....	130
93. C++Debug block diagram .....	131

## LIST OF TABLES

Table	Page
I. Description of the five test programs .....	80
II. Slicing-based metrics obtained from C-Sdicer for the five test programs .....	81
III. Slicing-based metrics obtained from C++Debug for the five test programs .....	81
IV. Background summary .....	136
V. Language frequency .....	136
VI. Slices and program changes .....	137
VII. Time measures for debugging by using the tool .....	137
VIII. Time measures debugging without using the tool .....	138

## CHAPTER I

### INTRODUCTION

#### 1.1 Introduction

Once a programmer finds that a program fails to function properly in the testing process, debugging techniques are used to localize the causes of the errors and to correct them. All too often, one finds that the cost associated with testing and correcting a program is likely to increase as the size of the program increases and as the program becomes more complicated [Tassel 74]. As a result, various tools and methods have been developed to debug programs; for example, file printing utilities, module testing packages, built-in language facilities and programmed-in aids, post-mortem dumps, and source code amendment facilities [Tassel 74].

Program slicing [Weiser 81, 82, and 84] 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 variable of interest (criterion variable), with the statements that are unrelated being omitted. Using the slicing method, one obtains a new program of generally smaller size, which still maintains all aspects of the original program's behavior with respect to the criterion variable. A dicing technique [Lyle 84] [Nanja 90] [Nanja and Samadzadeh 90] can then be used to compare two or more slices,



resulting from the program slicing technique, to identify the set of statements that are likely to contain an error.

Program slicing can be classified into two main categories according to how slices are computed: static slicing and dynamic slicing. Static slicing is a method of computing program slices directly from the original source programs. Dynamic slicing is a method used to compute program slices from the trajectory, which is a feasible path that has actually been executed for some input. Dynamic slicing differs from static slicing in that it is defined on the basis of one computation rather than for all possible computations [Korel and Laski 90]. As the results, the slice size computed based on the dynamic slicing technique is typically smaller. Furthermore, it allows us to treat the elements and fields in dynamic records as individual variables.

C++ is a general-purpose programming language and is successfully used in many application areas [Stroustrup 97]. Implementations of C++ exist from some of the most modest microcomputers to the largest supercomputers, and for almost all operating systems. C++ adds to C the concept of *class*, a mechanism for providing user-defined types that is also called *abstract data type* [Pohl 94]. C++ supports *object-oriented* programming by providing inheritance and run-time type binding in addition to the concept of class. As a result, a lot of programmers use C++ to implement programs and hence tools are needed to localize the causes of errors detected during testing.

## 1.2 Purpose of the Study

The objective was to create an interactive debugging tool, called C++Debug, for debugging a C++ program running under UNIX on the SUN machine in the Computer

Science Department at OSU. C++Debug was designed to function as a utility program of the UNIX system and was developed based on slicing and dicing techniques. It was designed in a way to provide ease of use and convenience on the part of the user. Using C++Debug, a user can interact with the computer in locating errors in a program. In order for C++Debug to give smaller slice sizes, dynamic slicing rather than static slicing was chosen.

The scope of C++Debug includes programs that contain ANSI C and C++ codes. Classes and objects, unions, records, arrays, pointers, references, dynamic allocations, function and operator overloading, copy constructors and defaults, inheritances, virtual functions and polymorphism, templates, and exception handling were included also.

### 1.3 Organization of the Report

The rest of this dissertation report is organized as follows. Chapter II reviews the literature related to general information on program slicing and dicing techniques. The chapter concludes with a discussion of the advantages and disadvantages of dynamic and static slicing, and the procedures used to locate errors in a program using dynamic slicing and dicing techniques. Chapter III presents definitions and algorithms to get slices and dices in a C++ program. Chapter IV presents the steps involved in the design and implementation of C++Debug, its testing and evaluation, and the advantages and limitations of C++Debug. Chapter V contains a summary, conclusions, and some areas of future work.

There are seven appendices: one on notation, one containing a user's manual for C++Debug, one containing datastructure design for C++Debug, one containing software

specifications, one containing testing and evaluation, one containing sample programs used for the computation of slicing-based metrics, and the final appendix contains sample source code listing of C++Debug.

## CHAPTER II

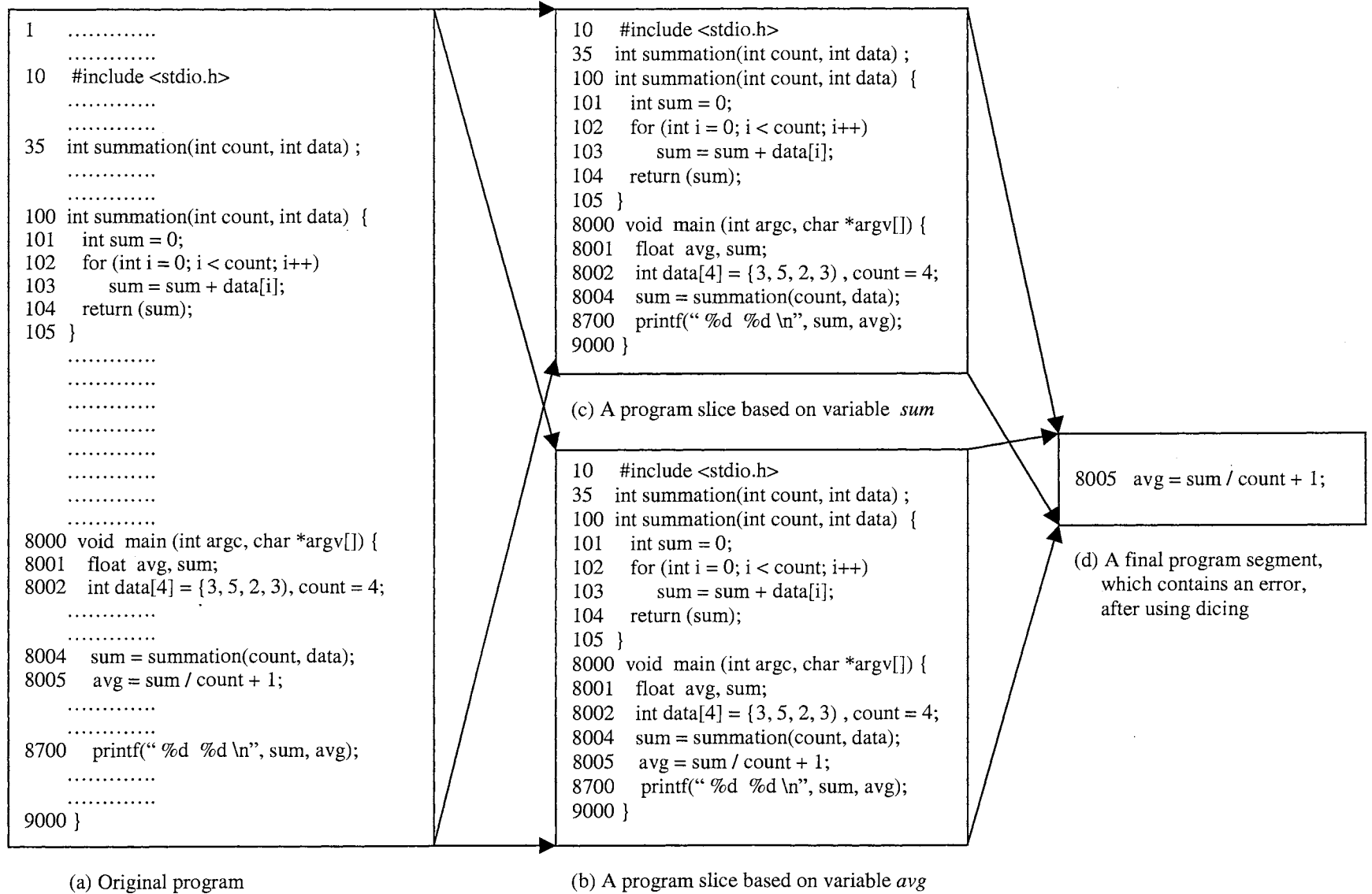
### LITERATURE REVIEW

#### 2.1 Introduction

Localizing program errors is an arduous and time-consuming task, especially when programs written by other people are involved. Several attempts have been made to find ways that can enable one to locate errors more rapidly and effectively. Program slicing [Weiser 81, 82, and 84] [Korel 88] [Gallagher and Lyle 91] is one of several methods that have been used for this purpose.

Figure 1 shows the basic idea of program slicing and dicing. Let us assume that Figure 1(a) is a program to compute a tax fee. It is a large program with, say, 9000 lines of code. In fact, the actual number of statements or functions is not very important. For this program, let us assume we have found that the variable `avg` in line 8700 gives an incorrect result, e.g., 4.25 instead of 3.25. Because the program is too large, it is difficult to localize where the error is. Using program slicing technique based on variable `avg`, we can get a new program of smaller size, 15 lines in this case, which still maintains all aspects of the original program's behavior (Figure 1(b)). Now, although the result is a new program of a smaller size, sometime we cannot find where the error is. Fortunately,

Figure 1. The basic idea of program slicing and dicing



in this example, variable `sum` in line 8700 gives the correct answer and yields the slice as shown in Figure 1(c). To localize an error, dicing technique can be used by comparing both slices, where one contains no errors but the other one does. Some statements sliced on correct variables will then be removed from those sliced on variables with incorrect values. Upon completion of these steps, we get a new slice that is smaller and contains an error as shown in Figure 1(d). Finally, it is discovered that the correct statement should be  $avg = sum/count$ , instead of  $avg = sum/count+1$ .

## 2.2 Program Slicing

The notion of slicing is based on the premise that instead of locating errors in the original program, which can be of large size, one can locate errors in a program of smaller size, which is sliced from the original program but still preserves part of the original program's behavior for a particular variable [Weiser 84].

Advantages of slices and slicing are based on four points [Weiser 84]. First, slices can be found automatically by a method used to decompose programs through analyzing their data flow and control flow. Second, a slice is normally smaller than the original program. As a consequence, when slicing at a variable of interest, the size of the resulting program slice is generally smaller than that of the original program. Third, slices can be executed independently of one another. In other words, a slice is itself an executable program whose behavior is identical to the specified subset of the original program's behavior. In other words, a slice produces a specific projection of the original program's behavior.

In addition to the four points mentioned above, Weiser also mentioned two

intuitively desirable properties of a slice [Weiser 81]. First, a slice must be obtained from the original program by statement deletion. Second, once the statement deletion has been accomplished, the behavior of the resulting slice must correspond to the behavior of the original program as observed through a particular variable in the slicing criterion.

Program slicing can be classified into two main categories: static slicing and dynamic slicing. These categories are discussed below.

### 2.2.1 Static Slicing

Static slicing [Weiser 81, 82, and 84] is a method defined on the basis of all computations of a program. It yields a program slice of generally larger size than that of dynamic slicing (or, in the best case, of equal size to that of dynamic slicing), because static slicing often gives a slice containing statements that have no influence on the values of variables of interest for a particular execution [Korel and Laski 90]. Also, static slicing cannot treat the array elements and fields in dynamic records as individual variables. Finally, static slicing cannot support run-time handling. A static program slice is determined directly from the original source program (see Section 2.4 for examples).

### 2.2.2 Dynamic Slicing

Unlike static slicing, dynamic slicing [Korel 88] [Korel and Laski 88 and 90] is defined on the basis of one computation rather than all computations, and generates a dynamic program slice by computing from the trajectory that is a feasible path that has actually been executed for some input of the original source program (see Section 2.4 for examples). In addition, this method enables one to treat the array elements and fields in dynamic records as individual variables [Korel and Laski 90]. In this way, the size of the

resulting slice becomes generally smaller. Moreover, dynamic slicing allows one to keep track of run-time type binding, which is unknown at compile time but is determined until when the program is executed.

### 2.3 Dicing

Dicing is the process of identifying a set of statements likely to contain an error [Lyle 84] [Nanja 90] [Nanja and Samadzadeh 90] [Samadzadeh and Wichaipanitch 93]. The idea is first to compare two or more slices using program slicing techniques (see Section 2.4 for an example). Only one of these will slice on a variable with an incorrect value and the other(s) will slice on variables with correct values. One must first ascertain that the latter contains no errors. Some statements sliced on correct variables will then be removed from those sliced on the variable with incorrect values. Upon completion of these steps, a new slice is obtained that is smaller and contains the error.

It should be noted that the validity of the use of dicing rests on three important assumptions [Weiser and Lyle 86]. First, it is necessary that testing be reliable and that all incorrectly computed variables be identified. Second, if the computation of a variable  $V$  depends on the computation of another variable  $W$ , then whenever  $W$  has an incorrect value, so does  $V$ . Third, it is necessary that one and only one fault exist in the program.

The next sections provide examples of the computation of slices (static and dynamic) and dices.

### 2.4 Examples

Illustrated below are the comparison of the sizes of program slices generated from



static slicing, dynamic slicing, and dicing techniques.

The program in Figure 2 is designed to count the number of individual integers falling between 1 and 5 read from the input data. Furthermore, this program computes the sum and average of these integers. In this example, the variable `MaxData` is 5 and the array called `Data` contains 3, 5, 5, 2, and 2. Upon completion of program execution, the program should yield the results as shown in Figure 3. However, this program

```

Var
  MaxData, Count      : integer;
  Sum, Avg            : real;
  Data, CountNumber  : array[1..10] of integer;
                      /* Data = (3,5,5,2,2) */
                      /* MaxData = 5 */

begin
  1  read(MaxData, Data);
  2  Count := 1;
  3  Sum   := 0;
  4  while Count <= MaxData do
      begin
        /* count occurrences of number */
  5    if Data[Count] = 1 then
  6      CountNumber[1] := CountNumber[1] + 1;
  7    if Data[Count] = 2 then
  8      CountNumber[2] := CountNumber[2] + 1;
  9    if Data[Count] = 3 then
 10     CountNumber[3] := CountNumber[3] + 1;
 11    if Data[Count] = 4 then
 12     CountNumber[4] := CountNumber[4] + 1;
 13    if Data[Count] = 5 then
 14     CountNumber[5] := CountNumber[5] + 1;
        /* computing summation */
 16    Sum := Sum + Data[Count];
 17    Count := Count + 1;
      end;
        /* computing average */
 18    Avg := Sum / (MaxData + 1);
        /* display output */
 19    write(CountNumber, Sum, Avg);
end

```

Figure 2. A program for counting occurrences and calculating the sum and average of a set of numbers

contains an error in line 18. Rather than  $Avg := Sum/MaxData$ , the program computes  $Avg := Sum/MaxData+1$ , thus yielding an error ( $Avg = 2.8$  instead of 3.4). To localize such an error, program slicing and dicing techniques can be used.

Figure 4 is a static program slice computed based on variable `CountNumber` in line 19. The static slicing method treats array variables as a single variable regardless of the number of elements in the array. In contrast, if the dynamic slicing approach is used, array elements are treated as individual variables. As a result, the size of a program slice is generally reduced by using dynamic slicing techniques. As shown in Figure 5, no

```
Number of each integer: 0,2,1,0, and 2, respectively
Sum = 17
Avg = 3.4
```

Figure 3. The output data of the program in Figure 2

```
Begin
  1 read(MaxData, Data);
  2 Count := 1;
  4 while Count <= MaxData do
    begin
  5   if Data[Count] = 1 then
  6     CountNumber[1] := CountNumber[1] + 1;
  7   if Data[Count] = 2 then
  8     CountNumber[2] := CountNumber[2] + 1;
  9   if Data[Count] = 3 then
 10    CountNumber[3] := CountNumber[3] + 1;
 11   if Data[Count] = 4 then
 12    CountNumber[4] := CountNumber[4] + 1;
 13   if Data[Count] = 5 then
 14    CountNumber[5] := CountNumber[5] + 1;
 17   Count := Count + 1;
    end;
 19 write(CountNumber, Sum, Avg);
end
```

Figure 4. A static program slice computed based on variable `CountNumber` in line 19 of the program in Figure 2

program slice results from variable `CountNumber [1]` in line 19.

Figure 6 shows the program slice resulting from computing a dynamic slice based on variable `CountNumber [2]` in line 19. Obviously, the sizes of the program slices resulting from `CountNumber [1]` and `CountNumber [2]` are different because dynamic slicing treats the two array elements as two different variables whereas static slicing does not.

Figures 7 and 8 depict program slices generated based on variables `Sum` and `Avg`, respectively, in line 19. For these variables, the static slicing method [Lyle 84] [Nanja 90] [Nanja and Samadzadeh 90] and the dynamic slicing method [Korel and Laski 90] yield identical results.

```

Begin
  19 write(CountNumber, Sum, Avg);
end

```

Figure 5. A dynamic program slice computed based on variable `CountNumber [1]` in line 19 of the program in Figure 2

```

Begin
  1 read(MaxData, Data);
  2 Count := 1;
  4 while Count <= MaxData do
    begin
  7   if Data[Count] = 2 then
  8     CountNumber[2] := CountNumber[2] + 1;
  17  Count := Count + 1;
    end;
  19 write(CountNumber, Sum, Avg);
end

```

Figure 6. A dynamic program slice computed based on variable `CountNumber [2]` in line 19 of the program in Figure 2

With the use of the dicing algorithm, a new program segment can be generated, as shown in Figure 9. This program segment contains the final erroneous line (which is line 18).

```

Begin
  1 read(MaxData, Data);
  2 Count := 1;
  3 Sum   := 0;
  4 while Count <= MaxData do
    begin
  16   Sum := Sum + Data[Count];
  17   Count := Count + 1;
    end;
  19 write(CountNumber, Sum, Avg);
end

```

Figure 7. A program slice computed based on variable Sum in line 19 of the program in Figure 2

```

Begin
  1 read(MaxData, Data);
  2 Count := 1;
  3 Sum   := 0;
  4 while Count <= MaxData do
    begin
  16   Sum := Sum + Data[Count];
  17   Count := Count + 1;
    end;
  18 Avg := Sum / (MaxData + 1);
  19 write(CountNumber, Sum, Avg);
end

```

Figure 8. A program slice computed based on variable Avg in line 19 of the program in Figure 2

```

18 Avg := Sum / (MaxData + 1);

```

Figure 9. A final program segment after using dicing

## 2.5 Dynamic Slicing Procedures

### 2.5.1 Background

To facilitate better understanding of program slicing, it is necessary that the following background [Korel and Laski 90] be presented.

Let the flowgraph of a program  $P$  be a directed graph  $(N, A, s, e)$  and  $C$  be a *slicing criterion*, where  $N$  is the set of nodes,  $A$  is a binary relation on  $N$  (a subset of  $N \times N$ ), 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: a single instruction or a control instruction. A single instruction can be an assignment statement, an input or output statement, etc. A control instruction can be such statements as an **if-then-else** statement or a **while** statement that are also called *test instructions*.

An  $\text{arc}(n, m) \in A$  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  $\langle n_1, n_2, \dots, n_q \rangle$  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, which cause a path to be traversed during program execution, the path is *feasible*.

A *trajectory* is a feasible path that has actually been executed for some input. For example, in Figure 10,  $\langle 1, 2, 3, 4, 9, 10, 16, 17, 4, 13, 14, 16, 17, 4, 18, 19 \rangle$  is the trajectory when the program in Figure 2 is executed on input data  $\text{MaxData} = 2$ ,  $\text{Data} = (3, 5)$ . A resulting trajectory can be the initial segment of an infinite path if the execution does not terminate in the case of an infinite loop. A trajectory is illustrated in terms of an

```

var
    MaxData, Count      : integer;
    Sum, Avg            : real;
    Data, CountNumber  : array[1..10] of integer;
                        /* Data = (3,5), and MaxData = 2 */
begin
/* action  instruction in action */

    11    read(MaxData, Data);
    22    Count := 1;
    33    Sum := 0;
    44    Count <= MaxData
    95    Data[Count] = 3
106    CountNumber[3] := CountNumber[3] + 1;
167    Sum := Sum + Data[Count];
178    Count := Count + 1;
    49    Count <= MaxData
1310   Data[Count] = 5
1411   CountNumber[5] := CountNumber[5] + 1;
1612   Sum := Sum + Data[Count];
1713   Count := Count + 1;
    414   Count <= MaxData
1815   Avg := Sum / (MaxData + 1);
1916   write(CountNumber, Sum, Avg);

Trajectory T = <1,2,3,4,9,10,16,17,4,13,14,16,17,4,18,19>

```

Figure 10. A trajectory of the program from Figure 2 on input data  
MaxData = 2, Data = (3,5)

abstract list whose elements are accessed according to their positions in it. For example,  $T(2) = 2$  and  $T(5) = 9$ . A trajectory is also illustrated in terms of a pair (instruction, its position in the trajectory), rather than the instruction itself, so as to distinguish between multiple occurrences of the same instruction in the trajectory.

For example, instruction  $X$  at position  $p$  in trajectory  $T$  is represented by  $\text{pair}(X, p)$ . For brevity and ease of understanding,  $\text{pair}(X, p)$  is replaced by  $X^p$  and is referred to as an *action*. For example,  $4^4$  and  $4^9$  in trajectory  $T$  in Figure 10 are actions that involve the same instruction 4. An action  $X^p$  is a *test action* if  $X$  is a test instruction. For

example,  $4^4$ ,  $4^9$ , and  $4^{14}$  in trajectory  $T$  in Figure 10 are test actions.

Let  $T = \langle X_1, X_2, \dots, X_m \rangle$  denote a trajectory of length  $m$ , and  $q$  be a position in  $T$ ,  $1 \leq q \leq m$ . Then the following can be obtained.

1.  $\text{Front}(T, q)$  denotes the sublist  $\langle X_1, X_2, \dots, X_q \rangle$ , consisting of the first  $q$  elements of  $T$ .
2.  $\text{Back}(T, q)$  denotes the sublist  $\langle X_{q+1}, X_{q+2}, \dots, X_m \rangle$ , consisting of elements that follows  $T(q)$ , a trajectory at position  $q$ .

So, for all  $T$  and  $q$  the following can be obtained.

$T = \text{Front}(T, q) \parallel \text{Back}(T, q)$ , where  $\parallel$  represents concatenation.

3.  $\text{DEL}(T, R)$ , where  $R$  is a predicate on the set of instructions in  $T$ , means a subtrajectory obtained from  $T$  by deleting from it all elements  $T(i)$  that satisfy  $R$ .

### 2.5.2 Slicing Criterion

A slicing criterion 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 [Weiser 81]. 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 the variables in  $P$  [Korel and Laski 90].

It is readily apparent that the slicing criterion of dynamic slicing differs from that of static slicing. The slicing criterion of dynamic slicing contains an input value  $x$ , whereas that of static slicing contains only a pair  $C = (I, V)$ . This is because a change in the value of input  $x$  will result in a change in the trajectory, which in turn may result in a

change in the size of the resulting slice. That is to say, the slicing criterion of dynamic slicing is defined in terms of a given trajectory on a specific input  $x$ , rather than in terms of the set of all possible paths. In the case of static slicing, a slicing criterion is an instruction  $I$  in a program  $P$ , while in the case of dynamic slicing, a slicing criterion is an instruction  $I$  at a particular execution position  $q$  in a trajectory  $T$ .

### 2.5.3 Steps Needed to Obtain a Dynamic Program Slice

The procedure needed to obtain a dynamic program slice can be summarized in five steps as explained below along with examples to illustrate the process.

1. Find a trajectory [Korel 88] [Korel and Laski 88 and 90] of the program (a trajectory is a feasible path traversed during program execution, see Subsection 2.5.1 for details). For the program in Figure 2, a trajectory is shown in Figure 10. In Figure 10, all instructions in the trajectory represent a pair consisting of an instruction and its position in the trajectory, instead of the instruction itself. In other words,  $X$  at position  $p$  in  $T$  will be referred to as  $pair(X, p)$  or  $X^p$ , which is referred to as an action [Korel and Laski 90]. For instance,  $4^4$  and  $4^9$  in trajectory  $T$  in Figure 10 are actions involving the same instruction 4. An action  $X^p$  is a test action provided that  $X$  is a test instruction [Korel and Laski 90].

2. For each line  $X^p$  in the trajectory, compute  $U(X^p)$ , the set of variables that are used in  $X^p$ , and also compute  $D(X^p)$ , the set of variables that are defined in  $X^p$  [Korel and Laski 90]. For example, in the execution trace of Figure 10 we have

$$18^{15} \quad Avg := Sum / (MaxData + 1);$$

$Avg$  is a set of variables defined in  $18^5$ ,  $D(18^5)$ .  $Sum$  and  $MaxData$  are a set of



variables that are used in  $18^5$ ,  $U(18^5)$ . The sets  $U(X^p)$  and  $D(X^p)$  for the trajectory in Figure 10 are shown in Figure 11.

Action	$D(X^p)$	$U(X^p)$
$1^1$	MaxData, Data	
$2^2$	Count	
$3^3$	Sum	
$4^4$		Count, MaxData
$9^5$		Data[1], Count
$10^6$	CountNumber[3]	CountNumber[3]
$16^7$	Sum	Sum, Data[1], Count
$17^8$	Count	Count
$4^9$		Count, MaxData
$13^{10}$		Data[2], Count
$14^{11}$	CountNumber[5]	CountNumber[5]
$16^{12}$	Sum	Sum, Data[2], Count
$17^{13}$	Count	Count
$4^{14}$		Count, MaxData
$18^{15}$	Avg	Sum, MaxData
$19^{16}$		CountNumber, Sum, Avg

Figure 11. The sets  $D(X^p)$  and  $U(X^p)$ , definition and use, for the trajectory in Figure 10

3. Compute the DU (Definition-Use) Relation, a relation in which one action assigns a value to an item of data and the other action uses that value [Korel and Laski 90]. For example, in the execution trace of Figure 11,  $2^2$  defines the variable Count, and  $4^4$ ,  $9^5$ ,  $16^7$ , and  $17^8$  use the defined value of that variable. Let  $M(T)$  be a set of actions in a given trajectory T, where  $M(T) = \{ (X, p) : T(p) = X \}$ . DU is a binary relation on  $M(T)$  defined below [Korel 88].

- $X^p \text{ DU } Y^t$ ,  $1 \leq p < t$ , iff there exists a variable  $v$  such that
- (1)  $v \in U(Y^t)$ , 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  $T$ .

For example, in the trajectory of Figure 11,  $2^2$  is the last definition of variable `Count` at the execution positions 3 through 8. The DU Relation for the trajectory in Figure 11 is shown in Figure 12.

$\text{DU}(1^1)$	=	$\{4^4, 9^5, 16^7, 4^9, 13^{10}, 16^{12}, 4^{14}, 18^{15}\}$
$\text{DU}(2^2)$	=	$\{4^4, 9^5, 16^7, 17^8\}$
$\text{DU}(3^3)$	=	$\{16^7\}$
$\text{DU}(10^6)$	=	$\{19^{16}\}$
$\text{DU}(16^7)$	=	$\{16^{12}\}$
$\text{DU}(17^8)$	=	$\{4^9, 13^{10}, 16^{12}, 17^{13}\}$
$\text{DU}(14^{11})$	=	$\{19^{16}\}$
$\text{DU}(16^{12})$	=	$\{18^{15}, 19^{16}\}$
$\text{DU}(17^{13})$	=	$\{4^{14}\}$
$\text{DU}(18^{15})$	=	$\{19^{16}\}$

Figure 12. The DU (definition-use) relation for the trajectory depicted in Figure 10

4. Compute the TC (Test-Control) Relation, capturing the effect between test actions and actions that have been chosen to execute by those test actions [Korel and Laski 90]. For example, in the execution trace of Figure 10, the scope of test action  $4^4$  influences the execution of  $9^5$ ,  $10^6$ ,  $16^7$ , and  $17^8$ , but it does not influence the execution of  $13^{10}$ ,  $14^{11}$ ,  $16^{12}$ , and  $17^{13}$ . Let  $M(T)$  be a set of actions in a given trajectory  $T$ . TC is a binary relation on  $M(T)$  defined below [Korel and Laski 90].

- $X^p \text{ TC } Y^t, 1 \leq p < t$ , 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.

For example, in the program of Figure 2, instructions 5, 7, 9, 11, 13, 16, and 17 are in the scope of influence of test instruction 4, but instructions 18 and 19 are not. The TC Relation for the trajectory in Figure 10 is shown in Figure 13.

5. Compute the slicing set  $S_c$  using the following definitions [Korel and Laski 90].

5.1 Let  $X^p \text{ IR } Y^t$ , iff  $X = Y$  is the identity Relation IR on  $M(\text{Front}(T, q))$ . The IR Relation for the trajectory in Figure 10 is obtained as shown in Figure 14.

$\text{TC}(4^4)$	$= \{9^5, 10^6, 16^7, 17^8\}$
$\text{TC}(4^9)$	$= \{13^{10}, 14^{11}, 16^{12}, 17^{13}\}$
$\text{TC}(4^{14})$	$= \{\}$
$\text{TC}(9^5)$	$= \{10^6\}$
$\text{TC}(13^{10})$	$= \{14^{11}\}$

Figure 13. The TC (test-control) relation for the trajectory depicted in Figure 10

$\text{IR}(4^4)$	$= \{4^9, 4^{14}\}$
$\text{IR}(4^9)$	$= \{4^{14}\}$
$\text{IR}(4^{14})$	$= \{\}$

Figure 14. The IR (identity relation) relation for the trajectory depicted in Figure 10

5.2 Let  $C = (x, I^q, V)$  be a slicing criterion and  $T$  be a trajectory on input  $x$ . 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

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

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

$S_c$  can be determined iteratively as the limit of a sequence  $S^0, S^1, \dots, S^n, 0 \leq n < q$ , which is defined as follows

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

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

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

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

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

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

$$S_c = S^k \cup \{ I^q \}$$

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

Example 1. Consider again trajectory  $T$  in Figure 10. Using the criterion

$$C1 = (x, 19^{16}, \{ \text{CountNumber}[1] \}), x = (\text{MaxData}, \text{CountNumber}) = (2, (3, 5)),$$

we have

$$LD(16, \{ \text{CountNumber}[1] \}) = \{ \}, LT(19^{16}) = \{ \},$$

$$A^0 = \{ \}, S^0 = \{ \},$$

$$S_{c1} = S^0 \cup \{ 19^{16} \} = \{ 19^{16} \}.$$

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

```

begin
  19 write(CountNumber, Sum, Avg);
end.

```

Figure 15. A dynamic program slice computed based on variable CountNumber[1] in line 19 of the program in Figure 2

Example 2. Consider again trajectory  $T$  in Figure 10. Using the criterion

$$C2 = (x, 19^{16}, \{\text{CountNumber}[5]\}), \quad x = (\text{MaxData}, \text{CountNumber}) = (2, (3, 5)),$$

we have

$$LD(16, \{\text{CountNumber}[5]\}) = \{14^{11}\}, \quad LT(19^{16}) = \{\},$$

$$A^0 = \{14^{11}\}, \quad S^0 = \{14^{11}\},$$

$$A^1 = \{4^9, 13^{10}\}, \quad S^1 = \{4^9, 13^{10}, 14^{11}\},$$

$$A^2 = \{1^1, 4^4, 17^8\}, \quad S^2 = \{1^1, 4^4, 17^8, 4^9, 13^{10}, 14^{11}\},$$

$$A^3 = \{2^2\}, \quad S^3 = \{1^1, 2^2, 4^4, 17^8, 4^9, 13^{10}, 14^{11}\},$$

$$A^4 = \{\},$$

$$S_{c2} = S^3 \cup \{19^{16}\} = \{1^1, 2^2, 4^4, 17^8, 4^9, 13^{10}, 14^{11}, 19^{16}\}.$$

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

```

begin
  1 read(MaxData, Data);
  2 Count := 1;
  4 while Count <= MaxData do
    begin
  13   if Data[Count] = 5 then
  14     CountNumber[5] := CountNumber[5] + 1;
  17   Count := Count + 1;
    end;
  19 write(CountNumber, Sum, Avg);
end.

```

Figure 16. A dynamic program slice computed based on variable CountNumber[5] in line 19 of the program in Figure 2

Example 3. Consider again trajectory T in Figure 10. Using the criterion

$C3 = (x, 19^{16}, \{\text{Sum}\})$ ,  $x = (\text{MaxData}, \text{CountNumber}) = (2, (3, 5))$ , we have

$$LD(16, \{\text{Sum}\}) = \{16^{12}\}, LT(19^{16}) = \{\},$$

$$A^0 = \{16^{12}\}, \quad S^0 = \{16^{12}\},$$

$$A^1 = \{1^1, 16^7, 17^8, 4^9\}, \quad S^1 = \{1^1, 16^7, 17^8, 4^9, 16^{12}\},$$

$$A^2 = \{2^2, 3^3, 4^4\}, \quad S^2 = \{1^1, 2^2, 3^3, 4^4, 16^7, 17^8, 4^9, 16^{12}\},$$

$$A^3 = \{\},$$

$$S_{c3} = S^2 \cup \{19^{16}\} = \{1^1, 2^2, 3^3, 4^4, 16^7, 17^8, 4^9, 16^{12}, 19^{16}\}.$$

And finally, the dynamic slice is shown in Figure 17.

```

begin
  1 read(MaxData, Data);
  2 Count := 1;
  3 Sum   := 0;
  4 while Count <= MaxData do
    begin
  16   Sum := Sum + Data[Count];
  17   Count := Count + 1;
    end;
  19 write(CountNumber, Sum, Avg);
end.

```

Figure 17. A dynamic program slice computed based on variable Sum in line 19 of the program in Figure 2

Example 4. Consider again trajectory T in Figure 10. Using the criterion

$C4 = (x, 19^{16}, \{\text{Avg}\})$ ,  $x = (\text{MaxData}, \text{CountNumber}) = (2, (3, 5))$ , we have

$$LD(16, \{\text{Avg}\}) = \{18^{15}\}, LT(19^{16}) = \{\},$$

$$A^0 = \{18^{15}\}, \quad S^0 = \{18^{15}\},$$

$$A^1 = \{1^1, 16^{12}\}, \quad S^1 = \{1^1, 16^{12}, 18^{15}\},$$

$$A^2 = \{16^7, 17^8, 4^9\}, \quad S^2 = \{1^1, 16^7, 17^8, 4^9, 16^{12}, 18^{15}\},$$

$$A^3 = \{2^2, 3^3, 4^4\}, \quad S^3 = \{1^1, 2^2, 3^3, 4^4, 16^7, 17^8, 4^9, 16^{12}, 18^{15}\},$$

$$A^4 = \{\},$$

$$S_{c4} = S^3 \cup \{19^{16}\} = \{1^1, 2^2, 3^3, 4^4, 16^7, 17^8, 4^9, 16^{12}, 18^{15}, 19^{16}\}.$$

And finally, the dynamic slice is shown in Figure 18.

```

begin
  1 read(MaxData, Data);
  2 Count := 1;
  3 Sum   := 0;
  4 while Count <= MaxData do
    begin
  16   Sum := Sum + Data[Count];
  17   Count := Count + 1;
    end;
  18 Avg := Sum / (MaxData + 1);
  19 write(CountNumber, Sum, Avg);
end.

```

Figure 18. A dynamic program slice computed based on variable Avg in line 19 of the program in Figure 2

## 2.6 Dicing Procedures

Dicing [Lyle 84] [Nanja 90] [Nanja and Samadzadeh 90] is the process of identifying a set of statements likely to contain an error. A dice is determined using the following process.

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

Example 5. Observe that the dynamic program slice in Example 3 is a subset of CSF, while the dynamic program slice in Example 4 is a subset of KBI. Consequently, using the definition of dicing, a dice program can be shown in Figure 19.

```
18 Avg := Sum / (MaxData + 1);
```

Figure 19. The final program segment after slicing and dicing

Once the procedure is finished, line 18 will be shown as the incorrect line.

## 2.7 Problems with Slices

Although a number of significant advantages exist with the use of program slicing, program slicing does have disadvantages [Weiser 84]. These disadvantages can be summarized as follows. First, slices can be expensive to find. Second, a program may contain no significant slices other than itself. Third, total independence of slices may result in additional complexity in each slice that could be cleaned up if simple dependencies could be identified among slices. Finally, the selection of variables for slicing and dicing could pose significant problems. However, it can be asserted that whenever the program to be debugged is large, program slicing could effectively be used.



## CHAPTER III

### C++ DYNAMIC SLICING AND DICING PROCEDURES

#### 3.1 Introduction

A number of definitions and algorithms originally introduced by Korel and Laski [Korel and Laski 90] 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.

#### 3.2 Definitions

Based on Korel and Laski's work [Korel and Laski 90], 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  $N \times N$ ) 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  $\text{arc}(n, m) \in A$  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  $\langle n_1, n_2, \dots, n_q \rangle$  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*.

The program in Figure 20 is designed to compute the factorial of a given number  $\text{Num}$ . For example, if  $\text{Num} = 3$  the program yields the result of 6. Figure 21 shows a trajectory of the program in Figure 20 on input data  $\text{Num} = 3$ .

#### Definition 1

Let  $X$  be an instruction in a program and  $X \in \mathbb{IN}^+$  (the set of non-negative integers). Let  $P$  be the set of instruction numbers in a tested C++ program, then  $P = \{1, 2, \dots, n\}$  represents a program of length  $n$ , where  $n$  is the size of the program. For example, the C++ program in Figure 20 is the program  $P = \{1, 2, \dots, 25\}$ , where `#include <iostream>` is instruction  $X = 1$ , `int Fac(int N)` is instruction  $X = 3$ , etc.

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

```

1 #include <iostream>
2
3 int Fac(int N);      // function prototype
4
5 int Fac(int N) {    // called function
6
7     int F = 1;
8     int I = 2;
9
10    while(I <= N) {
11        F = F * I;
12        I++;
13    }
14    return F;
15 }
16
17 main() {           // main program
18
19     int Num;       // number
20
21     cin>>Num;
22     cout<<Fac(Num); // calling function
23     cout<<Num;
24
25 }

```

Figure 20. A program for computing the factorial of a number

where  $n$  = length of the program.

#### Definition 2

Let  $F_{\text{name}}$  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. For example, in Figure 20,  $F_{\text{Fac}} = \{5, 7, 8, 10, 11, 12, 13, 14, 15\}$  and  $F_{\text{main}} = \{17, 19, 21, 22, 23, 25\}$ .  $F_{\text{name}} \subseteq P$ , and  $F_{\text{name}} = F_{\text{main}}$  if the program has one function.

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

- 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 [Korel and Laski 90]. A trajectory of length  $m$  is denoted by a list  $T = \langle X_1, X_2, \dots, X_m \rangle$ , where  $X$  is an instruction of a tested C++ program. For example, in Figure 21,  $\langle 17, 19, 21, 5, 7, 8, 10, 11, 12, 13, 10, 11, 12, 13, 14, 22, 23, 25 \rangle$  is the resulting trajectory when the program in Figure 20 is executed on input data  $\text{Num} = 3$ .

$T = \langle X \mid \text{for all } X, \text{ where } X\text{'s are in a feasible path executed for some input and } X \in P \rangle$

Action	Instruction in action
17 <sup>1</sup>	main() {
19 <sup>2</sup>	int Num;
21 <sup>3</sup>	cin>>Num;
5 <sup>4</sup>	int Fac(int N) {
7 <sup>5</sup>	int F = 1;
8 <sup>6</sup>	int I = 2;
10 <sup>7</sup>	while(I <= N) {
11 <sup>8</sup>	F = F * I;
12 <sup>9</sup>	I++;
13 <sup>10</sup>	}
10 <sup>11</sup>	while(I <= N) {
11 <sup>12</sup>	F = F * I;
12 <sup>13</sup>	I++;
13 <sup>14</sup>	}
14 <sup>15</sup>	return F; <== End of Function
22 <sup>16</sup>	cout<<Fac(Num);
23 <sup>17</sup>	cout<<Num;
25 <sup>18</sup>	} <== End of Function
T	= <17, 19, 21, 5, 7, 8, 10, 11, 12, 13, 10, 11, 12, 13, 14, 22, 23, 25>
TF <sub>main</sub>	= <17, 19, 21, 22, 23, 25>
TF <sub>Fac</sub>	= <5, 7, 8, 10, 11, 12, 13, 10, 11, 12, 13, 14>

Figure 21. A trajectory of the program in Figure 20 on input data  $\text{Num} = 3$

### Definition 4

Let  $TF_{\text{name}}$  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  $T = \langle X_1, X_2, \dots, X_m \rangle$ , then the function trajectory name is denoted by  $TF_{name} = \langle X_i, X_{i+1}, \dots, X_k \rangle$ , where  $X_i, X_{i+1}, \dots, X_k$  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, ( $1 \leq i < k$ , and  $i < k \leq m$ ). For example, in Figure 21,  $\langle 17, 19, 21, 22, 23, 25 \rangle$  is the trajectory of  $TF_{main}$ , and  $\langle 5, 7, 8, 10, 11, 12, 13, 10, 11, 12, 13, 14 \rangle$  is the trajectory of  $TF_{fac}$ , when the program in Figure 20 is executed on input data  $Num = 3$ .

$$TF_{name} = \langle X \mid \text{for all } X, \text{ where } X\text{'s are in a feasible path executed for some input,} \\ X \in F_{name}, \text{ and } X \in T \rangle$$

#### 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 [Korel and Laski 90]. For example,  $11^8$  and  $11^{12}$  in trajectory  $T$  in Figure 21 are actions that involve the same instruction 11. An action  $X^p$  is a *test action* if  $X$  is a test instruction such as **while** or **for**. For example,  $10^7$  and  $10^{11}$  in trajectory  $T$  in Figure 21 are test actions.

#### Definition 6

Let  $M(T)$  be a set of actions in a given trajectory  $T$ , where  $M(T) = \{ X^p : \text{instruction } X \text{ at position } p \text{ in trajectory } T \}$  [Korel and Laski 90]. For example, in Figure 21,  $\{17^1, 19^2, 21^3, 5^4, 7^5, 8^6, 10^7, 11^8, 12^9, 13^{10}, 10^{11}, 11^{12}, 12^{13}, 13^{14}, 14^{15}, 22^{16}, 23^{17}, 25^{18}\}$  is a set of actions  $M(T)$ .

### Definition 7

Let  $M(TF_{name})$  be a set of actions in a given function of a given trajectory  $TF_{name}$ , where  $M(TF_{name}) = \{ X^p : \text{instruction } X \text{ at position } p \text{ in trajectory } TF_{name} \}$ .  $M(TF_{name})$  is a subset of  $M(T)$ . For example, in Figure 21,  $\{17^1, 19^2, 21^3, 22^{16}, 23^{17}, 25^{18}\}$  is a set of actions  $M(TF_{main})$ , and  $\{5^4, 7^5, 8^6, 10^7, 11^8, 12^9, 13^{10}, 10^{11}, 11^{12}, 12^{13}, 13^{14}, 14^{15}\}$  is a set of actions  $M(TF_{Fac})$ .

### Definition 8

Let  $C$  be a slicing criterion, which is the specification for a particular behavior of interest (see Subsection 2.5.2 for more detail). A slicing criterion can be expressed as the values of some set of variables at some set of statements [Weiser 81]. 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  $P$  [Korel and Laski 90].

### Definition 9

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

For example, in the trajectory of Figure 21,

```
213  cin>>Num;
```

Num is a set of variables that are defined in  $21^3$ ,  $D(21^3) = \{Num\}$ .

Let  $DF_{name}(X^p)$  be the set of variables that are defined in action  $X^p$ , where  $X^p \in M(TF_{name})$ . In Figure 21, since  $21^3 \in M(TF_{main})$  and Num is a set of variables that are defined in function `main`,  $DF_{main}(21^3) = \{Num\}$ .

### Definition 10

Let  $U(X^P)$  be the set of variables that are used in action  $X^P$ , where  $X^P \in M(T)$ . For example in the trajectory of Figure 21,

$$10^{11} \quad \text{while}(I \leq N) \{$$

$I$  and  $N$  are the set of variables that are used in  $10^{11}$ ,  $U(10^{11}) = \{I, N\}$ .

Let  $UF_{\text{name}}(X^P)$  be the set of variables that are used in action  $X^P$ , where  $X^P \in M(TF_{\text{name}})$ . From last example, since  $10^{11} \in M(TF_{\text{Fac}})$  and  $I$  and  $N$  are the set of variables that are used in function  $\text{Fac}$ ,  $UF_{\text{Fac}}(10^{11}) = \{I, N\}$ .

### Definition 11

Let  $LF_{\text{name}}(X^P)$  be a set of variables and C++ preprocessors that are declared as a local declaration in function name. For example, in the trajectory of Figure 21,  $LF_{\text{Fac}}(7^5) = \{F\}$ ,  $LF_{\text{Fac}}(8^6) = \{I\}$ , and  $LF_{\text{main}}(19^2) = \{\text{Num}\}$ . There are no local C++ preprocessors in this example.

### 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 [Korel and Laski 90]. For example, in the trajectory of Figure 21,  $11^{12}$  assigns a value to variable  $F$  and  $14^{15}$  use that value. Instead of using  $M(T)$  as Korel and Laski did,  $M(TF_{\text{name}})$  was used in this work in order to compute a slice from functions or classes.

$M(TF_{Fac})$	$DF_{Fac}(X^P)$	$UF_{Fac}(X^P)$	$LF_{Fac}(X^P)$
$5^4$			N
$7^5$			F
$8^6$			I
$10^7$		I, N	
$11^8$	F	F, I	
$12^9$	I	I	
$13^{10}$			
$10^{11}$		I, N	
$11^{12}$	F	F, I	
$12^{13}$	I	I	
$13^{14}$			
$14^{15}$		F	

Figure 22. The sets  $M(TF_{Fac})$ ,  $DF_{Fac}(X^P)$ ,  $UF_{Fac}(X^P)$ , and  $LF_{Fac}(X^P)$  for the trajectory in Figure 21

$M(TF_{main})$	$DF_{main}(X^P)$	$UF_{main}(X^P)$	$LF_{main}(X^P)$
$17^1$			
$19^2$			Num
$21^3$	Num		
$22^{16}$		Num	
$23^{17}$		Num	
$25^{18}$			

Figure 23. The sets  $M(TF_{main})$ ,  $DF_{main}(X^P)$ ,  $UF_{main}(X^P)$ , and  $LF_{main}(X^P)$  for the trajectory in Figure 21

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



Let  $TF_{name} = \langle X_i, X_{i+1}, \dots, X_t, \dots, X_k \rangle$ ,

$X^p \text{ DUF}_{name} Y^t$ ,  $i \leq p < t$ , 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  $TF_{name}$ .

For example, in the trajectory of Figure 21,  $21^3$  is the last definition of variable Num at the execution positions 4 through 18. The  $DUF_{name}$  Relation for the trajectory in Figure 21 is shown in Figures 24 and 25.

### 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. For example, in the trajectory of Figure 21,  $7^5$  declares variable F and  $11^8, 11^{12}$  define and  $11^8, 11^{12}, 14^{15}$  use that variable.

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

Let  $TF_{name} = \langle X_i, X_{i+1}, \dots, X_t, \dots, X_k \rangle$ ,

$X^p \text{ LDRF}_{name} Y^t$ ,  $i \leq p < t$ , 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  $TF_{name}$ .

The  $LDRF_{name}$  Relation for the trajectory in Figure 21 is shown in Figures 26 and 27.

$$\begin{aligned}
\text{DUF}_{\text{Fac}}(11^8) &= \{11^{12}\} \\
\text{DUF}_{\text{Fac}}(12^9) &= \{10^{11}, 11^{12}, 12^{13}\} \\
\text{DUF}_{\text{Fac}}(11^{12}) &= \{14^{15}\} \\
\text{DUF}_{\text{Fac}}(12^{13}) &= \{\}
\end{aligned}$$

Figure 24. The  $\text{DUF}_{\text{Fac}}$  relation for the trajectory depicted in Figure 21

$$\text{DUF}_{\text{main}}(21^3) = \{22^{16}, 23^{17}\}$$

Figure 25. The  $\text{DUF}_{\text{main}}$  relation for the trajectory depicted in Figure 21

$$\begin{aligned}
\text{LDRF}_{\text{Fac}}(5^4) &= \{10^7, 10^{11}\} \\
\text{LDRF}_{\text{Fac}}(7^5) &= \{11^8, 11^{12}, 14^{15}\} \\
\text{LDRF}_{\text{Fac}}(8^6) &= \{10^7, 11^8, 12^9, 10^{11}, 11^{12}, 12^{13}\}
\end{aligned}$$

Figure 26. The  $\text{LDRF}_{\text{Fac}}$  relation for the trajectory depicted in Figure 21

$$\text{LDRF}_{\text{main}}(19^2) = \{21^3, 22^{16}, 23^{17}\}$$

Figure 27. The  $\text{LDRF}_{\text{main}}$  relation for the trajectory depicted in Figure 21

#### Definition 14

Let  $\text{TC}$  be a Test-Control Relation, capturing the effect between test actions and actions that have been chosen to execute by these test actions [Korel and Laski 90]. For example in the trajectory of Figure 21, the scope of test action  $10^7$  influences the execution of  $11^8$ ,  $12^9$ , and  $13^{10}$ , but it does not influence the execution of  $10^{11}$ ,  $11^{12}$ , and  $12^{13}$ . Instead of using  $M(\text{T})$  as Korel and Laski did,  $M(\text{TF}_{\text{name}})$  was used in this work in order to compute a slice from functions or classes. Let  $M(\text{TF}_{\text{name}})$  be a set of actions in a

given trajectory  $TF_{name}$ .  $TCF_{name}$ , a Test-Control-Function<sub>name</sub> Relation, is a binary relation on  $M(TF_{name})$  defined as follows:

Let  $TF_{name} = \langle X_i, X_{i+1}, \dots, X_t, \dots, X_k \rangle$ ,

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

For example, in the trajectory of Figure 21, instructions 11, 12, and 13 are in the scope of influence of test instruction 10, but instructions 17, 19, 21, 5, 7, 8, 14, 22, 23, and 25 are not. The  $TCF_{name}$  Test-Control-Function<sub>name</sub> Relation for the trajectory in Figure 21 is shown in Figure 28.

$TCF_{Fac}(10^7)$	$= \{11^8, 12^9, 13^{10}\}$
$TCF_{Fac}(10^{11})$	$= \{11^{12}, 12^{13}, 13^{14}\}$

Figure 28. The  $TCF_{Fac}$  relation for the trajectory depicted in Figure 21

#### Definition 15

Let  $IRF_{name}$  be an Identity Relation in Function<sub>name</sub>, 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  $TF_{name} = \langle X_i, X_{i+1}, \dots, X_t, \dots, X_q, \dots, X_k \rangle$  denotes a function trajectory,  $q$  is a position in  $TF_{name}$ ,  $1 \leq i < t$ , and  $t < q \leq k$ . The  $IRF_{name}$  Relation for the trajectory in Figure 21 is obtained as shown in Figure 29.

$IRF_{Fac}(10^{17})$	$= \{10^{11}\}$
$IRF_{Fac}(10^{11})$	$= \{10^{17}\}$

Figure 29. The  $IRF_{Fac}$  relation for the trajectory depicted in Figure 21

#### Definition 16

Figure 30 presents 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  $X^{1+1}$ . From Figure 30, we find that  $T = \langle \dots, X^{i-2}, X^{i-1}, X^i, X^{i+1}, X^{i+2}, \dots, X^j, X^{j+1}, \dots, X^k, X^{k+1}, \dots, X^l, X^{l+1}, \dots, X^m, \dots, X^n, X^{n+1}, X^{n+2}, \dots \rangle$ , where  $i < j < k$ ,  $l < m < n$  and  $X$  is any statement in a program  $P$ ,  $TF_{FuncA} = \langle X^i, X^{i+1}, X^{i+2}, \dots, X^j, X^{j+1}, \dots, X^m, \dots, X^n \rangle$ , and  $TF_{FuncB} = \langle X^{j+1}, \dots, X^k, X^{k+1}, \dots, X^l \rangle$ . 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  $X^{1+1}$  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 from a called action first and then from a calling action. For example, in Figure 31, suppose one needs to find a slice

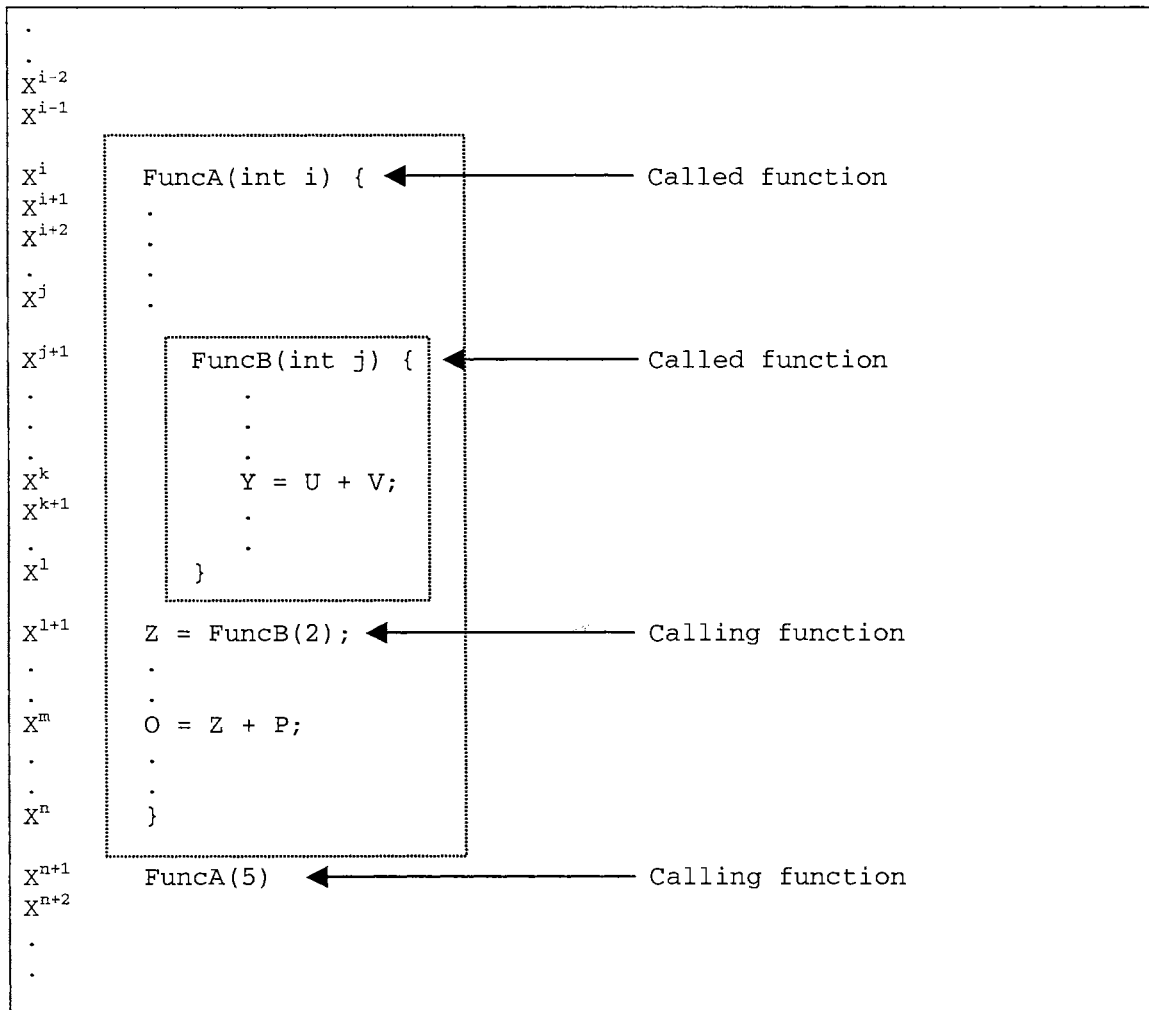


Figure 30. A trajectory of functions A and B where function A calls function B

of variable  $U$  at  $X^k$ . The process starts from  $X^k$  (which is in the scope of influence of called function  $\text{FuncB}(\text{int } j)$ ), which is called by calling function  $\text{FuncB}(2)$  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 32, 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

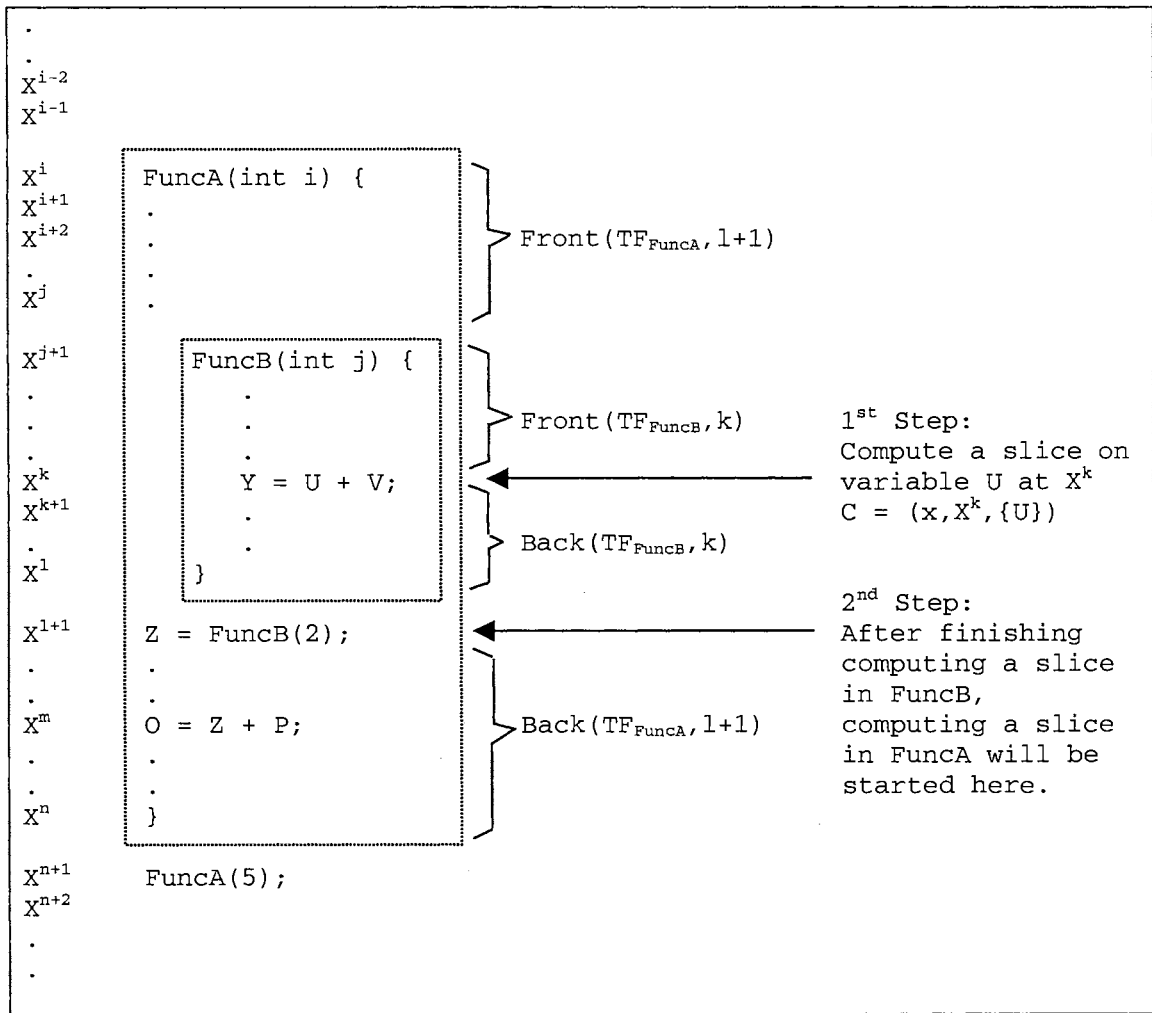


Figure 31. Illustrate Called-to-Calling

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 [Korel and Laski 90], let  $TF_{name} = \langle X_i, X_{i+1}, X_{i+2}, \dots, X_k \rangle$  be a trajectory of function name, and  $q$  be a position in  $TF_{name}$ ,  $i \leq q \leq k$ . Then  $Front(TF_{name}, q)$  is a sublist  $\langle X_i, X_{i+1}, \dots, X_q \rangle$  and  $Back(TF_{name}, q)$  is a sublist  $\langle X_{q+1}, X_{q+2}, \dots, X_k \rangle$  as shown in Figures 31 and 32. All

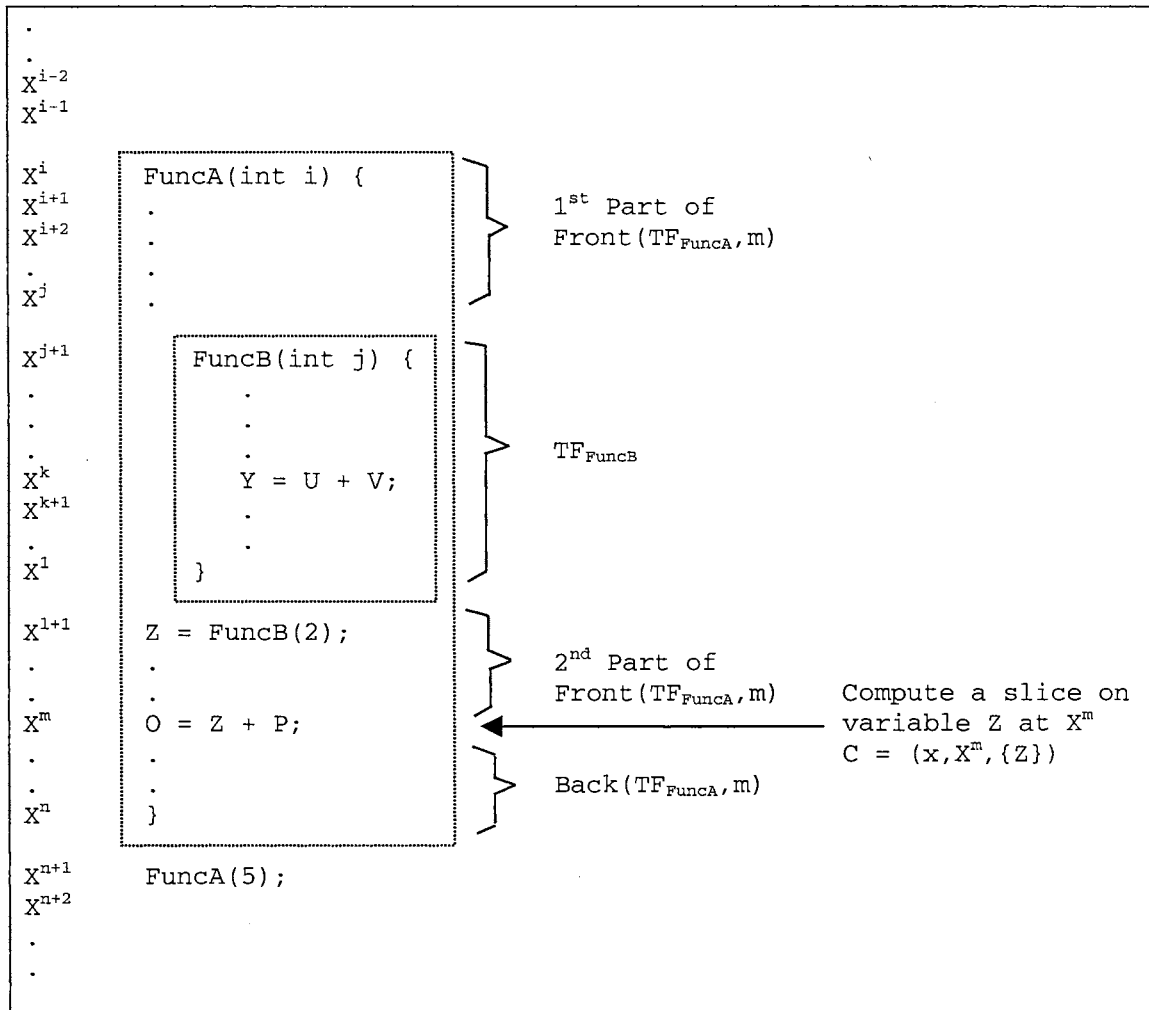


Figure 32. Illustrate Calling-to-Called

$\text{Back}(\text{TF}_{\text{name}}, q)$ 's can be ignored in computing a slice. Just  $\text{Front}(\text{TF}_{\text{name}}, q)$  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)  $\text{Slice}_B$  is a slice computed based on  $\text{Front}(\text{TF}_B, k)$  and slicing criterion  $C = (x, X^k, V)$
- (2)  $\text{Slice}_A$  is a slice computed based on  $\text{Front}(\text{TF}_A, l+1)$  and used variables at calling action  $X^{l+1}, U(X^{l+1})$ .

## 2) Calling-to-Called

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

where

- (1)  $\text{Slice}_A$  is a slice computed based on  $\text{Front}(\text{TF}_A, m)$  and slicing criterion  $C = (x, X^m, V)$
- (2)  $\text{TF}_B$  is a function trajectory of function B.

Let  $\text{Calling}(X^p)$  be a set of calling functions that are used to call a called function in action  $X^p$ , where  $X^p \in M(T)$ . For example in the trajectory of Figure 21,  $\text{Calling}(22^{16}) = \{\text{Fac}\}$ .

Let  $\text{Called}(X^p)$  be a set of called functions that are called by a calling function in action  $X^p$ , where  $X^p \in M(T)$ . For example in the trajectory of Figure 21,  $\text{Called}(5^4) = \{\text{Fac}\}$ .

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:

$$\text{Let } T = \langle X_1, X_2, \dots, X_t, \dots, X_m \rangle,$$

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

- (1) a called function  $f \in \text{Called}(Y^t)$ ,
- (2) a calling function  $f \in \text{Calling}(X^p)$ , and
- (3)  $X^p$  is the calling action, where the calling function  $f$  at  $p$  calls a called function  $f$  at  $t$

For example in the trajectory of Figure 21, we have  $22^{16} EI 5^4$ , as shown in Figure 33.



$$EI(22^{16}) = \{5^4\}$$

Figure 33. The EI relation for the trajectory depicted in Figure 21

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:

$$\text{Let } T = \langle X_1, X_2, \dots, X_t, \dots, X_m \rangle,$$

$X^p$  IE  $Y^t$ ,  $1 \leq p < t$ , iff there exists function  $f$  such that (1) a calling function  $f \in \text{Calling}(Y^t)$ ,  
 (2) a called function  $f \in \text{Called}(X^p)$ , and  
 (3)  $X^p$  is the called action where the called function  $f$  at  $p$  is called by a calling function  $f$  at  $t$

For example in the trajectory of Figure 21, we have  $5^4$  IE  $22^{16}$ , as shown in Figure 34.

$$IE\{5^4\} = \{22^{16}\}$$

Figure 34. The IE relation for the trajectory depicted in Figure 21.

#### 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 [Korel and Laski 90].

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

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  $S^0, S^1, \dots, S^n, 0 \leq n < q$ , which is defined as follows.

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

where

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

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

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

$$\text{where } Z = \text{DU} \cup \text{TC} \cup \text{IR} \cup \text{LDR}$$

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  $\text{FN}(q)$  be a string of function name such that  $X^q, X$  is in the scope of influence. For example in Figure 21,  $\text{TF}_{\text{Fac}} = \{5^4, 7^5, 8^6, 10^7, 11^8, 12^9, 13^{10}, 10^{11}, 11^{12}, 12^{13}, 13^{14}, 14^{15}, 15^{16}\}$ , then  $\text{FN}(8) = \text{"Fac"}$ , because 11 is in the scope of influence of function name Fac.  $\text{FN}(17) = \text{"main"}$  for the same reason.

#### Definition 19

Let  $G(X)$  be a set of variables and precedences that are declared as a part of global declaration.  $G(X)$  is computed from the source program, not from a trajectory path. In Figure 20,  $G(1) = \{\text{include}\}$  and  $G(3) = \{\text{Fac}\}$ .

### Definition 20

Let  $VDU(\text{FunctionName})$  be a set of variables that are used,  $UF_{\text{name}}$ , and defined,  $DF_{\text{name}}$ , in a given function name. For example,  $VDU(\text{main}) = \{\text{Num}\}$  and  $VDU(\text{Fac}) = \{\text{F}, \text{I}, \text{N}\}$  in Figure 21.

### 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  $X_{DCL}$ .

For example in Figure 20, we get  $VS\{11\} = \{7, 8\}$  since variable  $F$  at  $X_{DU} = 11$  was declared in  $X_{DCL} = 7$ , and variable  $I$  at  $X_{DU} = 11$  was declared in  $X_{DCL} = 8$ . The  $VS$  relation for the program depicted in Figure 20 is shown in Figure 36.

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 [Lyle 84].

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 Figure 35 and will still be called the control scope, CS, set.

Based on the rules in Figure 35, Figure 36 shows an example of computing the CS set of a tested program that computes the factorial in Figure 20.

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  $F^0, F^1, \dots, F^n$ ,  $0 \leq i < n$ ,  $n$  = length of program  $P$ , which is defined as follows.

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

where

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

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

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

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 the 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

Figure 35. Rules for computing the CS (control scope) set

$$\text{where } Z = VS \cup CS$$

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  $\{F^i\}$ .

Instruction (X)	Prototype	Called	Calling	D-set	U-set	DCL-set	VS-set	CS-set
1 #include <iostream>						include		
2								
3 int Fac(int N);	Fac(2)					N(1)		
4								
5 int Fac(int N) {		Fac(2)				N(3)	3	15
6								
7 int F = 1;						F(4)		5
8 int I = 2;						I(5)		5
9								
10 while(I <= N) {							5, 8	5, 13
11 F = F * I;				F(4)	N(3), I(5)		7, 8	10
12 I++;				I(5)	F(4), I(5)		8	10
13 }					I(5)			10
14 return F;					F(4)		7	5
15 }								5
16								
17 main() {		main(6)						25
18								
19 int Num;						Num(7)		17
20								
21 cin>>Num;				Num(7)	cin(8)		19	17
22 cout<<Fac(Num);			Fac(10)		Num(7), cout(9)		19	17
23 cout<<Num;					cout(9), Num(7)		19	17
24								
25 }								17

Figure 36. The Prototype, Called, Calling, D, U, DCL, VS, and CS sets for the program depicted in Figure 20

### 3.3 Algorithms

#### 3.3.1 Algorithm for Computing a Slice

Figure 37 presents the algorithm designed and implemented for C++Debug. The algorithm is separated into 4 parts: *Datastructures*, *Initialize*, *PASS I*, and *PASS II*. The *Datastructures* part is shown in Figure 38. 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` (for more detail see Subsection 4.3.1).

*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 function `main()`. `Compute_scope_of_influence(C)` makes the final slice completed by adding some statements that may govern each statement in the slice.

```

Datastructures      // see Figure 38

Begin

  Initialize();      // initialize files, variables, etc.;

  // PASS I
  // compute from source code program P

  Create_Information_Database(P); // see Appendix C

                                // compute trajectory code T
                                // see Definition 3
                                // by using tool named cppttrace
  T = gen_T(P);      // see Subsection 4.3.1

  // 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
      Xp EI Yt; Yt ∈ S[0] // get a new position of its calling function
      C.q = Xp                // 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

```

Figure 37. Algorithm to compute a set of slices



```

Constants
  MaxLine = Maximum line numbers of a source code
  MaxTraj = Maximum linenumber of a trajectory
  VarLength= Maximum number of variables per instruction
  N       = Maximum number of slices

Types
  X       = 1..MaxLine // an instruction in a program, see Definition 1
  Action { // instruction X at position q, see Definition 2
    X     : 1..MaxLine
    q     : 1..MaxTraj
  }
  Variable = string // variable name is a string of characters
  SliceCriterion, LastDef { // slicing criterion, see Definition 8,
    q     : 1..MaxTraj // and last definition, see Definition 16
    V     : set [Variable] // variable V at position q in a trajectory
  }

Variables
  P       : set [X] // a source program , see Definition 1
  Fname   : set [X] // a function, see Definition 2
  T       : list [X] // a trajectory, see Definition 3
  TFname  : list [X] // a function trajectory, see Definition 4
  MT      : set [Action] // a set of Action in trajectory T, see Definition 6
  MTFname : set [Action] // a set of Action in trajectory T, see Definition 6
  C       : SliceCriterion // a slice criterion, see Definition 8
  DFname  : set [Variable] // defined variables, see Definition 9
  UFname  : set [Variable] // used variable, see Definition 10
  LFname  : set [variable] // local var & pre declaration , see Definition 11
  DUFname : set [action] // Definition-Use-FunctionName-Relation, see Def. 12
  LDRFname : set [action] // Local-Declaration-FunctionName-Relation, see Def. 13
  TCFname : set [action] // Test-Control-FunctionName-Relation, see Definition 14
  IRFname : set [action] // Identity-Relation-FunctionName, see Definition 15
  A       : array[1..N] of set [Action] // see Definition 16
  S       : array[1..N] of set [Action] // see Definition 16
  LD      : LastDef // a set of last definition, see Definition 16
  LT      : set [Action] // a set of test actions, see Definition 16
  VS      : set [X] // Variable-Scope, see Definition 21
  CS      : set [X] // Control-Scope, see Definition 21
  Slice   : array[1..N] of set [X] // Slices, see Definition 17
  Dice    : set [X] // a final dice, see Section 3.6

```

Figure 38. Slicing data structures

```

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

Compute_Slice_in_Function_Name(SliceCriterion C) {

    name      = FN(C.q); // get function name, see Definition 18
    TFname    = SubT(LF(C)); // compute a sublist function trajectory, see Definition 4

    DFname    = ComputeDFname(TFname); // compute defined var., see Definition 9
    UFname    = ComputeUFname(TFname); // compute used var., see Definition 10
    DUFname   = ComputeDUFname(TFname); // compute defined used rel., see Def. 12
    TCFname   = ComputeTCFname(TFname); // compute test control rel., see Def. 14
    IRFname   = ComputeIRFname(TFname); // compute identity rel., see Definition 15
    LDRFname  = ComputeLDRFname(TFname); // compute local declaration rel.,
                                         // see Definition 13

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 ( $X^p \in Y^t$ ;  $Y^t \in S$ ) // check Calling-to-Called function
        name = FN(p) // get calling function name
        S = S  $\cup$  TFname // where  $\in$ , a Calling-to-Called function, is an element of S
    return (S);
}

```

Figure 39. Algorithm to compute a slice of each function

```

// function to compute the scope of influence of a slice

Add_Scope_of_Influent(array[1..n] of set[action] S) {

    S = Var_Control_Scope(S); // Add scope of influence to a slice,
                              // see Definition 21

    return S;
}

```

Figure 40. Function to compute the scope of influence of a slice

### 3.4 Examples: How to Compute a Slice of a Program Containing Functions

There are three examples in this section: an example of slicing a program based on variable `Num` (which is in the function `main`), an example of slicing a program based on the calling function `Fac` (in the case of Calling-to-Called function), and an example of slicing a program based on the calling function `I` (in the case of Called-to-Calling function).

Example 1. This example shows how to compute a slice based on variable `Num`, which is in the function `main`. Consider trajectory `T` in Figure 21. Using the criterion  $C = (x, 23^{17}, \{\text{Num}\})$ , we have  $x = (\text{Num}) = (3)$ .

The step-by-step trace of the algorithm in Figure 37 follows.

Step 1:

Compute  $S[0] = S[0] \cup \text{Compute\_Slice\_in\_Function\_Name}(C)$

Step 1.1: // start from `Compute_Slice_in_Function_Name(C)`

$\text{FN}(C.q) = \text{FN}(17) = \text{"main"}$

// therefore compute slice in function "main"

compute  $\text{TF}_{\text{main}} = \{17^1, 19^2, 21^3, 22^{16}, 23^{17}, 25^{18}\}$  // as shown in Figure 21

compute  $\text{DF}_{\text{main}}, \text{UF}_{\text{main}}$  // as shown in Figure 23

compute  $\text{DUF}_{\text{main}}$  // as shown in Figure 25

compute  $\text{TCF}_{\text{main}} = \{\}$  // as shown in Definition 13

compute  $\text{IRF}_{\text{main}} = \{\}$  // as shown in Definition 15

compute  $\text{LDRF}_{\text{main}} = \{\}$  // as shown in Figure 27

Step 1.2:

Compute  $S = \text{ComputeSlice}(DUF_{\text{main}}, TCF_{\text{main}}, IRF_{\text{main}}, LDRE_{\text{main}}, C)$

Since  $C = (x, 23^{17}, \{\text{Num}\})$  // given

$LD(17, \{\text{Num}\}) = \{21^3\}$ ,  $LT(23^{17}) = \{17^1\}$ ,  $I^q = 23^{17}$

$A^0 = \{17^1, 21^3, 23^{17}\}$ ,  $S^0 = \{17^1, 21^3, 23^{17}\}$ ,

$A^1 = \{19^2\}$ ,  $S^1 = \{17^1, 19^2, 21^3, 23^{17}\}$ ,

$A^2 = \{\}$ ,  $S^2 = \{17^1, 19^2, 21^3, 23^{17}\}$ .

$S_c = S^2 = \{17^1, 19^2, 21^3, 23^{17}\}$ .

Step 1.3: Check Calling-to-Called functions

No.

Finally, we get  $S[0] = S[0] \cup S_c = \{17^1, 19^2, 21^3, 23^{17}\}$ .

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

since  $FN(17) = \text{"main"}$  then no more calling functions and break.

Step 3: Add scope of influence

$\text{Slice}[1] = \text{Add\_Scope\_of\_Influence}(S[0])$

Let  $F^0 = S_0 = S[0] = \{17, 19, 21, 23\}$ ,

$F^0 = \{17, 19, 21, 23\}$ ,  $S^0 = \{17, 19, 21, 23\}$ ,

$F^1 = \{1, 25\}$ ,  $S^1 = \{1, 17, 19, 21, 23, 25\}$ ,

$F^2 = \{\}$ ,  $S^2 = \{1, 17, 19, 21, 23, 25\}$ ,

$\text{Slice}[1] = S^2 = \{1, 17, 19, 21, 23, 25\}$ .

And finally, the dynamic slice is shown in Figure 41.

Example 2. This example shows how to compute a slice based on the calling function  $F_{ac}$  (in case of Calling-to-Called function). Consider trajectory  $T$  in Figure 21. Using

```

1 #include <iostream>
17 main() {           // main program
19   int Num;         // number
21   cin>>Num;
23   cout<<Num;       // calling function
25 }

```

Figure 41. A dynamic program slice computed based on variable Num in line 23 of the program in Figure 20

the criterion  $C = (x, 22^{16}, \{\text{Fac}\})$ , we have  $x = (\text{Num}) = (3)$ .

The step-by-step trace of the algorithm in Figure 37 follows.

Step 1:

Compute  $S[0] = S[0] \cup \text{Compute\_Slice\_in\_Function\_Name}(C)$

Step 1.1: // start from  $\text{Compute\_Slice\_in\_Function\_Name}(C)$

$\text{FN}(C.q) = \text{FN}(16) = \text{"main"}$

// therefore compute slice in function "main"

compute  $\text{TF}_{\text{main}} = \{17^1, 19^2, 21^3, 22^{16}, 23^{17}, 25^{18}\}$  // as shown in Figure 21

compute  $\text{DF}_{\text{main}}, \text{UF}_{\text{main}}$  // as shown in Figure 23

compute  $\text{DUF}_{\text{main}}$  // as shown in Figure 25

compute  $\text{TCF}_{\text{main}} = \{\}$  // as shown in Definition 14

compute  $\text{IRF}_{\text{main}} = \{\}$  // as shown in Definition 15

compute  $\text{LDRF}_{\text{main}} = \{\}$  // as shown in Figure 27

Step 1.2:

Compute  $S = \text{ComputeSlice}(\text{DUF}_{\text{main}}, \text{TCF}_{\text{main}}, \text{IRF}_{\text{main}}, \text{LDRF}_{\text{main}}, C)$

Since  $C = (x, 22^{16}, \{\text{Fac}\})$  // given

$$LD(16, \{Fac\}) = \{\}, LT(22^{16}) = \{17^1\}, I^q = 22^{16}$$

$$A^0 = \{17^1, 22^{16}\}, \quad S^0 = \{17^1, 22^{16}\},$$

$$A^1 = \{19^2, 21^3\}, \quad S^1 = \{17^1, 19^2, 21^3, 22^{16}\},$$

$$A^2 = \{\}, \quad S^2 = \{17^1, 19^2, 21^3, 23^{16}\},$$

$$S_c = S^2 = \{17^1, 19^2, 21^3, 22^{16}\},$$

Step 1.3: Check Calling-to-Called functions

Yes, because  $\{5^4\} \in \{22^{16}\},$

$$FN(4) = \text{"Fac"}$$

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

$$TF_{Fac} = \langle 5^4, 7^5, 8^6, 10^7, 11^8, 12^9, 13^{10}, 10^{11}, 11^{12}, 12^{13}, 13^{14}, 14^{15} \rangle,$$

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

$$= \{17^1, 19^2, 21^3, 5^4, 7^5, 8^6, 10^7, 11^8, 12^9, 13^{10}, 10^{11}, 11^{12}, 12^{13}, \\ 13^{14}, 14^{15}, 22^{16}\}.$$

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

since  $FN(16) = \text{"main"}$  then no more calling functions and break.

Step 3: Add scope of influence

$$Slice[1] = Add\_Scope\_of\_Influence(S[0])$$

$$\text{Let } F^0 = S_0 = S[0]$$

$$F^0 = \{5, 7, 8, 10, 11, 12, 13, 14, 17, 19, 21, 22\},$$

$$S^0 = \{5, 7, 8, 10, 11, 12, 13, 14, 17, 19, 21, 22\},$$

$$F^1 = \{1, 3, 15, 25\},$$

$$S^1 = \{1, 3, 5, 7, 8, 10, 11, 12, 13, 14, 15, 17, 19, 21, 22, 25\},$$

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

$$S^2 = \{1, 3, 5, 7, 8, 10, 11, 12, 13, 14, 15, 17, 19, 21, 22, 25\},$$

$$\text{Slice}[1] = S^2 = \{1, 3, 5, 7, 8, 10, 11, 12, 13, 14, 15, 17, 19, 21, 22, 25\}.$$

And finally, the dynamic slice is shown in Figure 42.

```

1 #include <iostream>
3 int Fac(int N);      // function prototype
5 int Fac(int N) {    // called function
7     int F = 1;
8     int I = 2;
10    while(I <= N) {
11        F = F * I;
12        I++;
13    }
14    return F;
15 }
17 main() {           // main program
19     int Num;        // number
21     cin>>Num;
22     cout<<Fac(Num); // calling function
25 }

```

Figure 42. A dynamic program slice computed based on variable Fac in line 22 of the program in Figure 20

Example 3. This example shows how to compute a slice based on the calling function I (in case of Called-to-Calling function). Consider trajectory T in Figure 21. Using the criterion  $C = (x, 8^6, \{I\})$ , we have  $x = (\text{Num}) = (3)$ .

The step-by-step trace of the algorithm in Figure 37 follows.

Step 1:

Compute  $S[0] = S[0] \cup \text{Compute\_Slice\_in\_Function\_Name}(C)$

Step 1.1: // start from  $\text{Compute\_Slice\_in\_Function\_Name}(C)$

$\text{FN}(C.q) = \text{FN}(6) = \text{"Fac"}$

// therefore compute slice in function “Fac”

compute  $TF_{Fac}$  // as shown in Figure 21

compute  $DF_{Fac}, UF_{Fac}$  // as shown in Figure 22

compute  $DUF_{Fac}$  // as shown in Figure 24

compute  $TCF_{Fac} = \{\}$  // as shown in Definition 14

compute  $IRF_{Fac} = \{\}$  // as shown in Definition 15

compute  $LDRF_{Fac} = \{\}$  // as shown in Figure 26

Step 1.2:

Compute  $S = \text{ComputeSlice}(DUF_{Fac}, TCF_{Fac}, IRF_{Fac}, LDRF_{Fac}, C)$

Since  $C = (x, 8^6, \{I\})$  // given

$LD(6, \{I\}) = \{\}, LT(8^6) = \{5^4\}, I^q = 8^6$

$A^0 = \{5^4, 8^6\}, S^0 = \{5^4, 8^6\},$

$A^1 = \{\}, S^1 = \{5^4, 8^6\},$

$S_c = S^1 = \{5^4, 8^6\}.$

Step 1.3: Check Calling-to-Called functions

No.

Finally, we get  $S[0] = S[0] \cup S_c = \{5^4, 8^6\}.$

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

since  $FN(6) = \text{“Fac”}$ , there is more calling functions

since  $22^{16} \in I 5^4$ , then  $C.q = 16$ ; Go to Step 1

Step 1:

Compute  $S[0] = S[0] \cup \text{Compute\_Slice\_in\_Function\_Name}(C)$

Step 1.1: // start from  $\text{Compute\_Slice\_in\_Function\_Name}(C)$



$\text{FN}(\text{C.q}) = \text{FN}(16) = \text{"main"}$

// therefore compute slice in function "main"

compute  $\text{TF}_{\text{main}} = \{17^1, 19^2, 21^3, 22^{16}, 23^{17}, 25^{18}\}$  // as shown in Figure 21

compute  $\text{DF}_{\text{main}}, \text{UF}_{\text{main}}$  // as shown in Figure 23

compute  $\text{DUF}_{\text{main}}$  // as shown in Figure 25

compute  $\text{TCF}_{\text{main}} = \{\}$  // as shown in Definition 14

compute  $\text{IRF}_{\text{main}} = \{\}$  // as shown in Definition 15

compute  $\text{LDRF}_{\text{main}} = \{\}$  // as shown in Figure 26

Step 1.2:

Compute  $S = \text{ComputeSlice}(\text{DUF}_{\text{main}}, \text{TCF}_{\text{main}}, \text{IRF}_{\text{main}}, \text{LDRF}_{\text{main}}, C)$

Since  $C = (x, 22^{16}, \{\text{Fac}\})$  // given

$\text{LD}(16, \{\text{Fac}\}) = \{\}$ ,  $\text{LT}(22^{16}) = \{17^1\}$ ,  $\text{I}^q = 22^{16}$

$A^0 = \{17^1, 22^{16}\}$ ,  $S^0 = \{17^1, 22^{16}\}$ ,

$A^1 = \{19^2, 21^3\}$ ,  $S^1 = \{17^1, 19^2, 21^3, 22^{16}\}$ ,

$A^2 = \{\}$ ,  $S^2 = \{17^1, 19^2, 21^3, 23^{16}\}$ ,

$S_c = S^2 = \{17^1, 19^2, 21^3, 22^{16}\}$ .

Step 1.3: Check Calling-to-Called functions

No.

Finally, we get  $S[0] = S[0] \cup S_c = \{21^3, 5^4, 8^6, 22^{16}\}$ .

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

since  $\text{FN}(16) = \text{"main"}$ , no more calling functions and break.

Step 3: Add scope of influence

$\text{Slice}[1] = \text{Add\_Scope\_of\_Influence}(S[0])$

$F^0 = S_0 = S[0] = \{5, 8, 17, 19, 21, 22\},$   
 $F^0 = \{5, 8, 17, 19, 21, 22\},$   
 $S^0 = \{5, 8, 17, 19, 21, 22\},$   
 $F^1 = \{1, 3, 15, 25\},$   
 $S^1 = \{1, 3, 5, 8, 15, 17, 19, 21, 22, 25\},$   
 $F^2 = \{ \},$   
 $S^2 = \{1, 3, 5, 8, 15, 17, 19, 21, 22, 25\},$   
 $\text{Slice}[1] = S^2 = \{1, 3, 5, 8, 15, 17, 19, 21, 22, 25\}.$

And finally, the dynamic slice is shown in Figure 43.

```

1 #include <iostream>
3 int Fac(int N);      // function prototype
5 int Fac(int N) {    // called function
8   int I = 2;
15 }
17 main() {           // main program
19   int Num;         // number
21   cin>>Num;
22   cout<<Fac(Num); // calling function
25 }

```

Figure 43. A dynamic program slice computed based on variable  $I$  in line 8 of the program in Figure 20

### 3.5 A Slice with Classes, Structures, and Unions

A class contains members, variables, and functions. Each slice of the member functions is computed in the same way as a normal function mentioned in Section 3.4. After a slice of a member function is computed, the rest of the slice code in the class is determined by variable scope  $VS$  and control scope  $CS$ . The  $VS$  and  $CS$  sets are the key to obtaining a slice program of a program with classes. A slice of a program with

Structures and Unions is computed the same way as a slice of a program with classes, since all have the same grammar structures.

The program in Figure 44 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 44 is shown in Figure 45.

```

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)
13:                 Num[I] = N[I];
14:         }
15:
16:         float Sum(void) {
17:             float Tsum = 0;
18:             for(int I=0; I<Max; ++I)
19:                 Tsum = Tsum + Num[I];
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: }

```

Figure 44. A program for calculating the sum and average of a set of numbers

```

281  main() {
292      int Max = 4;
303      float Num[4] = {10.0, 20.0, 15.0, 5.0};

94      Compute (int M, float *N) {
105          Max = M; allocate mem
116          cout<<"allocate mem"<<endl;
127          for(int I=0; I<Max; ++I)
              Num[0] = N[0];
128          for(int I=0; I<Max; ++I)
              Num[1] = N[1];
129          for(int I=0; I<Max; ++I)
              Num[2] = N[2];
1210         for(int I=0; I<Max; ++I)
              Num[3] = N[3];
1411     }

3112    Compute A(Max, Num);

1613    float Sum(void) {
1714        float Tsum = 0;
1815        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[0];
1816        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[1];
1817        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[2];
1818        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[3];
2019        return Tsum; 50

3220    cout<<A.Sum()<<endl;

2321    float Avg(void) {
2422        return Sum()/(Max + 1);

1623    float Sum(void) {
1724        float Tsum = 0;
1825        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[0];
1826        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[1];
1827        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[2];
1828        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[3];
2029        return Tsum; 10

3330    cout<<A.Avg()<<endl;
3431 }

T      = <281, 292, 303, 94, 105, 116, 127, 128, 129, 1210, 1411, 3112, 1613, 1714,
      1815, 1816, 1817, 1818, 2019, 3220, 2321, 2422, 1623, 1724, 1825, 1826,
      1827, 1828, 2029, 3330, 3431>

TFMain   = <281, 292, 303, 3112, 3220, 3330, 3431>
TFCompute = <94, 105, 116, 127, 128, 129, 1210, 1411>
TFSum(1)  = <1613, 1714, 1815, 1816, 1817, 1818, 2019>
TFSum(2)  = <1623, 1724, 1825, 1826, 1827, 1828, 2029>
TFAvg     = <2321, 2422>

```

Figure 45. The trajectory of the program from Figure 44 on input data  
Max = 4, Num = (10.0, 20.0, 15.0, 5.0)

Instruction (X)	Prototype	Called	Calling	D-set	U-set	DCL-set	VS-set	CS-set
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: }	Compute(01)					include		26
		Compute(01)		Max(02) Num(03) I(09)	M(04) cout(07) endl(08) Max(02) N(05) I(09)	M(04) N(05) I(09)	5, 9 5,6,9	3,14 9 9 9
		Sum(10)		Tsum(11) I(12)	Max(02) Num(03) Tsum(11) I(12)	Tsum(11) I(12)	5,6,17	3,21 16 16
		Avg(13)	Sum(10)		Max(02)		5	3,25 23 23 3
		main(14)		Compute(01) Sum(19) Avg(20)	Max(15) Num(16) cout(07) endl(08) A(17) cout(07) endl(08) A(17)	Max(15) Num(16) A (17)	9,29,30 31 31	34 28 28 28 28 28

Figure 46. The Prototype, Called, Calling, D, U, DCL, VS, and CS sets for the program depicted in Figure 44

$$\begin{aligned} \text{DUF}_{\text{Main}} &= \{\} \\ \text{TCF}_{\text{Main}} &= \{\} \\ \text{IRF}_{\text{Main}} &= \{\} \end{aligned}$$

$$\begin{aligned} \text{LDF}_{\text{Main}}(29^2) &= \{31^{12}\} \\ \text{LDF}_{\text{Main}}(30^3) &= \{31^{12}\} \end{aligned}$$

Figure 47. The  $\text{DUF}_{\text{Main}}$ ,  $\text{TCF}_{\text{Main}}$ ,  $\text{LDF}_{\text{Main}}$ , and  $\text{IRF}_{\text{Main}}$  relations that are called by  $32^{20}$  for the trajectory depicted in Figure 45

$$\begin{aligned} \text{DUF}_{\text{Compute}}(10^5) &= \{12^7\} & \text{IRF}_{\text{Compute}}(12^7) &= \{12^8, 12^9, 12^{10}\} \\ \text{DUF}_{\text{Compute}}(12^7) &= \{\} & \text{IRF}_{\text{Compute}}(12^8) &= \{12^7, 12^9, 12^{10}\} \\ \text{TCF}_{\text{Compute}} &= \{\} & \text{IRF}_{\text{Compute}}(12^9) &= \{12^7, 12^8, 12^{10}\} \\ & & \text{IRF}_{\text{Compute}}(12^{10}) &= \{12^7, 12^8, 12^9\} \\ \text{LDF}_{\text{Compute}}(9^4) &= \{10^5, 12^7, 12^8, 12^9, 12^{10}\} \end{aligned}$$

Figure 48. The  $\text{DUF}_{\text{Compute}}$ ,  $\text{TCF}_{\text{Compute}}$ ,  $\text{LDF}_{\text{Compute}}$ , and  $\text{IRF}_{\text{Compute}}$  relations that are called by  $32^{20}$  for the trajectory depicted in Figure 45

$$\begin{aligned} \text{DUF}_{\text{Sum}}(18^{15}) &= \{18^{16}\} & \text{IRF}_{\text{Sum}}(18^{15}) &= \{18^{16}, 18^{17}, 18^{18}\} \\ \text{DUF}_{\text{Sum}}(18^{16}) &= \{18^{17}\} & \text{IRF}_{\text{Sum}}(18^{16}) &= \{18^{15}, 18^{17}, 18^{18}\} \\ \text{DUF}_{\text{Sum}}(18^{17}) &= \{18^{18}\} & \text{IRF}_{\text{Sum}}(18^{17}) &= \{18^{15}, 18^{16}, 18^{18}\} \\ \text{DUF}_{\text{Sum}}(18^{18}) &= \{20^{19}\} & \text{IRF}_{\text{Sum}}(18^{18}) &= \{18^{15}, 18^{16}, 18^{17}\} \\ \text{TCF}_{\text{Sum}} &= \{\} \\ \text{LDF}_{\text{Sum}}(17^{14}) &= \{18^{15}, 18^{16}, 18^{17}, 18^{18}, 20^{19}\} \end{aligned}$$

Figure 49. The  $\text{DUF}_{\text{Sum}}$ ,  $\text{TCF}_{\text{Sum}}$ ,  $\text{LDF}_{\text{Sum}}$ , and  $\text{IRF}_{\text{Sum}}$  relations that are called by  $32^{20}$  for the trajectory depicted in Figure 45

$$\begin{aligned} \text{DUF}_{\text{Sum}}(18^{25}) &= \{18^{26}\} & \text{IRF}_{\text{Sum}}(18^{25}) &= \{18^{26}, 18^{27}, 18^{28}\} \\ \text{DUF}_{\text{Sum}}(18^{26}) &= \{18^{27}\} & \text{IRF}_{\text{Sum}}(18^{26}) &= \{18^{25}, 18^{27}, 18^{28}\} \\ \text{DUF}_{\text{Sum}}(18^{27}) &= \{18^{28}\} & \text{IRF}_{\text{Sum}}(18^{27}) &= \{18^{25}, 18^{26}, 18^{28}\} \\ \text{DUF}_{\text{Sum}}(18^{28}) &= \{20^{29}\} & \text{IRF}_{\text{Sum}}(18^{28}) &= \{18^{25}, 18^{26}, 18^{27}\} \\ \text{TCF}_{\text{Sum}} &= \{\} \\ \text{LDF}_{\text{Sum}}(17^{24}) &= \{18^{25}, 18^{26}, 18^{27}, 18^{28}, 20^{29}\} \end{aligned}$$

Figure 50. The  $\text{DUF}_{\text{Sum}}$ ,  $\text{TCF}_{\text{Sum}}$ ,  $\text{LDF}_{\text{Sum}}$ , and  $\text{IRF}_{\text{Sum}}$  relations that are called by  $24^{22}$  for the trajectory depicted in Figure 45

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

Figure 51. The  $DUF_{Avg}$ ,  $TCF_{Avg}$ , and  $IRF_{Avg}$  relations for the trajectory depicted in Figure 45

Example 4. Consider trajectory T in Figure 45. 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 37 follows.

Step 1:

Compute  $S[0] = S[0] \cup \text{Compute\_Slice\_in\_Function\_Name}(C)$

Step 1.1: // start from  $\text{Compute\_Slice\_in\_Function\_Name}(C)$

$FN(C.q) = FN(30) = \text{"main"}$

// therefore compute slice in function "main"

compute  $TF_{main}$  // as shown in Figure 47

compute  $LDF_{main}$  // as shown in Figure 47

compute  $DUF_{main}$  // as shown in Figure 47

compute  $TCF_{main} = \{ \}$  // as shown in Definition 47

compute  $IRF_{main} = \{ \}$  // as shown in Definition 47

compute  $LDRF_{main} = \{ \}$  // as shown in Definition 47

Step 1.2:

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

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

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

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

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

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

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

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

Step 1.3: Check Calling-to-Called functions

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

$FN(4) = \text{“Compute”}$ , and  $FN(21) = \text{“Avg”}$ ,

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

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

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

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

since  $\{16^{23}\} \text{ IE } \{24^{22}\}$ ,

$FN(23) = \text{“Sum”}$ ,

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

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

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  $FN(30) = \text{“main”}$  then no more calling functions and break.



Step 3: Add scope of influence

$$\text{Slice}[1] = \text{Add\_Scope\_of\_Influence}(S[0])$$

$$\text{Let } F^0 = S_0 = S[0]$$

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

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

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

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

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

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

$$F^3 = \{\},$$

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

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

And finally, the dynamic slice is shown in Figure 52.

Example 5. Consider trajectory T in Figure 45. Using the criterion  $C = (x, 32^{20}, \{\text{Sum}\})$ , we have  $x = (\text{Max}, \text{Num}) = (3, (10.0, 20.0, 15.0, 5.0))$ .

The step-by-step trace of the algorithm in Figure 37 follows.

Step 1:

$$\text{Compute } S[0] = S[0] \cup \text{Compute\_Slice\_in\_Function\_Name}(C)$$

```

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)
13:                 Num[I] = N[I];
14:         }
16:         float Sum(void) {
17:             float Tsum = 0;
18:             for(int I=0; I<Max; ++I)
19:                 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: }

```

Figure 52. A dynamic program slice computed based on variable Avg in line 33 of the program in Figure 44

Step 1.1: // start from Compute\_Slice\_in\_Function\_Name(C)

$FN(C.q) = FN(20) = \text{"main"}$

// therefore compute slice in function "main"

compute  $TF_{main}$  // as shown in Figure 47

compute  $LDF_{main}$  // as shown in Figure 47

compute  $DUF_{main}$  // as shown in Figure 47

compute  $TCF_{main} = \{\}$  // as shown in Definition 47

compute  $IRF_{main} = \{\}$  // as shown in Definition 47

compute  $LDRF_{main} = \{\}$  // as shown in Definition 47

Step 1.2:

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

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

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

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

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

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

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

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

Step 1.3: Check Calling-to-Called functions

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

$FN(4) = \text{"Compute"}$ , and  $FN(13) = \text{"Sum"}$ ,

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

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

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

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

$16^{23}, 17^{24}, 18^{25}, 18^{26}, 18^{27}, 18^{28}, 20^{29}, 32^{20}\}$ ,

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  $FN(20) = \text{"main"}$  then no more calling function and break

Step 3: Add scope of influence

$\text{Slice}[1] = \text{Add\_Scope\_of\_Influence}(S[0])$

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

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

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

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

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

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

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

$$F^3 = \{\},$$

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

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

And finally, the dynamic slice is shown in Figure 53.

```

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)
13:                 Num[I] = N[I];
14:         }
16:         float Sum(void) {
17:             float Tsum = 0;
18:             for(int I=0; I<Max; ++I)
19:                 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: }

```

Figure 53. A dynamic program slice computed based on variable Sum in line 32 of the program in Figure 44

### 3.6 Problems and Situations in C++ That Were Taken into Account in the Design

There are eight major problems and situations in C++ that were taken into account in the design of C++Debug. They are discussed below.

1. Problems and situations with classes and objects such as classes, structures, unions, anonymous unions, friend functions, friend classes, inline functions, defining inline functions within a class, parameterized constructors, static class members, static data members, static member functions, the scope resolution operator, nested classes, local classes, passing objects to functions, returning objects, and object assignment.
2. Problems and situations with arrays, pointers, references, and the dynamic allocation operators such as arrays of objects, uninitialized arrays, pointers to objects, type checking C++ pointers, the `this` pointer, pointers to derived types, pointers to class members, reference parameters, passing references to objects, returning references, independent references, references to derived types, restrictions to references, dynamic allocation operators (i.e., the `new` operator in C++), initializing allocated memory, allocating arrays, allocating objects, the `nothrow` alternative, and the placement forms of `new` and `delete`.
3. Problems and situations with function overloading, copy constructors, and default arguments such as function overloading, overloading constructor functions, overloading a constructor to gain flexibility, initialized and uninitialized objects, copy constructors, finding the address of an overloaded function, the `overload` anachronism, default function arguments, default arguments vs. overloading, using default arguments correctly, and function overloading and ambiguity.

4. Problems and situations with operator overloading such as operator overloading using a friend function, using a friend to overload ++ or --, friend operator functions adding flexibility, overloading new and delete, overloading new and delete for arrays, overloading the nothrow version of new and delete, overloading some special operators, overloading [], overloading ( ), overloading ->, and overloading the comma operator.

5. Problems and situations with inheritance such as base-class access control, inheritance and protected members, protected base-class inheritance, inheriting multiple base classes, constructors, destructors, inheritance, passing parameters to base-class constructors, granting access, and virtual base classes.

6. Problems and situations with virtual functions and polymorphism such as virtual functions, calling a virtual function through a base class reference, the inherited virtual attribute, hierarchical virtual functions, pure virtual functions abstract classes, and late binding.

7. Problems and situations with templates such as generic functions, a function with two generic types, explicitly overloading a generic function, overloading a function template, using standard parameters with template functions, generic function restrictions, applying generic functions, a generic sort, compacting an array, generic classes, a generic array class, using non-type arguments with generic classes, using default arguments with template classes, explicit class specializations, and the typename and export keywords.

8. Problems and situations with exception handling such as exception handling fundamentals, catching class types, using multiple catch statements, handling derived-

class exceptions, exception handling captions, catching all exceptions, restricting exceptions, rethrowing an exception, `terminate()` and `unexpected()`, the `uncaught_exception()` function, and the `exception` and `bad_exception` classes.

### 3.7 Dicing Procedures

Dicing [Lyle 84] [Nanja 90] is the process of identifying a set of statements likely to contain an error. A dice is determined as follows:

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

Example 6. Observe that a dynamic program slice in Example 4 is a subset of KBI, while a dynamic program slice in Example 5 is a subset of CSF. Consequently, using the definition of dicing, a dice program can be shown as follows

```

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

```

Figure 54. The final program segment after slicing and dicing

Once the procedure is finished, line 24 will be shown as the incorrect line.

## CHAPTER IV

### C++DEBUG

#### 4.1 Introduction

C++Debug is an interactive debugging tool designed to function as a utility program of the UNIX system. C++Debug was developed based on slicing and dicing techniques. In order for C++Debug to be more powerful, dynamic slicing rather than static slicing was chosen for implementation. C++Debug was designed in a way to allow ease and convenience on the part of the user. Using C++Debug, the user can interact directly with the computer in locating errors in a program. Menus are provided to allow the user to select any one of a number of functions (Slice, Dice, Help, etc.) supported by C++Debug.

To produce the C++Debug tool, three activities of a software process are introduced: software specification, software development, and software validation. Some parts of the waterfall approach are used to take those three activities and represent them as separate process phases: requirements specifications, software design, implementation, testing, and valuation. In order to make C++Debug a good piece of software, essential attributes such as maintainability, dependability, efficiency, and usability were considered.



## 4.2 Software specification

According to Sommerville [Sommerville 01], the intention of this phase is to establish what services are required from C++Debug and the constraints on C++Debug's operation and development. The requirements document of C++Debug is shown in Appendix D.

## 4.3 Software Design and Implementation

In order to convert the C++Debug software specification, mentioned above in Section 4.2, into an executable system, architectural design, abstract specification, interface design, component design, datastructure design, and algorithm design were carried out [Sommerville 01]. However, because of the limitation of the size of this dissertation, only a few parts are introduced in the following subsections.

### 4.3.1 C++Debug Block Diagram

C++Debug is comprised of four parts: *Cpptrace*, *Database*, *Slicer*, and *Dicer* (as shown in Figure 55).

1. *Cpptrace* was designed as a tool allowing one to follow the execution of a C++ program, statement-by-statement. *Cpptrace* reads the C++ source program in a file, inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to generate two major parts: 1. a trajectory of the program and 2. some databases, where a trajectory is a feasible path that has actually been executed for some input and the databases are a list of reserved words, a list of basic types, identifier information, types, symbol tables, and

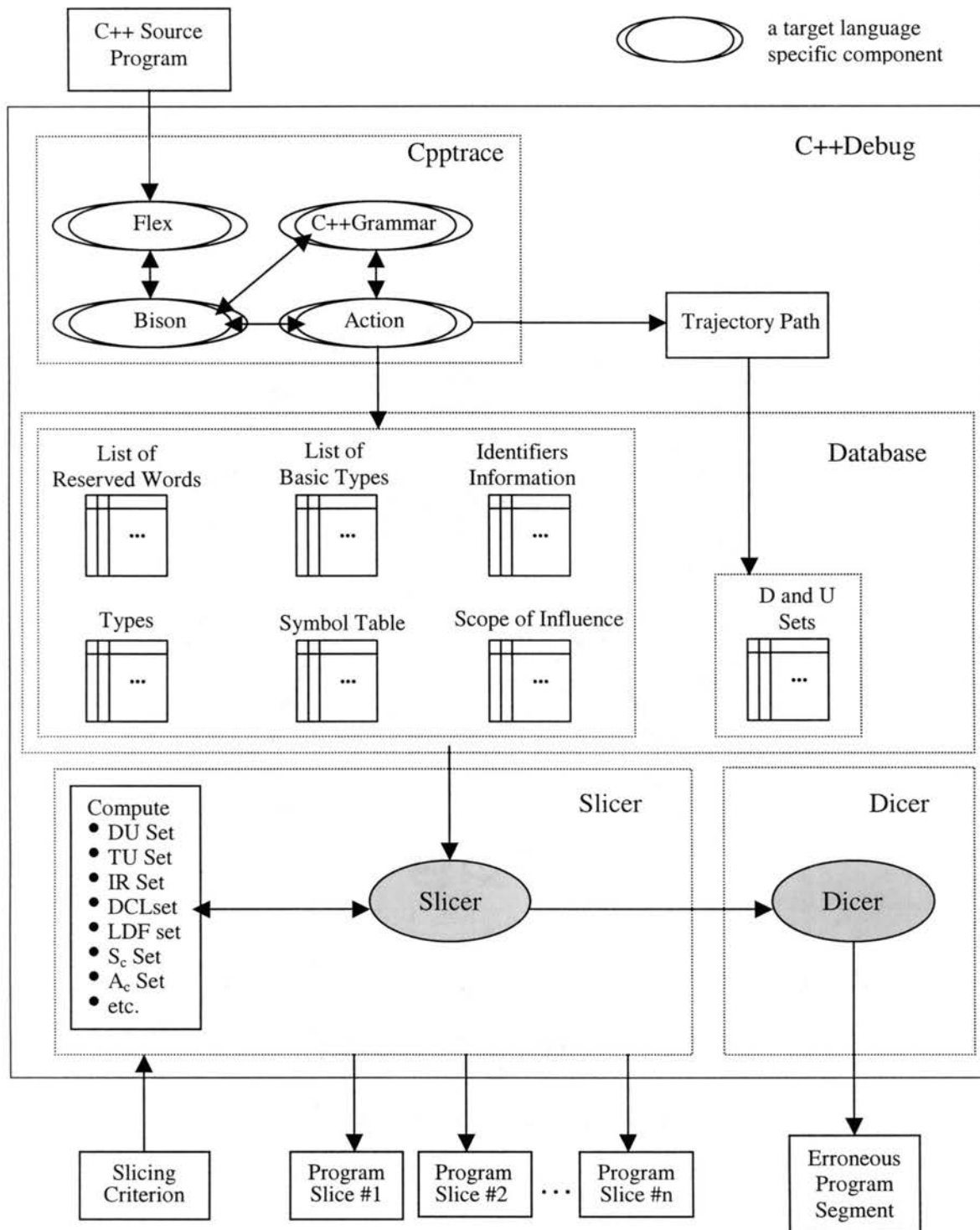


Figure 55. Block diagram of C++Debug

scope of influence. flex and bison are tools used to implement Cpptrace. flex reads

a specification file containing regular expressions for pattern matching and generates a C or C++ routine that performs lexical analysis [Flex 01]. This routine reads a stream of characters and matches sequences that identify tokens. Bison reads a specification file that codifies the grammar of a language and generates a parsing routine [Bison 00]. This routine groups tokens into meaningful sequences and invokes action routines to act upon them. C++ grammar from Stroustrup's textbook was used in this implementation [Stroustrup 97].

2. *Database* stores ordered sets of data such as a list of reserved words, a list of basic types, identifier information, types, symbol tables, and scope of influence, etc. All data are created by Cpptrace as a database. The D and U ordered sets of data are computed from the trajectory path. This database is used by Slicer to compute a program slice(s).

3. *Slicer* was created by using the algorithms in Figure 37. The number of program slices is dependent on the slicing criterion.

4. *Dicer* was created by using the techniques mentioned in Section 3.6.

#### 4.3.2 Datastructures

The datastructures of a source program, functions, a trajectory, sets such as D, U, DU, DCL, etc. were implemented based on datastructures shown in Figure 38.

#### 4.3.3 Symbol Tables

Symbol tables were designed by following the concepts of symbol tables that are used in cool, the Classroom Object-Oriented Language [Cool 94]. cool is a small language designed for use in an undergraduate compiler course project at the University of California at Berkeley [Cool 94]. The key is two functions: `enterscope()` and

`exitscope()`. Function `enterscope()` makes the table point to a new scope whose parent is the scope it pointed to previously, while function `exitscope()` makes the table point to the parent scope.

## 4.4 Testing and Evaluation

### 4.4.1 Introduction

After C++Debug was implemented, the testing process was applied to verify that each unit met its specification (unit testing) and to ensure that the software requirements had been met (integration and system testing) [Sommerville 01]. Testing is the primary means for showing that the implementation has the requisite functionality and other non-functional properties [McDermid 93].

### 4.4.2 Testing

Each problem and situation in Section 4.2 was tested independently upon completion of the tool. C++Debug was also tested on non-trivial programs containing several problems and situations identified. For more information see Appendix E.

### 4.4.3 Evaluation

C++Debug was evaluated by a number of graduate students at the Computer Science of Oklahoma State University. They used C++Debug to locate errors in their programs. For more information see Appendix E.

## 4.5 Limitations

C++Debug has some limitations as listed bellow.

1. Limitation of OS : UNIX
2. Limitation of language: GNU G++
3. Limitations of algorithm: worst-case  $O(N^2V)$ , average-case  $O(N \log N)$ , best-case  $O(N)$ , where  $N$  is the #LOC of the trajectory part, and  $V$  is the maximum number of variables in each line in a debugged program.
4. In the current implementation, limitation of #LOC of the executable part: 1,000.

#### 4.6 Program Documentation

The main purpose of program documentation is to communicate with other people about a finished program [Hedrick 75]. In this study, program documentation for C++Debug was prepared in two parts. The first part involves comments internal to the program. The second part is an auxiliary paper accompanying the program that is included in Chapter III on Software Design. Furthermore, a user's manual was prepared for the convenience of the users of C++Debug.

#### 4.7 System Evolution

System evolution describes the system base, anticipated change due to hardware and software evolution, and the changing user needs [Sommerville 01].

##### 1. System Base

C++Debug is a slicing and dicing based debugging tool for C++ which runs under UNIX on the SUN machine in the Computer Science Department at OSU.

##### 2. Anticipated Change Due to Hardware Evolution

C++Debug is designed to be a portable tool. It is a machine independent tool. It can run on every hardware with a UNIX run-time support. However, C++Debug should be provided on PC as well.

### 3. Anticipated Change Due to Software Evolution

In case of ANSI C++ is updated e.g., if new functions or instructions are added, C++Debug must be updated too.

### 4. Changing User Needs

C++Debug was designed by using menus in a way to allow ease and convenience on the part of the user. C++Debug should be provided in a windowing environment as well.

## 4.8 Slicing-Based Metrics

Program slicing is applied to two main areas [Weiser 81]. First, program slicing is used for debugging and maintenance purposes. This is due to the fact that the size of a resulting slice is relatively smaller than the original program in general, thus making it easier to locate errors or to modify the program at the stage of program maintenance. Second, program slicing is used to obtain slicing based program metrics. It allows the analysis of the structure of the program. Weiser proposed three slicing-based program metrics.

- i. *Coverage* compares the length of slices to the length of the entire program. Coverage might be expressed as the ratio of mean slice length to program length. A low coverage value, indicating a long program with many short slices, may indicate a program which has several distinct conceptual purposes.
- ii. *Overlap* is a measure of how many statements in a slice are found only in that slice. This could be computed as the mean

of the ratios of non-unique to unique statements in each slice. A high overlap might indicate very interdependent code.

- iii. *Clustering* reveals the degree to which slices are reflected in the original code layout. It could be expressed as the mean of the ratio of statements formerly adjacent to total statements in each slice. A low cluster value indicates slices intertwined like spaghetti, while a high cluster value indicates slices physically reflected in the code by statement grouping.

In order to compare the output obtained using C++Debug (which is dynamic slicing based) with the output obtained using C-Sdicer (which is static slicing based), the test programs must be the same ones as used in Nanja's study in testing C-Sdicer [Nanja 90]. These test programs are listed in Appendix F. The number of output variables and the size of each program is shown in Table I.

The results obtained from C-Sdicer and C++Debug are shown in Tables II and III, respectively.

TABLE I  
DESCRIPTION OF THE FIVE TEST PROGRAMS

Metric	P1	P2	P3	P4	P5
Size (# of lines)	120	35	56	67	58
# of output variables	26	3	3	10	1

(Source: [Nanja 90])

TABLE II  
SLICING-BASED METRICS OBTAINED FROM C-SDICER  
FOR THE FIVE TEST PROGRAMS

Metric	P1	P2	P3	P4	P5
Coverage	0.86	0.77	0.57	0.75	0.83
Overlap	0	4.42	10.13	0	1.00
Clustering	0.66	0.64	0.87	0.65	0.95

(Source: [Nanja 90])

TABLE III  
SLICING-BASED METRICS OBTAINED FROM C++DEBUG  
FOR THE FIVE TEST PROGRAMS

Metric	P1	P2	P3	P4	P5
Coverage	0.26	0.48	0.58	0.35	0.70
Overlap	52.33	3.60	14.60	57.00	1.00
Clustering	0.06	0.44	0.30	0.11	0.42



## CHAPTER V

### SUMMARY, CONCLUSIONS, AND FUTURE WORK

#### 5.1 Summary

Chapter I discusses the necessity of using debugging tools in locating and correcting the errors contained in programs. Included in this chapter are the purposes of the study as well as the organization of the study.

Chapter II describes the general knowledge on program slicing and dicing techniques. The chapter concludes with a discussion of both advantages and disadvantages of dynamic slicing and static slicing, and the procedures used to locate errors in a program using dynamic slicing and dicing techniques.

Chapter III presents 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.

Chapter IV presents the steps involved in producing the C++Debug tool. The C++Debug block diagram, the results of the experiment, slicing-based metrics, testing and evaluation, documents, and the advantages and limitations of C++Debug were presents also.

## 5.2 Conclusions

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 certain 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. C++Debug can work on both C and C++.

By using the `-g` option, C++Debug supports the generation of grammar derivation trees. A users can study how the parser checks the syntax of a program. By using the `-i` option, all information about C++Debug can be displayed. One who is interesting in the dynamic slicing area can use the information provided by C++Debug, such as D, U, DU, symbol tables, etc., to investigate the process of slicing, dicing, or compiling in general.

## 5.3 Future Work

Based on the initial experiments with C++Debug, we found that improvements and additions can be made to C++Debug in the following aspects.

### 5.3.1 Improvements

The size of C++Debug after compiling by an optimized compiler is 2,088,720 bytes. It appears that it should be smaller if some algorithms and memory uses are

managed better. Time and space complexities are dependent on the size of the trajectory (and not necessarily the size of the source code). To avoid running out of disk space (which is needed to store the trajectory path), the user must know how far the trajectory must go and how much disks space is required. It would be better if C++Debug can automatically check and tell the user about the sufficiency of the disk space. And it should also estimate the time that C++Debug is going to take to obtain the slices and the dices.

### 5.3.2 Additions

Instead of just menus, some windows should be supported so that a user can view the source code, the trajectory path, the program slice, etc. on the screen. Using a mouse can help a user probably better than using the keyboard in selecting which function to use, or selecting the variables and positions required to compute a slice.

### 5.3.3 Future Work

For a tested C++ program that has pointers, global variables, and static declarations in classes, the algorithm that was used to implement C++Debug yields an output slice larger than it should be (however, it still gives the correct output and its size is smaller than the original source program). Some lines that should be eliminated are not eliminated. If a better algorithm to manage pointers, global variables, and static declarations in classes is implemented, the size of the resulting slice will be smaller.

It will be desirable if C++Debug can be made a multi-user-tool. However, in the current implementation, since C++Debug saves specific files in a local directory, it cannot be used in the multi-user mode.

Because of the complexities of the C++ symbol table and the time constraint, the current version of C++Debug cannot treat array elements and fields in dynamic records as individual variables.

## REFERENCES

- [Bison 00] "Bison 1.35 Manual," [http://www.gnu.org/manual/bison-1.35/html\\_mono/bison.html.gz](http://www.gnu.org/manual/bison-1.35/html_mono/bison.html.gz), Last Update: March 2000, Last Access: April 30, 2003.
- [Cool 94] "CoolAid: The Cool Reference Manual," <http://www.cs.berkeley.edu/~aiken/ftp/cool-manual.ps>, Last Update: January 1994, Last Access: April 30, 2003.
- [Flex 01] "Flex, version 2.5 A Fast Scanner Generator Edition 2.5, March 1995," [http://www.gnu.org/manual/flex-2.5.4/html\\_mono/flex.html](http://www.gnu.org/manual/flex-2.5.4/html_mono/flex.html), Last Update: February 23, 2001, Last Access: April 30, 2003.
- [Gallagher 90] Keith Brian Gallagher, *Using Program Slicing in Software Maintenance*, Ph.D. Dissertation, Computer Science Department, University of Maryland, Baltimore County, MD, 1990.
- [Gallagher and Lyle 91] Keith B. 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.
- [Hedrick 75] G. E. Hedrick, "Program Documentation," *Journal of Data Education*, Vol. 15, No. 4, pp. 20-21, July 1975.
- [Korel 88] Bogdan Korel, "PELAS-Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253-1260, September 1988.
- [Korel and Laski 88] Bogdan Korel and Janusz Laski, "Dynamic Program Slicing," *Information Processing Letters*, Vol. 29, No. 3, pp. 155-163, October 1988.
- [Korel and Laski 90] Bogdan Korel and Janusz Laski, "Dynamic Slicing of Computer Programs," *Journal of Systems and Software*, Vol. 13, No. 3, pp. 187-195, November 1990.
- [Lyle 84] James R. Lyle, *Evaluating Variations on Program Slicing for Debugging*, Ph.D. Dissertation, Computer Science Department, University of Maryland, College Park, MD, 1984.

- [McDermid 93] John McDermid, *Software Engineer's Reference Book*, CRC Press, Inc., Boca Raton, Florida, 1993.
- [Nanja 90] 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.
- [Nanja and Samadzadeh 90] Sekaran Nanja and Mansur H. Samadzadeh, "A Slicing/Dicing-Based Debugger for C," *The 8<sup>th</sup> Annual Pacific Northeast Software Quality Conference*, Portland, OR, pp. 204-212, October 1990.
- [Pohl 94] Ira Pohl, *C++ for C Programmers*, 2<sup>nd</sup> Edition, The Benjamin/Cummings publishing Company, Inc., Redwood City, CA, 1994.
- [Samadzadeh and Wichaipanitch 93] 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.
- [Sommerville 01] Ian Sommerville, *Software Engineering*, 6<sup>th</sup> Edition, Addison Wesley Publishing Company, New York, N.Y., 2001.
- [Stroustrup 97] B. Stroustrup, *C++ Programming Language*, 3<sup>rd</sup> Edition, Addison-Wesley, Inc., Reading, Massachusetts, 1997.
- [Tassel 74] Dennie V. Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.
- [Weiser 81] Mark Weiser, "Program Slicing," *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, pp. 439-449, March 1981.
- [Weiser 82] Mark Weiser, "Programmers Use Slices When Debugging," *Communications of the ACM*, Vol. 25, No. 7, pp. 446-452, July 1982.
- [Weiser 84] Mark Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 352-357, July 1984.
- [Weiser and Lyle 86] 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.

## APPENDICES

## APPENDIX A

### GLOSSARY

- Action:** An action, e.g., an instruction  $X$  at position  $p$  in a trajectory  $T$ , sometimes represented as the pair  $(X,p)$ . For example,  $4^4$  and  $4^9$  in trajectory  $T$  in Figure 10 are actions that involve the same instruction 4. See also trajectory.
- Back( $T,q$ ):** Denotes the sublist  $\langle X_{q+1}, \dots, X_m \rangle$  of  $T$ , consisting of elements that follows  $T(q)$ , a trajectory at position  $q$ . Where  $T = \langle X_1, X_2, \dots, X_m \rangle$  denotes a trajectory of length  $m$ , and  $q$  is a position in  $T$ ,  $1 < q < m$ . See also trajectory, **Front( $T,q$ )** and **Del( $T,R$ )**.
- Bug:** An error in a computer program that may be either a syntax error or a logical error.
- Called action:** An action  $X^p$  is a called action if  $X$  is a called function instruction.
- Called-to-Calling:** Occurred when a slice is computed from a called action first and then a calling action.
- Calling action:** An action  $X^p$  is a calling action if  $X$  is a calling function instruction.
- Calling-to-Called:** Occurred when a slice is computed from a calling action first and then a called action.
- D( $X^p$ ):** The set of variables that are defined in action  $X^p$ . For example, in the execution trace of Figure 10,  $18^{15}$   $Avg := Sum/(MaxData + 1)$ ;  $Avg$  is a set of variables that are defined in  $18^{15}$ ,  $D(18^{15})$ . See also trajectory,  $U(X^p)$ ,  $DU(X^p)$ ,  $IR(X^p)$ , and  $TC(X^p)$ .
- Debugging:** A process to locate and correct errors or bugs. Debugging differs from testing in that testing is used to determine whether a program is working properly, whereas debugging localizes and corrects the errors.



**DEL(T,R):** A subtrajectory obtained from T by deleting from it all elements  $T(i)$  that satisfy R. Where  $T = \langle X_1, X_2, \dots, X_m \rangle$  denotes a trajectory of length m, and  $q$  is a position in T,  $1 < q < m$ . Where R is a predicate on the set of instructions in T. See also trajectory,  $Front(T,q)$ , and  $Back(T,R)$ .

**DF<sub>name</sub>(X<sup>p</sup>):** A set of variables that are defined in action X<sup>p</sup>; where  $X^p \in M(TF_{name})$ .

**Dicing:** The process of identifying a set of statements prone to contain an error.

**DU(X<sup>p</sup>):** Definition-Use Relation, a binary relation on  $M(T)$  in which one action assigns a value to an item of data and the other action uses that value. For example, in the execution trace of Figure 10,  $2^2$  assigns a value to variable Count and  $4^4$ ,  $9^5$ ,  $16^7$ , and  $17^8$  use that value. See also trajectory,  $M(T)$ ,  $D(X^p)$ ,  $U(X^p)$ ,  $IR(X^p)$ , and  $TC(X^p)$ .

**DUF<sub>name</sub>:** A Definition-Use-Function<sub>name</sub> Relation, is a binary relation on  $M(TF_{name})$ .

**DV(V,TF<sub>name</sub>):** A function that return a set of line numbers, where V is set of variables.

**Dynamic Slicing:** A slicing method defined on the basis of a computation rather than all computations. It generates a dynamic program slice by computing from the executable part of the original source program. See also program slicing and static slicing.

**EI:** A Called-to-Calling Relation between 2 functions,  $X^p \text{ IE } Y^t$ , iff both are in a Called-to-Calling situation, where  $X^p$  is a calling action and  $Y^t$  is a called action.

**F<sub>name</sub>:** A function, a set of instruction X's which is in the scope of influence of function name.

**Feasible Path:** Let a flowgraph of program P be a directed graph and  $C = (N, A, s, e)$  be a slicing criterion where:

1. N is a set of nodes,
2. A is a binary relation on N (a subset of  $N \times N$ ), referred to as a set of arcs,
3.  $s \in N$  is a unique entry node, and
4.  $e \in N$  is a unique exit node.

A node in N is referred to as an instruction, including a single instruction and a control instruction. A single instruction includes, for example, an assignment statement, an input or output statement, etc. A control instruction includes such statements as if-then-else or while statements, which are called test instructions. An  $arc(n,m) \in A$  corresponds to a

possible transfer of control from instruction  $n$  to instruction  $m$ . A path from entry node  $s$  to some node  $k$  when  $k \in N$  is called a sequence  $\langle n_1, n_2, \dots, n_q \rangle$  of instructions, such that  $n_1 = s$ ,  $n_q = k$  and  $(n_i, n_{i+1}) \in A$ , for all  $n_i$ ,  $1 < i < q$ . If there is input data, which causes the path to be traversed during program execution, the path is called feasible.

- FN( $q$ ): A function name such that  $X^q$ ,  $X$  is in the scope of influence.
- Front( $T, q$ ): The sublist  $\langle X_1, X_2, \dots, X_q \rangle$  of  $T$ , consisting of the first  $q$  elements of  $T$ , where  $T = \langle X_1, X_2, \dots, X_m \rangle$  denotes a trajectory of length  $m$ , and  $q$  is a position in  $T$ ,  $1 < q < m$ . See also trajectory, Back( $T, q$ ), and DEL( $T, R$ ).
- G( $X$ ): A set of variables and precedences that are declared as a global declaration.
- IE: A Calling-to-Called Relation between 2 functions,  $X^p$  IE  $Y^t$ , iff both are in a Calling-to-Called situation, where  $X^p$  is a called action and  $Y^t$  is a calling action.
- IR( $X^p$ ): Let  $X^p$  IR  $Y^t$ , iff  $X = Y$  is the Identity Relation IR on  $M(\text{Front}(T, q))$ . See also trajectory,  $M(T)$ ,  $D(X^p)$ ,  $U(X^p)$ ,  $DU(X^p)$ ,  $TC(X^p)$ , and  $\text{Front}(T, q)$ .
- IRF<sub>name</sub>: An Identity Relation in Function<sub>name</sub>,  $X^p$  IRF<sub>name</sub>  $Y^t$ , iff  $X = Y$  is the identity relation IRF<sub>name</sub> on  $M(\text{Front}(TF_{\text{name}}, q))$ .
- Last Definition: Last definition  $X^p$  of variable  $v$  at  $t$  is the action which has last assigned a value to  $v$  when  $t$  is reached on trajectory  $T$ . See also trajectory.
- LF<sub>name</sub>( $X^p$ ): A set of variables and C++ precedences that are declared as a local declaration in function name.
- $M(T)$ : A set of actions in a given trajectory  $T$ , where  $M(T) = \{ (X, p) : T(p) = X \}$ . See also trajectory.
- $M(TF_{\text{name}})$ : A set of actions in a given function of a given trajectory  $TF_{\text{name}}$ , where  $M(TF_{\text{name}}) = \{ X^p : \text{instruction } X \text{ at position } p \text{ in trajectory } TF_{\text{name}} \}$ .  $M(TF_{\text{name}})$  is a subset of  $M(T)$ .
- P: A set of instruction  $X$ 's, in a C++ tested program.
- Preprocessor: A separate first step in compilation, e.g., **#include**, **#define**, or **#if**.
- Program Slicing: A segment of a program that is separated and identified based on the premise that instead of localizing errors in the original program, which can be of large size, one can locate such errors in a program of smaller size which is sliced from the original program but still preserves part of the

original program's behavior for a particular input or relative to a particular variable.

**Slicing Criterion:** The specification that a behavior of interest of a program can be expressed as the values of some set of the variables at some set of statements.

**Static Slicing:** A method defined on the basis of all computations and used for generating a static program slices. The computations of static slices are done directly from the original source program. See also program slicing and dynamic slicing.

**T(p) :** The abstract list of a trajectory T whose elements are accessed at position p, e.g., for T in Figure 10,  $T(3) = 3$ ,  $T(5) = 9$ , etc. See also trajectory.

**TC( $X^p$ ):** Test-Control Relation, a binary relation on  $M(T)$ , capturing the effect between test actions and actions that have been chosen to execute by those test actions. For example, in the execution trace of Figure 10, the scope of the test action  $4^4$  influences the execution of  $9^5$ ,  $10^6$ ,  $16^7$ , and  $17^8$ , but it does not influence the execution of  $13^{10}$ ,  $14^{11}$ ,  $16^{12}$ , and  $17^{13}$ . See also trajectory,  $M(T)$ ,  $D(X^p)$ ,  $U(X^p)$ ,  $DU(X^p)$ , and  $IR(X^p)$ .

**TCF<sub>name</sub>:** A Test-Control-Function<sub>name</sub> Relation, is a binary relation on  $M(TF_{name})$ .

**Test Action:** An action  $X^p$  is a test action if X is a test instruction. See also trajectory.

**Test Instruction Statements:** A control instruction such as an **if-then-else** or a **while** statement.

**TF<sub>name</sub>:** A function trajectory, a feasible path of a function name that has actually been executed for some input.  $TF_{name}$  is a sublist of T.

**Trajectory:** A feasible path that has actually been executed for some input. For example,  $\langle 1,2,3,4,9,10,16,17,4,13,14,16,17,4,18,19 \rangle$  is the trajectory when the program in Figure 1 is executed on the input data  $MaxData = 2$ ,  $Data = (3,5)$ . A trajectory will be illustrated in terms of a pair (instruction, its position in the trajectory) rather than the instruction itself so as to distinguish between multiple occurrences of the same instruction in the trajectory. For example, instruction X at position p in T will be represented by the pair (X, p). For ease of understanding, the pair (X, p) will be replaced by  $X^p$  and will be referred to as an action. For example,  $4^4$  and  $4^9$  in trajectory T in Figure 10 are actions that involves the same instruction 4. See also feasible path.

- $U(X^p)$ : The set of variables that are used in  $X^p$ . For example, in the execution trace of Figure 10,  $18^{15}$   $Avg := Sum/(MaxData + 1)$ ;  $Sum$  and  $MaxData$  are a set of variables that are used in  $18^{15}$ ,  $U(18^{15})$ . See also trajectory,  $D(X^p)$ ,  $DU(X^p)$ ,  $IR(X^p)$ , and  $TC(X^p)$ .
- $UF_{name}(X^p)$ : A set of variables that are used in action  $X^p$ ; where  $X^p \in M(TF_{name})$ .
- $VDU(FunctionName)$ : A set of variables that are used,  $UF_{name}$ , and defined,  $DF_{name}$ , in the given function name.
- $VS(X_{DU})$ : A variable scope relation at  $X_{DU}$ , be a set of instructions  $X_{DCL}$ , where  $X_{DU}$  is in the scope of influence of  $X_{DCL}$ .
- $X$ : An instruction in a program and  $X \in IN^+$ . See also program  $P$ .
- $X_{DCL}$ : An instruction that declared variables such as “`int I;`”.
- $X_{DU}$ : An instruction that used or defined the variables that declared by  $X_{DCL}$ , where variables that used or defined are in the scope of influence of variables that declared in  $X_{DCL}$ .
- $X^p$ : An action, e.g., an instruction  $X$  at position  $p$  in a trajectory  $T$ , sometimes represented as the `pair(X,p)`. For example,  $4^4$  and  $4^9$  in trajectory  $T$  in Figure 10 are actions that involve the same instruction 4. See also trajectory.
- $Y^t$ : An action, instruction  $Y$  at position  $t$  in a trajectory  $T$ . See also  $X^p$ .

## APPENDIX B

### USER'S MANUAL FOR C++DEBUG

#### B.1 Introduction

C++Debug is a slicing and dicing based debugging tool for ANSI C++ that runs under the UNIX or Linux operating system. It has been designed in a way to provide ease and convenience to the user. Using C++Debug, the user can interact with the computer in locating errors in a program. For convenience of the user, the menu shown in Figure 56 allows the user to select any of the available functions.

#### B.2 C++Debug's Commands

At a UNIX prompt, C++Debug is invoked by typing the following command:

```
$C++Debug [prog_name]
```

where  $\$$  is a UNIX Bourne shell prompt and *prog\_name*, the optional parameter, is the name of the program to be loaded into the C++Debug environment. Once this command is executed, C++Debug will return to the help menu so that additional commands can be executed.

```

*****
*                C++Debug                *
*      A slicing and dicing              *
*      based debugging tool              *
*                version 1.31            *
*                                04/10/03 *
*****

S (lice)      produces slice(s)
D (ice)       produces dice(s)
V (iew)       display source program
T (rajectory) displays trajectory path
R (un)        a program slice to check output

L (evel)      select level of slice
                ( now set to level 1 )
E (ditor)     select editors 'VI' or 'EMACS'
                ( now set to use 'EMACS' )

!             invokes UNIX command interpreter
Q (uit)       quit from C++Debug environment

C++Debug>

```

Figure 56. Help menu and prompt

The following commands are available within the C++Debug environment. They are explained in the order that they appear in the C++Debug menu.

**S or Slice** Produces a slice of the program currently resident within the C++Debug environment with the variables supplied to it as its arguments.

Example:

```

C++Debug>S line_num var1 [var2...]
C++Debug>Slice line_num var1 [var2...]

```

where `line_num` is a valid line number in the program and `var1`, `var2` are variables in the program. This command requires a line number and at least one variable to produce a slice.

**D or Dice** Produces a dice, given a set of variables and a line number.

Example:

```

C++Debug>D line_num var1 [var2...] | var3 [var4...]
C++Debug>Dice line_num var1 [var2...] | var3 [var4...]

```

where *line\_num* is a valid line number in the program and *var1*, *var2*, *var3*, and *var4* are variables in the program. This command requires a line number and at least two variables to be supplied, separated by a |.

V or  
View Views or displays the source program resident within the C++Debug environment on the display unit. VI or EMACS is used to view the source program. One can select to uses VI or EMACS by using E or Editor.

T or  
Trajectory Displays the trajectory path or the execution trace of the source program resident within the C++Debug environment on the display unit.

R or  
Run Compiles and run the slice program currently resident within the C++Debug environment with the G++ compiler (to compare the slice output with the original program output).

L or  
Level In level 1:  
  
C++Debug can work with any size source program. C++Debug will use some harddisk space (about 1 K-byte) to keep track of information. Level 1 yields an output slice larger than level 2.

In level 2:

Level 2 allows the user to get a program slice smaller than Level 1. The limitation is that C++Debug will use more of your space than Level 1 in order to keep track of the trajectory path.

E or  
Editor Selects editor from VI, EMACS, or EMACS for windows.

!  
Invokes a UNIX command.  
Example:  
*C++Debug>!ls -l*  
*C++Debug>!who*

Q or  
Quit Exits from the C++Debug environment to the UNIX system.

## B.3 Tutorial

In order to make it easier to understand, this tutorial walks the user through C++Debug step by step. The tutorial guide you from the basic commands, such as view manual or version, to the more complicated commands such as Slice or Dice.

### B.3.1 A Step-by-Step Guide

1. At the UNIX prompt, type C++Debug. You will see the following message:

```

$ C++Debug

usage: C++Debug [-mvg] [-lnnnn] [file]

-m Display manual           Ex. $C++Debug -m
-v Display version         Ex. $C++Debug -v
-g Display grammar         Ex. $C++Debug -g filename.cpp
-t Display trajectory      Ex. $C++Debug -t filename.cpp
-l Check nnnn consecutively executed statements for looping
  by default nnnn = 10    Ex. $C++Debug -l40 filename.cpp

```

2. Type

```
$ C++Debug -m
```

C++Debug's user's manual should be display on the screen.

3. Type

```
$ C++Debug -v
```

C++Debug's version should be display on the screen.

4. Type

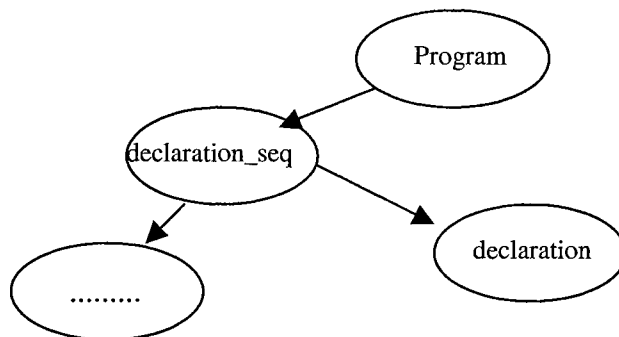
```
$ C++Debug -g Test1.cpp
```

For a grammar, you will see the tree corresponding to the derivation of program Test1.cpp (see Subsection B.3.2.1). The tree is up side down (the root is at the bottom).



Display on the screen	Should look like
<pre> ..... compound_statement ..... declaration: function_definition  declaration_seq: declaration_seq declaration  Program: declaration_seq </pre>	<pre> Program: declaration_seq  declaration_seq: declaration_seq declaration  declaration: function_definition  .... compound_statement .... </pre>

Or, it can be represented graphically as follow:



This area is left for someone who is interested and would like to translate a derivation tree from the text mode to graphical representation (a directed graph).

## 5. Type

```
$ C++Debug -t Test1.cpp
```

The trajectory path that allows you to follow the execution of a C++ program, statement by statement, will show on the screen as follows:

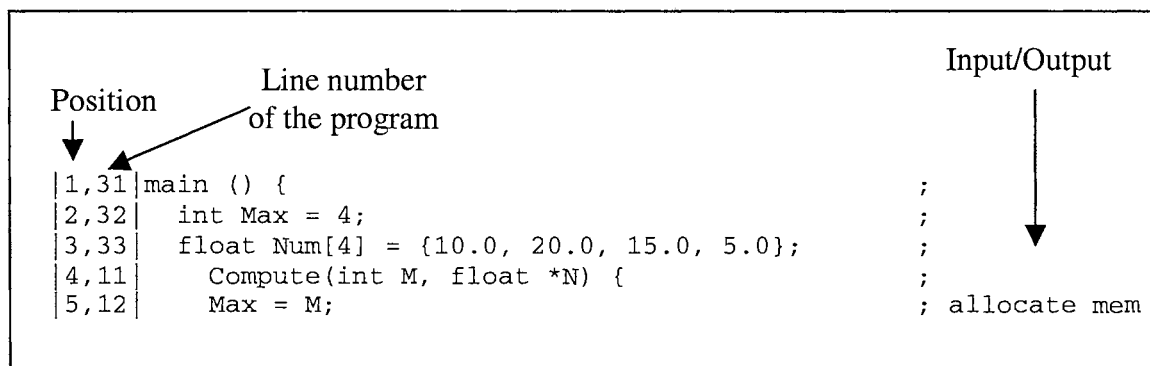


Figure 57. The trajectory path

where 1, 2, 3, ... are positions and 31, 32, 33, 11, 12, ... are line numbers of program `Test1.cpp`. The input and/or output of the program will be shown after ';'.  
 Now you can keep track of your program as to what line numbers are running.

6. Let's try program `Test2.cpp` (see Subsection B.3.2.2). with multiple loops, type

```
$ C++Debug -l10 Test2.cpp
```

Be careful, -l10 is letter ell and one zero.

Compare it with:

```
$ C++Debug -l100 Test2.cpp
```

Now you have some idea about `C++Debug` and how it works as a `cpptest` tool, and what a trajectory is. However, locating errors in a trajectory still requires work. `C++Debug` can help users to find errors. Please follow the examples to gain facility in using the tool.

Next, let's try program slicing.

7. Type

```
$ C++Debug Test1.cpp
```

The screen should display the menu shown in Figure 56.

## 8. Type

```
C++Debug> s 21 Tsum
```

That means you are going to slice a program based on variable `Tsum` at line 21. The new program (which is smaller than the original) will be shown in the editor mode (by selecting between `VI` and `EMACS`).

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.

Now, let's try another line number:

```
C++Debug> s 32 Max
```

That means you are going to slice a program based on variable `Max` at line 32. The new program (which is smaller than the original) will be shown.

Now, let's try another line number:

```
C++Debug> s 10 public
```

You will get an empty slice, because that line has not been executed.

**Tip:** To select 'Line' and 'Variable', one must be careful. One must make sure that the line will be executed.

For example,

```
10:  if( a > 100)
11:    cout<<"greater than 100"<<endl;
12:  else
13:    cout<<"less than or equal 100"<<endl;
```

Assume that `a = 150`.

In this case, suppose that you select line 13, you will get an empty slice, because line 13 has not been executed.

Let's slice program `Test1.cpp` based on variable `Sum` and `Avg` by first typing

```
C++Debug> s 35 Sum
```

you should get a slice program based on variable `Sum` at line 35, and by typing

```
C++Debug> s 36 Avg
```

you should get a slice program based on variable `Avg` at line 36.

## 9. Dice

Now you know to slice a program.

Now, let's try 'Dice'.

Dicing technique is 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.

In the previous examples, `Avg` gives an incorrect output. The correct output of `Avg` must be

$$(10.0 + 20.0 + 15.0 + 5.0) / 4 = 12.5$$

However, `Sum` gives the correct output of 50. Therefore you can locate the error in the program by using the dicing technique.

```
C++Debug> d 36 Avg 35 Sum
```

where `36 Avg` is a slice at line 36 based on variable `Avg` (which gave an incorrect result) and `35 Sum` is a slice at line 35 based on variable `Sum` (which gave a correct result).

You should get the following output:

```

25:    float Avg(void) {
26:        return Sum()/(Max + 1);
36:    cout<<A.Avg()<<endl;

```

You know that line 25 and line 36 are correct (obviously). So we have line 26 left.

We find that line 26 should be

```

26: return Sum()/(Max);    // which is correct

```

instead of

```

26: return Sum()/(Max + 1);

```

There exists an extra '+ 1', which is incorrect.

## B.3.2 Source Code Listing

### B.3.2.1 Test1.cpp

```

/*****
 *
 *   A program for calculating the sum
 *   and average of a set of numbers.
 *
 *****/
#include <iostream>
using namespace std;

class Compute {

private:
    int Max;
    float Num[4];

public:
    Compute(int M, float *N) {
        Max = M;
        cout<<"allocate mem"<<endl;
        for(int I=0; I<Max; ++I)
            Num[I] = N[I];
    }

    float Sum(void) {
        float Tsum = 0;
        for(int I=0; I<Max; ++I)
            Tsum = Tsum + Num[I];
        return Tsum;
    }

    float Avg(void) {
        return Sum()/(Max + 1);
    }

};

main () {
    int Max = 4;
    float Num[4] = {10.0, 20.0, 15.0, 5.0};
    Compute A(Max, Num);
    cout<<A.Sum()<<endl;

```

```
    cout<<A.Avg()<<endl;
}
```

### B.3.2.2 Test2.cpp

```
/*
 *
 *   A program for loops testing
 *
 */
#include <iostream>
using namespace std;

int main (void) {

    for(int i = 1; i<=100; i++) {
        for(int j = 1; j<=100; j++) {
            }
        }
    }
}
```

## APPENDIX C

### DATASTRUCTURE DESIGN FOR C++DEBUG BASED ON DYNAMIC PROGRAMSLICING AND DICING

#### C.1 Introduction

In order to design datastructures for C++Debug, the grammar that appears in *C++ Programming Language Third Edition* written by Bjarne Stroustrup, the creator of C++, has been used in this design. The design was started from basic functions and expanded to pointers, structures, functions, and classes.

#### C.2 Types and Declarations

##### C.2.1 Types

Every name (identifier) in a C++ program has a type associated with it. The type determines what operations can be applied to the name and how such operations are interpreted. In this design, the data structure of `Types` used to store all built-in and user-defined types used in the program is shown in Figure 58.

`InsertTypes(ID)` is used to manage database in Figure 60, which includes all of these six functions:

`InsertBasicType(ID)` is used to insert basic types into the list such as **bool**, **char**, **int**, **double**, **void**, **unsigned**, **long**, **short**, etc.

`InsertClassName(ID)` is used to insert class names into the list.

```

enum Groups = { Basic, Structure, Class, Template, TypeDef };
struct Types {
    char Type[TYPELENGTH];
    char Group[Groups];
}

```

Figure 58. Data structure of Types

```

1  #include <assert.h>
2  #include <stdlib.h>
3  #include <iostream.h>

4  typedef int Item;

5  class ItemArray {
6      friend class ItemIterator;
7      int size;
8      Item* array;
9      public:
10     ItemArray(int elms) : size(elms)
11     {
12         assert(elms>0);
13         array = new Item[elms];
14         assert(array!=NULL);
15     }
16     ~ItemArray() {
17         delete []array;
18         size=0;
19         array=NULL;
20     };
21 };
22 class ItemIterator {
23     int index;
24     ItemArray* obj;
25     public: ItemIterator(ItemArray& i):obj(&i),
26         index(0) {} ;
27     Item* operator() () {
28         if (index < obj->size)
29             return &obj->array[index++];
30         else
31             return NULL;
32     };
33 main() {
34     ItemArray a(100);
35     ItemIterator p(a);
36     Item* ptr;
37     Item i(0);
38     while ((ptr=p()) != NULL)
39         *ptr=i++;
40     // The same without ptr
41     ItemIterator check(a), use (a);
42     while (check() != NULL)
43         cout << *use() << '\n';
44     return 0;
45 }

```

Figure 59. A C++ program that uses iterators



InsertStructureName(ID) is used to insert structure names into the list.  
 InsertEnumName(ID) is used to insert enumeration names into the list.  
 InsertTemplateName(ID) is used to insert template names into the list.  
 InsertTypeDefName(ID) is used to insert type definition names into the list.

For example, all types of the program in Figure 59 can be stored in the database as shown in Figure 61.

Type	Group
<b>bool</b>	Basic
<b>char</b>	Basic
<b>unsigned</b>	Basic
<b>long</b>	Basic
<b>short</b>	Basic
<b>int</b>	Basic
<b>float</b>	Basic
<b>double</b>	Basic
<b>void</b>	Basic
List of Enum Names	Enum
List of Structure Names	Structure
List of Class Names	Class
List of Template Names	Template
List of TypeDef Names	TypeDef

Figure 60. Show the database of Types used in C++

Type	Group
<b>int</b>	Basic
Item	TypeDef
ItemArray	Class
ItemIterator	Class

Figure 61. Show how the database stores Types of the program in Figure 59

## C.2.2 Declarations

Before a name (identifier) can be used in a C++ program, it must be declared. That is, its type must be specified to inform the compiler to what kind of entity the name refers. The data structure of Declarations is used to store all variables declared in the program with their characteristic and scopes. It is designed as shown in Figure 62. For example, for the program in Figure 59, all variables can be stored in the database as shown in Figure 63.

```

struct Declarations {
    char VariableName    [VARIABLELENGTH];
    char Type[TYPELENGTH];
    bool Array;
    bool Pointer;
    bool Reference;
    bool Const;
    bool Function;
    bool Argument;
    int  ScopeStart;
    int  ScopeEnd;
}

```

Figure 62. Data structure of Declarations

Variable Name	Type	Array	Pointer	Reference	Const	Function	Argument	Scope Start	Scope End
Size	int	F	F	F	F	F	F	7	21
Array	Item	F	T	F	F	F	F	8	21
Elms	int	F	F	F	F	F	T	10	15
Index	int	F	F	F	F	F	F	23	32
Obj	ItemArray	F	T	F	F	F	F	24	32
I	ItemArray	F	F	T	F	F	T	25	25
Operator	Item	F	T	F	F	T	F	1	45
A	ItemArray	F	F	F	F	F	F	34	45
P	ItemArray	F	F	F	F	F	F	35	45
Ptr	Item	F	F	F	F	F	F	36	45
I	Item	F	F	F	F	F	F	37	45

Figure 63. Show how the database stores Declarations of the program in Figure 59

Where the elements in the field named `Type` is one of the elements contained in the field named `Type` in Figure 58. Field named `Array`, `Pointer`, `Reference`, `Const`, `Function`, and `Argument` are used to specify the kinds of each variable in the field of `VariableName`. Function `InsertDeclaration(ID)` is used to insert the variables into the database as shown in Figure 63. Function `InsertTypes(ID)` in Section C.2.1 is called to store all built-in and user-defined types also.

Function `ScopeStart(ID)` and `ScopeEnd(ID)` are used to determined the scope of each variable name. See Section C.2.3 for more information.

### C.2.3 Scope

A declaration introduces a name into a scope; that is, a name can be used only in a specific part of the program text. Figure 64 shows the example of the scopes. Function `ScopeStart(ID)` and `ScopeEnd(ID)` are used to determined the scope of each identifier and store it in the database as shown in Figure 63.

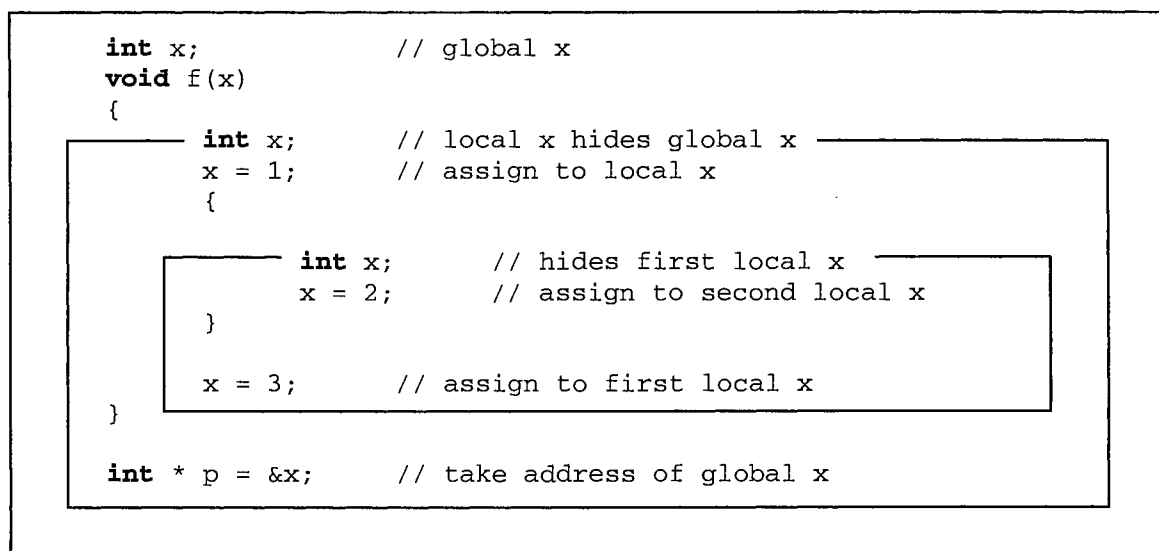


Figure 64. Scopes of variable `x` as a global, local, and second local

### C.2.4 Typedef

A declaration prefixed by the keyword **typedef** declares a new name for the type rather than a new variable of the given type. The data structure of `Typedef` is used to store the real types of the types defined by **typedef** statement. Its data structure is shown in Figure 65. The field named `Type` in Figure 65 is related to the field named `Type` in Figure 58. For example, the part of the program in Figure 66 can be stored in the database as shown in Figure 67.

```

struct Typedef {
    char Type[TYPELENGTH];
    char RealType[TYPELENGTH];
}

```

Figure 65. Data structure of `Typedef`

```

typedef    short    int16;
typedef    int      int32;
typedef    long     int64;
typedef    char *   Pchar;
Pchar      P1, P2;
char      P3 = P1;

```

Figure 66. A program segment that uses **typedef**

Type	Real Type
Pchar	<b>char*</b>
int16	<b>short</b>
int32	<b>int</b>
int64	<b>long</b>

Figure 67. Show how the database stores `Typedef` defined by **typedef** in Figure 66

### C.3 Pointers, Arrays, Constant, References, and Structures

#### C.3.1 Pointers

For a type  $T$ ,  $T^*$  is the type pointer to  $T$ . That is, a variable of type  $T^*$  can hold the address of an object of type  $T$ . The data structure named `Pointers` is used to store the variables pointed to. The field named `VariableName` in Figure 68 is related to the field `VariableName` in Figure 62. The function named `InsertPointerName(ID)` is used to manage Figure 70. For example, the part of the program in Figure 69 can be stored in the database as shown in Figure 70.

```

struct Pointers {
    char VariableName[VARIABLELENGTH];
    char PointTo[VARIABLELENGTH];
    int NoOfStars;
}

```

Figure 68. Data structure of Pointers

```

1 int * pi;           // point to int
2 char ** ppc;       // point to pointer to char
3 int * ap[15];      // array of 15 pointers to ints
4 int (*fp) (char*); // pointer to function taking a
                       // char * argument; return an int
5 int * f (char *);
6 char c = 'a';
7 char *p = &c;      // p holds the address of c
8 char c2 = *p;      // c2 == 'a'

```

Figure 69. A program segment that uses pointers

Note: We do not care prefixed ‘\*’ and ‘&’ of variables `p` and `c`, because their characteristic was stored in Figure 70.

VariableName	PointTo	NoOfStars
pi	NULL	1
ppc	NULL	2
ap	NULL	1
fp	NULL	1
f	NULL	1
p	&c	1

Figure 70. Show how the database stores Pointers of the part of the program in Figure 69

Line Number	Def(n)	Ref(n)
6	c	-
7	p	c
8	c2	p

Figure 71. Show how the database stores Def(n) and Ref(n) of the part of the program in Figure 69

### C.3.2 Arrays

For a type T, T[Size] is the type array of size elements of type T. The elements are indexed from 0 to size-1. The data structure named Arrays is used to store all array variables' dimensions. The field named VariableName in Figure 72 is related to the field named VariableName in Figure 62. The function InsertArrayName(ID) is used to manage Figure 74.

```

Struct Arrays {
    char VariableName[VARIABLELENGTH];
    char Dimension[DIMENSIONLENGTH];
}

```

Figure 72. Data structure of Arrays

```

float    v[3];        // 1 Dimension and size = 3
char     a[32][10];   // 2 dimension and size = 32x10

```

Figure 73. A program segment that uses arrays

VariableName	Dimension
v	[3]
a	[32][10]

Figure 74. Show the data base of Arrays used by the part of the program in Figure 73

### C.3.3 Pointers into Arrays

In C++, pointers and arrays are closely related. The name of array can be used as a pointer to its initial element. Therefore, all variables of these types will be managed by the mixing of Pointers in Section C.3.1 and Arrays in Section C.3.2.

```

1  int v[] = {1,2,3,4};
2  int * p1 = v;           // pointer to initial element
                             // (implicit conversion)
3  int * p2 = &v[0];      // pointer to initial element
4  int * p3 = &v[4];      // pointer to one beyond last element

```

Figure 75. A program segment that uses pointers into arrays

VariableName	PointTo	NoOfStars
p1	v	1
p2	&v[0]	1
p3	&v[4]	1

Figure 76. Show how the database uses function InsertPointerName(ID) in Section C.3.1 to store variables of the part of the program in Figure 75

VariableName	Dimension
v	[4]

Figure 77. Show how the database uses function `InsertArrayName(ID)` in Section C.3.2 to store variables of the part of the program in Figure 75

Line Number	Def(n)	Ref(n)
1	v	-
2	p1	v
3	p2	v[0]
4	p3	v[4]

Figure 78. Show how the database stores `Def(n)` and `Ref(n)` of the part of the program in Figure 75

#### C.3.4 Constant

C++ offers the concept of a user-defined constant, a constant, to express the notation that a value doesn't change directly. The datastructure named `Declarations` in Section C.2.2 is used to manage by using the function named `InsertDeclaration(ID)`, while the data structure named `Pointers` in Section C.3.1 used to manage their pointers.

```

1  const int model = 90;
2  const int x;
3  const char * pc = 9;
4  char *strcpy(char *p, const char *q) // cannot modify *q
5  int v[] = {1,2,3,4};
6  const int c3 = my_f93);

```

Figure 79. A program segment that uses `const`



Variable Name	Type	Array	Pointer	Reference	Const	Function	Argument	Scope Start	Scope End
model	<b>int</b>	F	F	F	T	F	F	1	6
x	<b>int</b>	F	F	F	T	F	F	2	6
pc	<b>char</b>	F	T	F	T	F	F	3	6
strcpy	<b>char</b>	F	T	F	T	T	F	4	6
v	<b>int</b>	T	F	F	T	F	F	5	6
c3	<b>int</b>	F	F	F	T	F	F	6	6
p	<b>char</b>	F	T	F	F	F	T	4	6
q	<b>char</b>	F	T	F	T	F	T	4	6

Figure 80. Show how the database stores constant declared in Figure 79

For example, from the part of the program in Figure 79, all variables can be stored in the data base as shown in Figure 80, all pointers are stored in the database as shown in Figure 80, and their Def(n) and Ref(n) can be determined as shown in Figure 82.

VariableName	PointTo	NoOfStars
pc	9	1
strcpy	NULL	1

Figure 81. Show how the database uses function InsertPointerName(ID) in section C.3.1 to variables of the part of the program in Figure 79

Line Number	Def(n)	Ref(n)
1	model	-
2	pc	-
3	c3	my_f

Figure 82. Show how the database stores Def(n) and Ref(n) of the part of the program in Figure 79

### C.3.5 References

A reference is an alternative name for an object. The main use of references is for specifying arguments and returns values for functions in general and for overloaded operators in particular. The notation X& means reference to X. In this design we will see a reference variable as copied variable as shown in Figure 85 which is managed by the function named `InsertReferenceName(ID)`.

```

structure References {
    char VariableName [VARIABLELENGTH];
    char ReferenceTo [VARIABLELENGTH];
}

```

Figure 83. Data structure of References

```

1 int i = 1;
2 int &r = i;           // x and i now refer to the same int
3 int x = r;           // x = 1;
4     r = 2;           // i = 2;

```

Figure 84. A program segment that uses references

VariableName	ReferenceTo
r	i

Figure 85. Show how the database uses function `InsertReferenceName(ID)` to store variables of the part of the program in Figure 84

### C.3.6 Pointer to Void

A pointer of any type of object can be assigned to a variable of type `void*`, a `void*` can be assigned to another `void*`, `void*`s can be compared for equality and

inequality, and a **void\*** can be explicitly converted to another type. In this design, functions named `InsertDeclarationName(ID)` in Section C.3.2 and `InsertPointerName(ID)` in Section C.3.1 are used to manage their variables.

```

void f(int* pi)
{
    void* pv = pi;
    *pv;
    pv++;
    int* pi2 = static_cast<int*>(pv);
    double* pd1 = pv;
    double* pd2 = pi;
}

```

Figure 86. A program segment that uses pointer to **void**

### C.3.7 Structures

An array is an aggregate of elements of the same type. A **struct** is an aggregate of elements of (nearly) arbitrary types. In this design, the data structure of Structures used to store all structures defined in the program is shown in Figure 88. Functions `InsertStructureName(ID)` and `InsertDeclaration(ID)` in section C.2.2 are used to manage as shown in Figure 90.

VariableName	PointTo	NoOfStars
pi	NULL	1
pv	pi	1
pi2	static_cast	1
pd1	pv	1
pd2	pi	1

Figure 87. Show how the database uses function `InsertPointerName(ID)` in section C.3.1 to store variables of the part of the program in Figure 86

```

struct Structures {
    char StructureName [VARIABLELENGTH];
    Declarations * Elements [MAXELEMENTS]; // See Figure 62
}

```

Figure 88. Data structure of Structures

```

struct address {
    char *    name;
    long int  number;
    char *    street;
    char *    town;
    char      state[2];
    int      zip;
};

struct List;    // to be defined later
struct Link {
    Link * pre;
    Link * suc;
    Link * member_of;
};

struct      List {
    Link * head;
};

```

Figure 89. A program segment that uses structures

#### C.4 Operators

The function named `Def(n)` is used to determine the set of variables whose values may be defined at line number `n`, while the function named `Ref(n)` is used to determine the set of variables whose values may be referenced at line number `n`, as shown in Figure 91.

Structure Name	Elements
address	
Link	
List	

Variable Name	Type	Array	Pointer	Reference	Const	Function	Argument	Scope Start	Scope End
name	<b>char</b>	F	T	F	F	F	F	-	-
number	<b>long</b>	F	F	F	F	F	F	-	-
street	<b>char</b>	F	T	F	F	F	F	-	-
town	<b>char</b>	F	T	F	F	F	F	-	-
state	<b>char</b>	T	F	F	F	F	F	-	-
zip	<b>int</b>	F	F	F	F	F	F	-	-

Variable Name	Type	Array	Pointer	Reference	Const	Function	Argument	Scope Start	Scope End
pre	Link	F	T	F	F	F	F	-	-
suc	Link	F	T	F	F	F	F	-	-
member_of	Link	F	T	F	F	F	F	-	-

Variable Name	Type	Array	Pointer	Reference	Const	Function	Argument	Scope Start	Scope End
head	Link	F	T	F	F	F	F	-	-

Figure 90. Show how the database stores Structures of the part of the program in Figure 89

Description	Grammar	Statement	Def(n)	Ref(n)
post increment	lvalue ++	I++	I	I
post decrement	lvalue --	d--	d	d
size of object	sizeof (expr)	a = sizeof(int)	a	sizeof
pre increment	++ lvalue	++I	I	I
pre decrement	-- value	--d	d	d
complement	~ expr	t = ~e	t	e
not	! expr	t = !e	t	e
unary minus	- expr	t = -e	t	e
unary plus	+ expr	t = +e	t	e
address of	& lvalue	&t = e	t	e
dereference	* expr	*t = e	t	e
create(allocate)	new [type]	t = new [Type]	t	e, Type
multiply	expr * expr	t = e1 * e2	t	e1, e2
divide	expr / expr	t = e1 / e2	t	e1, e2
modulo (remainder)	expr % expr	t = e1 % e2	t	e1, e2
add (plus)	expr + expr	t = e1 + e2	t	e1, e2
subtract (minus)	expr - expr	t = e1 - e2	t	e1, e2
shift left	expr << expr	t = e1 << e2	t, e1	e2
shift right	expr >> expr	t = e1 >> e2	t, e1	e2
less than	expr < expr	b = e1 < e2	b	e1, e2
less than or equal	expr <= expr	b = e1 <= e2	b	e1, e2
greater than	expr > expr	b = e1 > e2	b	e1, e2
greater than or equal	expr >= expr	b = e1 >= e2	b	e1, e2
equal	expr == expr	b = e1 == e2	b	e1, e2
not equal	expr != expr	b = e1 != e2	b	e1, e2
bitwise AND	expr & expr	b = e1 & e2	b	e1, e2
bitwise exclusive OR	expr ^ expr	b = e1 ^ e2	b	e1, e2
bitwise inclusive OR	expr   expr	b = e1   e2	b	e1, e2
logical AND	expr && expr	b = b1 && b2	b	b1, b2
logical inclusive OR	expr    expr	b = b1    b2	b	b1, b2
simple assignment	lvalue = expr	t = e	t	e
multiply and assign	lvalue *= expr	t *= e	t	t, e
divide and assign	lvalue /= expr	t /= e	t	t, e
modulo and assign	lvalue %= expr	t %= e	t	t, e
add and assign	lvalue += expr	t += e	t	t, e
subtract and assign	lvalue -= expr	t -= e	t	t, e
shift left and assign	lvalue <<= expr	t <<= e	t	t, e
shift right and assign	lvalue >>= expr	t >>= e	t	t, e
AND and assign	lvalue &= expr	t &= e	t	t, e
inclusive OR and assign	lvalue  = expr	t  = e	t	t, e
exclusive OR and assign	lvalue ^= expr	t ^= e	t	t, e

Figure 91. Show how to determine the set of variables by using functions Def(n) and Ref(n)

## APPENDIX D

### SOFTWARE SPECIFICATION

#### D.1 Introduction

The main purpose of software specification is used to define the functionality of C++Debug and constraints on its operation, plan the system development process, develop validation tests for the system, and help understand the system and the relationships between its parts.

#### D.2 General Description

C++Debug is an interactive debugging tool designed to function as a utility program of the UNIX system. C++Debug is developed based on slicing and dicing techniques. In order for C++Debug to be more powerful, dynamic slicing rather than static slicing is chosen for implementation. C++Debug was designed in a way to allow ease and convenience on part of the user. Using C++Debug, the user can interact directly with the computer in locating errors in a program. Menus are provided to allow the user to select any one of a number of functions (Slice, Dice, Help, etc.) supported by C++Debug.

## D.3 Specific Requirements

Functional and non-functional requirements are introduced in this part. Functional requirements provide how the system react to particular inputs, behave in particular situations, and explicitly state what the system should not do [Sommerville 01]. Non-functional requirements are about constraints such as timing constraints, constraints on the development process, standards, etc.

### D.3.1 Functional Requirements

Function requirements describe services provided for the user by using natural language with cross-references to requirement specifications [Sommerville 94].

#### 4.2.0 General

##### 4.2.0.1 Name

C++Debug

Rational:

C++Debug is a slicing and dicing based debugging tool for C++.

##### 4.2.0.2 Purpose

This project develops an interactive debugging tool, called C++Debug, for debugging a C++ language program.

Rational:

C++Debug is designed to function as a utility program of the UNIX system and is developed based on slicing and dicing techniques.

##### 4.2.0.3 Hardware and Software

C++Debug runs under UNIX machine.

Rational:

The SUN machine locates on the second floor of the Computer Science Building.

#### 4.2.1. Program Slicing

##### 4.2.1.1 ANSI C++



Can be used with every command and every instruction of ANSI C++ based on UNIX environment.

Rational:

C++Debug has to generate a program slice for every user program that uses ANSI C++ based on UNIX environment to implement the program.

#### 4.2.1.2 Automatic

Program slices can be found automatically by a method used to decompose programs through analyzing their data flow and control flow.

Rational:

C++Debug automatically generates program slices.

#### 4.2.1.3 Eighty percent of A program slice must be smaller size than that of the original program.

Rational:

This is because there always at least one slice, that is, program itself. As a consequence, when slicing at a variable of interest, the size of the resulting program slice is generally smaller than that of the original program.

#### 4.2.1.4 Property Consistency

Program slice can be executed independently of one another.

Rational:

The smaller size of the program slice is a C++ program that still maintains all aspects of the original program behavior with respect to the criterion variable.

#### 4.2.1.5 Produces Exactly one projection

Each slice produces exactly one projection of the original program's behavior.

Rational:

The smaller size of the program slice must still maintain all aspects of the original program behavior with respect to the criterion variable.

#### 4.2.1.6 Reduction

The slice must have been obtained from the original program by statement deletion.

Rational:

The idea of a program slicing is to focus on the statements that have something to do with a variable of interest (criterion variable), with those statements that are unrelated being omitted.

### 4.2.3. Dynamic Slicing

#### 4.2.3.1 Computation

C++Debug generates a dynamic program slice by computing from the trajectory of the original source program.

Rational:

Contradict with static slicing, which generates a static program slice directly from the original source program.

#### 4.2.3.2 Arrays and Fields

C++Debug treats array elements and fields in dynamic records as individual variables.

Rational:

Dynamic slicing characteristics [Korel 90].

#### 4.2.3.3 Size Comparing with Static Slicing

Dynamic slicing yields a program slice of generally smaller size than that of static slicing, or in the worst case, of equal size to that of static slicing.

Rational:

The runtime handling of arrays and pointer variables helps to reduce the size of the slice.

4.2.3.4 # lines of executable path of the original source program at least 5,000 lines of the executable path of the original codes can be computed by C++Debug.

Rational:

To make sure that it can work with for any small run-time programs, medium run-time programs and any run-time modules.

### 4.2.4. Dicing

#### 4.2.4.1 Computation of a Variable

If the computation of a variable,  $V$ , depends on the computation of another variable,  $W$ , then when ever  $W$  has an incorrect value, so does  $V$ .

Rational:

From dicing characteristics [Lyle 84].

4.2.4.2 Using the dicing technique, C++Debug can then be used to compare two or more slices resulting from the program slicing technique to identify the set of statements that are likely to contain an error.

Rational:

From dicing characteristics [Lyle 84].

### 4.2.5 Time Complexity

#### 4.2.5.1 Dynamic Slicing

Time to compute the program slice is less than 2 minutes at 5,000 lines of the executable path of the original codes.

Rational:

Protected from infinite loops.

#### 4.2.5.2 Static Slicing

Time to compute the program slice is less than 2 minutes at 5,000 lines of the original codes.

Rational:

Protected from infinite loops.

#### 4.2.5.3 Dicing

For two variables and 1000 lines of program slices the time to find the error line is less than 30 seconds.

Rational:

Users cannot wait for a long time.

### 4.2.6 Space Complexity

#### 4.2.6.1 Size of the machine code.

After compiled, the total size of machine code is not more than 100 k-byte.

Rational:

Comparing with the other debugger e.g. SDB, DBX etc.

#### 4.2.6.2 Memory Space

While executing, C++Debug can use the total memory in the system e.g. stack, heap, code etc., not more than 1 m-byte.

Rational:

If C++Debug uses a small primary and secondary storage, it can be used on a small machine.

### 4.2.7 Single-user and Multi-user

C++Debug can be used for both single-user and multi-user modes.

Rational:

C++Debug has been designed to have no critical section, shared memory and shared process, but it was designed to run independently like a UNIX utility command.

### 4.2.8 GUI

GUI's menus are provided by C++Debug to allow the users to select any one of the functions of slice, dice, help, etc.

Rational:

To make system user-friendly.

### 4.2.9 Software Metrics

Following by Cyclomatic complexity theory

## D.3.2 Non-function Requirements

### 5.2.1 User Interface

Although GUI's menus are provided by C++Debug to allow the user to select any one of the functions i.e. slice, dice, help, etc., for other function we cannot specify one. However, C++Debug must be designed to user-friendly.

### 5.2.2 System Cost

In order to compete with other debugger tools in the market, the price of C++Debug at full functions should not more than 49\$. So the total cost of C++Debug project should not more than 10,000 dollars.

### 5.2.3 Software Size

Although a large program can be run in UNIX environment, however, the size of C++Debug should not be more than 100 k-byte. The reason is that average size of other debuggers e.g. DBX, SDB are not more than 100k bytes.

### 5.2.4 Reliability

After delivering C++Debug to the customer, the number of errors must exist not more than 3 times a month. And the existing errors must be corrected in 1 week since it has been found.

## APPENDIX E

### TESTING AND EVALUATION

#### E.1 Introduction

C++Debug was designed to function as a utility program of the UNIX system and was developed based on slicing and dicing techniques. After C++Debug was implemented, testing was conducted to ensure that each unit met its specification (unit testing), and to ensure that the software requirements had been met integration and system testing was done. Testing is the primary means for showing that the implementation has the requisite functionality and satisfies other non-functional properties [McDermid 93]. Testing and other forms of verification and validation are important at all stages of the software development process. In order to know how C++Debug can be used to enhance the debugging process, evaluation was introduced.

#### E.2 Testing

##### E.2.1 Black and White Testing

Black and white testing was used to test C++Debug. In black-box testing, the internal structure and behavior of a system is not considered when the test data is selected

[McDermid 93]. Acceptance testing is the testing of a software system to ensure that it meets user requirements (see Appendix D). At this stage the test data is chosen by a careful reading of the requirements specification. In white-box testing, (e.g., unit testing), the internal structure and behavior of a system is considered when the test data is selected. Here a program unit (subroutine, procedure) of C++Debug was exercised with data, with the aim of ensuring that the code inside the unit implemented its specification. In this form of testing, a major aim is to ensure that a certain proportion of the software structure are exercised, a typical target being the execution of about 85% of the branches and 100% of all the statement in a program unit [McDermid 93]. The test data sets have to be developed to maximize the proportion of structural elements being exercised. To do this, the internal structure of a unit has to be examined.

## E.2.2 Testing and The Software Life Cycle

During the various phases of the software life cycle, C++Debug was tested as follows.

### E.2.2.1 Testing and Requirements Analysis

The major development activities that take place during this phase are the elicitation and clarification of requirements and the subsequent construction of the system specification [McDermid 93]. The major testing activity that occurs during this phase is the derivation of the verification requirements. The C++Debug requirements are listed in Appendix D. During the latter stages of C++Debug development, their requirements were converted into system validation tests and acceptance tests. Their tests determine whether a system meets its requirements. For example,

- 7.8 When the slice-criterion command is typed with a correct variable name and with a valid trajectory number, the program slice will be generated and stored in the file named "\_cpptrace\_slice.dat".

This leads to a number of tests as follows:

1. If there exists an invalid variable name, an error message will be displayed.
2. If there exists an invalid trajectory number, an error message will be displayed.
3. If there is no slice, a prompt message "No Slice" will be displayed, otherwise the program slice will be stored in the file named "\_cpptrace\_slice.dat".

#### E.2.2.2 Testing and System Design

There are a number of activities during system or architecture design that are relevant to testing [McDermid 93]. First the verification requirements were expanded so that they would correspond more closely to the individual tests.

A second testing-related activity, which should occur during this phase, is to develop the test coverage matrix. This is a matrix, which relates the expanded verification requirements to the modules, which implement the requirements.

A third activity is the development of the integration test strategy. This involves specifying the order in which the program units are to be added to the system, which is being built. A bottom-up strategy, instead of a top-down strategy, was used to test C++Debug, because it is easier to detect flaws that occur toward the bottom of a design. For example, using the verification requirement in previous example, we will have the following situation.

The number of tests will be expanded as follows:

- V 7.8/1 When the slice-criterion command is typed, with a user identification A, the error message will be displayed on the originating console.
- V 7.8/2 When the slice-criterion command is typed, with a user identification B, the error message will be displayed on the originating console.
- V 7.8/3 When the slice-criterion command is typed with a user identification which does not match a user currently logged on, the error message will be displayed on the originating console.

### E.2.2.3 Testing and Detailed Design

The main testing activity that occurs during this phase is the construction of C++Debug test procedures. A test procedure is a detailed step-by-step set of instructions [McDermid 93]. A test procedure contains details of the software configuration used, the hardware configuration, the location of the job control language commands necessary for carrying out the test, the files containing test data, the expected outcomes of the tasks, and the location of the files that contain the test outcome. For example,

```
bool UsedVariable( Type Var)
```

executed the function `UsedVariable` with parameter `Var` with type `Type`. If variable `Var` is a “used” set, then return **true**, otherwise return **false**.

### E.2.2.4 Testing and Programming

The primary activity in this phase is programming or coding the individual units or modules [McDermid 93]. Work may also be carried out on producing test harnesses or stubs. The second activity is the testing of the program units after they have been programmed. The aim of unit testing is to check that a program unit matches the specification produced for it during C++Debug system design. Unit testing is a structural testing activity, the aim being to ensure some degree of test thoroughness with respect to



some measure of structural coverage. A typical measure is that the test data generated should ensure that 100% of the statements in a unit are executed [McDermid 93]. Although this is a common metric, it is beginning to be regarded as inadequate, and the better metric of 100% statement coverage and 85% branch coverage is being gradually adopted in industry. For example, Figure 92 shows the template of function used variable and its path. We must make sure that every path in the program is tested.

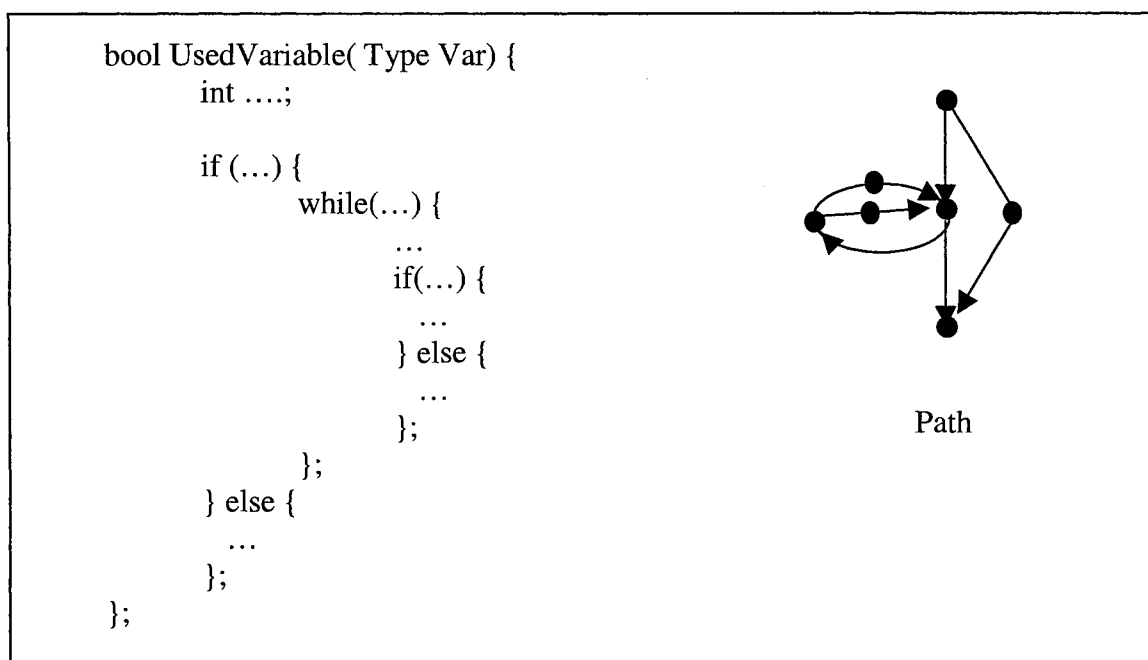


Figure 92. Part of function `UsedVariable` and its path

#### E.2.2.5 Testing and Integration

Testing during the integration phase will follow the plan set out in the system or architecture design [McDermid 93]. The primary aim of testing at this stage is to verify the design, but a subsidiary aim is to begin to verify the requirements.

After coding and testing, individually, tested program unit are produced. These units are then progressively integrated according to the agreed strategy, e.g., top-down or bottom-up. A number of specific facets of the design are tested, leading up to the testing of the full design and requirements functions. For example, in C++Debug, module coupling and cohesion, as depicted in Figure 93, was tested.

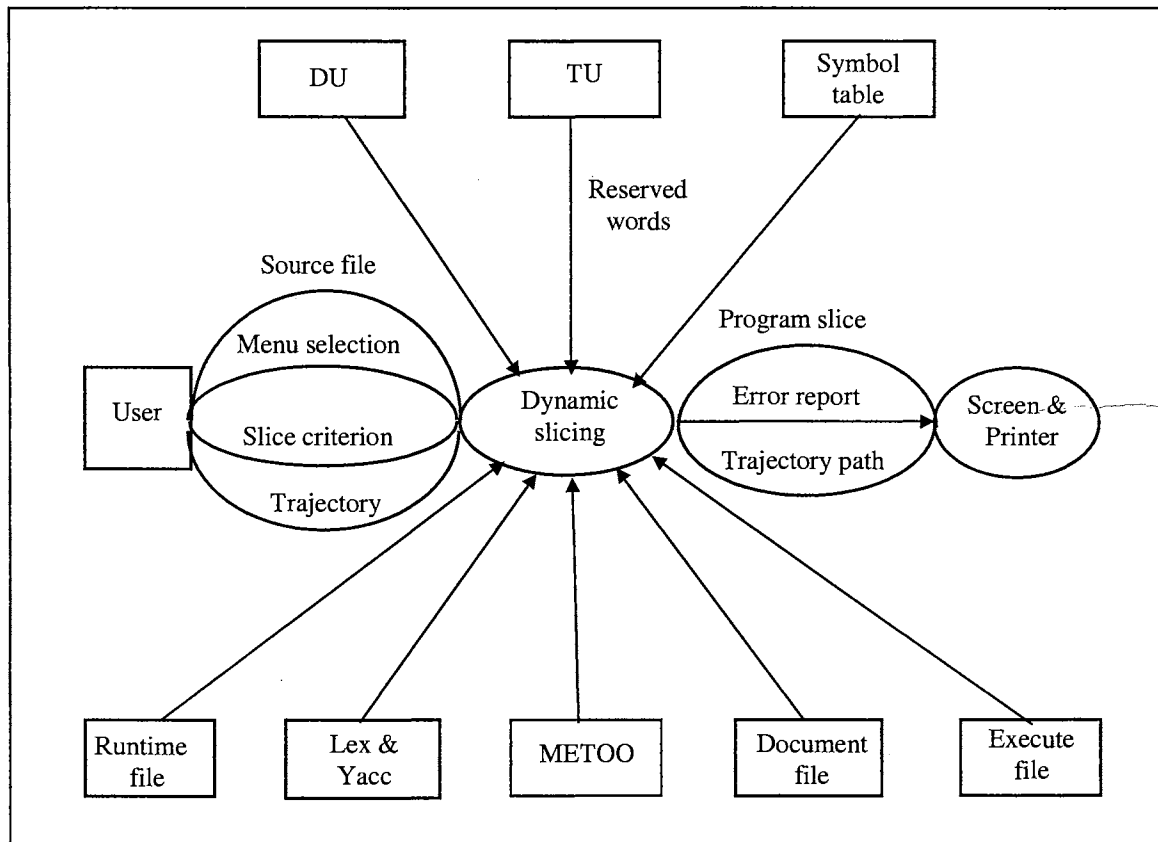


Figure 93. C++Debug block diagram

#### E.2.2.6 System and Acceptance Testing

In contrast to unit testing, system and acceptance testing are black-box activities [McDermid 93]. System testing is the process of executing the test procedures associated

with the verification tests. Acceptance testing is the process of executing the test procedures associated with a subset of the verification requirements that are agreed on by both the customer and the developer as being an adequate representation of the user requirements. The major difference between system and acceptance testing is the fact that the former takes place in a simulated environment. For example, in C++Debug all modules in Figure 93 were tested as a system.

#### E.2.2.7 Testing and Maintenance

The last testing activity associated with the software life cycle is regression testing. This occurs during maintenance after a system has been modified [McDermid 93]. Therefore, this kind of test will be not applied to C++Debug. Regression testing is the execution of a series of tests to check that a modification, applied during maintenance, has not affected the code corresponding to those function of the system which should be unaffected by the maintenance modification that had been carried out.

### E.2.3 Testing Techniques

The aim of this section is to show that the various techniques, which can be used to support the testing activities described in the previous section, were used to test C++Debug.

#### E.2.3.1 Random Testing

Random testing is a technique that is applicable for unit testing as a useful adjunct to other testing techniques [McDermid 93]. It involves identifying the input data space for a program unit and randomly generating test data from inside that space. In the case of C++Debug, a number of C++ programs were used for testing.

### E.2.3.2 Structural Testing

Structural testing involves testing a software system so that some structural metric or a particular path is traversed [McDermid 93]. In the case of C++Debug, every path of the control statements was tested.

### E.2.3.3 Assertion Testing

An assertion is a predicate that relates the value of variables in a program and describes a condition that must be true during the execution of a program unit or a program [McDermid 93]. In the case of C++Debug, for example,

```
Dice_A_B == Slice_A & Slice_B
```

represents a relation that must hold between the three variables used in the condition. Such conditions can be inserted by hand or by mean of software tools.

### E.2.3.4 Grammar-Based Testing

Grammar-based testing is based on describing the data set to be used in a test by means of some grammar formalism [McDermid 93]. In the case of C++Debug, for example from Figure 93, the C++ grammars, which implemented in the "Lex & Yacc" block, were tested.

### E.2.3.5 Functional Testing

Functional testing involves checking the functions of a computer system by means of examining either the system specification or the system design [McDermid 93]. In the case of C++Debug, for example, all functions in Appendices C and D were tested.

## E.3 Evaluation

### E.3.1 Introduction

C++Debug was evaluated based on Lyle [Lyle 84] and Gallagher's [Gallagher 90] approach by training several Computer Sciences graduate students at Oklahoma State University in its operation and by collecting data on how the students used C++Debug to locate faults in C++ programs. The main objective of the evaluation was how can C++Debug be used to enhance the debugging process and localize errors.

### E.3.2 Evaluation Procedure

The debugging process was studied by allowing each student to debug one program with and without C++Debug. There were four steps as listed below.

#### E.3.2.1 Step I: Familiarization

Let each student answer a questionnaire covering background information (see Subsection E.3.2.4), read an overview of the evaluation, and finally read the manual on how to use C++Debug (see Appendix B).

#### E.3.2.2 Step II: First Treatment

Let each student debug C++ programs without using the C++Debug tool. Each student can use other tools such as DBX, GDB, etc.

#### E.3.2.3 Step III: Second Treatment

In this step, the C++ programs in step II were debugged by using the C++Debug tool.

### E.3.2.4 Step IV: Subject Remarks

All information from Step I, Step II, and Step III were collected and analyzed based on Lyle's [Lyle 84] approach to find out:

1. Is C++Debug useful?
2. Are there some negative and positive comments?
3. What do they like about C++Debug?
4. What don't they like about C++Debug?

The students involved in the evaluation of C++Debug were asked to fill out a questionnaire based on Lyle's [Lyle 84] approach as follows.

#### Questionnaire

- (1) How long have you been programming (Years/Months)?
- (2) How many CS, (Computer Science), classes in your BS/BA?
- (3) How many CS classes taken so far in grad school?
- (4) How many other CS classes have you taken?
- (5) Which programming languages are you familiar with? Familiar means you used the language for at least a semester's work.
- (6) On a scale from 0 to 10, how familiar are you with C++?  
where  
0 = I've never used C++  
2 = I know some C++  
5 = I know C++ about average  
7 = I am comfortable with C++  
10 = I know C++ well

0            2            4            6            8            10  
+-----+-----+-----+-----+-----+-----+  
(put a check mark on the scale)

- (7) On the same scale from 0 to 10, how familiar are you with the VI or EMACS text editor?

0            2            4            6            8            10  
+-----+-----+-----+-----+-----+-----+  
(put a check mark on the scale)

- (8) Do you know about program slicing?

The subjects involved in the evaluation of C++Debug were ten graduate students at the Computer Science Department of Oklahoma State University. The student responses to the questions are summarized in Table IV and V. The number of changes made to the tested programs by each student, and the number of slices each student computed are shown in Table VI. And finally, edit times, compile times, and execution times are presented in Tables VII and VIII.

Variable	N	mean	sd	min	max	median
time_programming	10	8.1	3.0	3.0	13.0	8.5
n_bs_classes	10	7.4	5.8	0.0	15.0	8.5
n_ms_classes	10	10.2	2.5	6.0	15.0	9.5
n_other_classes	10	1.6	1.8	0.0	4.0	1.0
n_languages	10	7.9	2.2	4.0	12.0	8.0
skill_C++	10	7.1	2.5	2.0	10.0	8.0
skill_vi_or_emacs	10	7.8	2.9	1.0	10.0	9.0

Language	Number of Subjects
Assembler	3
C	9
C++	7
Java	6
Lisp	2
Pascal	3

Subject	Slices	Changes
1	4	3
2	*	2
3	*	3
4	1	3
5	*	2
6	*	3
7	4	3
8	8	7
9	3	5
10	*	3

\* not slicing

	N	mean	sd	min	max	median
Edit user time	10	832	397	352	1177	782
Edit system time	10	437	184	194	486	412
Compile user time	10	15490	2822	11882	16957	14510
Compile system time	10	4664	842	3872	5543	4602
Execute user time	10	580	223	391	774	460
Execute system time	10	845	212	618	1021	757

### E.3.3 Comments on C++Debug

Seven of the ten subjects reported that in the slicing mode C++Debug was very useful. In the dicing mode, four subjects reported that C++Debug can help them to locate errors in a program. Five subjects felt surprised that C++Debug could eliminate



irrelevant statements. Three subjects said that in the `-t` mode the trajectory path generated by C++Debug worked like the `cpptrace` tool in C, in an effective and useful manner.

Name	N	mean	sd	min	max	median
Edit user time	10	1224	1012	371	2501	902
Edit system time	10	713	633	286	1522	411
Compile user time	10	12501	492	11903	12833	12532
Compile system time	10	3962	557	3255	4482	3921
Execute user time	10	588	113	464	621	521
Execute system time	10	730	248	492	919	627

On the negative side, one subject felt that C++Debug was not more powerful than other debugging tools like GDB. Two subjects mentioned that the dicing process is quite complicated because of the process of selecting the appropriate slicing criteria (variables and positions for dicing). One subject mentioned that in the `-g` mode, C++Debug generated derivation tree that were too long, and that it was difficult to understand all of them.

## APPENDIX F

### SAMPLE PROGRAMS USED FOR THE COMPUTATION OF SLICING-BASED METRICS

The sample programs shown below were used for the computation of slicing-based metrics (see Section 4.8 for more detail). In order to compare the output obtained from C++Debug (which is based on dynamic slicing) with C-Sdicer (which is based on static slicing), the test programs must be the same as the ones used in Nanja's study in testing C-Sdicer [Nanja 90]. These test programs are provided bellow.

```
/******  
* TEST PROGRAM 1 *  
* This program reads a file of text, echoes the text and *  
* computes some statistics on lines, letters, words, and *  
* sentences. *  
*****/  
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
#define BIG 16384  
#define MAX_CHAR 80  
#define MAX_LINE 100  
#define TRUE 1  
#define FALSE 0  
main()  
{  
  char ll[MAX_CHAR][MAX_LINE];  
  char l[MAX_CHAR + 1];  
  char c, fname[20];  
  int let, word, i, j;  
  int nl,ncl,tnlet,tnw,tnl,tns,nlettw,nlett1,nwt1,nwts;  
  int mlettw,mlett1,mwt1,mwts,aletpw,aletpl,awpl,awps;  
  int xlettw,xlett1,xwts,xwt1,qlettw,qlett1,qwt1,qwts;  
  float sletpw,sletpl,swpl,swps;  
  FILE *infile;  
  nl = 0;  
  printf("\nEnter filename:");  
  gets(fname);  
  infile = fopen(fname,"r");
```

```

if (infile == NULL) {
    printf("%s does not exist --- program aborted...",fname);
    exit(1);
}
while (fgets(ll[nl],MAX_CHAR,infile) != NULL) {
    printf("%s",ll[nl]);
    ++nl;
}
tnl = nl;
nwts = 0;
mlettw = mlettl = mwts = BIG;
mwtl = qlettw = qlettl = qwtl = qwts = xlettw = 0;
xlettl = xwts = tnlet = tnw =tns = 0;
ncl = MAX_CHAR + 1;
for( i = 0; i < tnl; ++i) {
    strcpy(l,ll[i]);
    nwtl = nlettl = 0;
    word = FALSE;
    for (j = 0; j < strlen(ll[i]); ++j) {
        c = ll[i][j];
        if(isalpha(c)) {
            let = TRUE;
            ++nlettl;
        }
        if(let && !word) {
            ++nwtl;
            ++nwts;
            nlettw = 0;
            word = TRUE;
        }
        if(let && word)
            ++nlettw;
        if(!let && word) {
            word = FALSE;
            ++tnw;
            if(nlettw < mlettw)
                mlettw = nlettw;
            if(nlettw > xlettw)
                xlettw = nlettw;
            qlettw += nlettw * nlettw;
        }
        if(let)
            ++tnlet;
        if((c == '.' || c == '?' || c == '!') && (j != 0)) {
            ++tns;
            ++nlettl;
            if(nwts < mwts)
                mwts = nwts;
            if(nwts > mwts)
                xwts = nwts;
            qwts += nwts * nwts;
            nwts = 0;
        }
    }
    if(nlettl < mlettl)
        mlettl = nlettl;
    if(nlettl > xlettl)
        xlettl = nlettl;
    qlettl += nlettl * nlettl;
    nlettl = 0;
    if(nwtl < mwtl)
        mwtl = nwtl;
    if(nwtl > xwtl)
        xwtl = nwtl;
    qwtl += nwtl * nwtl;
    nwtl = 0;
}
if(tnw != 0) {
    aletpw = tnlet / tnw;
    if(tnw != 1)
        sletpw = sqrt((float)(qlettw - tnlet*tnlet/tnw)

```

```

        / (float)(tnw - 1));
    }
    if(tnw != 0) {
        aletpl = tnlet/tnl;
        if(tnl != 1)
            sletpl = sqrt((sqrt)(qlettl - tnl * tnl/tnl)
                / (sqrt)(tnl - 1));
    }
    if(tnl != 0) {
        awpl = tnw / tnl;
        if(tnl != 1)
            swpl = sqrt((float)(qwtl - tnw*tnw/tns)
                / (float)(tnl -1));
    }
    if(tns != 0) {
        awps = tnw/tns;
        if(tnl != 1)
            swps = sqrt((float)(qwts - tnw*tnw/tns)
                / (float)(tns - 1));
    }
    printf("\nmin %d %d %d %d",mlettw,mlettl,mwtl,mwts);
    printf("\nmax %d %d %d %d",xlettw,xlettl,xwtl,xwts);
    printf("\navg %d %d %d %d",aletpw,aletpl,awpl,awps);
    printf("\nssq %d %d %d %d",qlettw,qlettl,qwtl,qwts);
    printf("\nstd %d %d %d %d",sletpw,sletpl,swpl,swps);
    printf("\n\n letters words lines sentences ");
    printf("\n %d %d %d %d", tnlet,tnw,tnl,tns);
}

```

---

Program 1 (P1) Source: [Nanja 90]

```

/*****
*           TEST PROGRAM 2
*   This program inputs and echoes back integers,
*   beginning a new output line at each point where a comma
*   appears in the input. Each line is labeled, and at the
*   end of each output line, total of all integers on that
*   line is displayed. The input must itself consist of
*   only one line. Any characters other than digits and
*   commas are ignored, except as delimiters for the
*   numbers. The new line is used to detect the end of the
*   line.
*****/
#include <stdio.h>
main ( )
{
    char character, last_char;
    int line_total, next_line, current_number;
    line_total = 0;
    next_line = 2;
    current_number = 0;
    last_char = 0;
    printf("Type a line of integers,
        with a comma everywhere\n");
    printf("the line is to split. Any other characters\n");
    printf("are ignored: \n\n");
    scanf("%c", &character);
    printf("Line 1> ");
    while (character != '\n') {
        if(character == ',') {
            if(last_char >= '0' && last_char <= '9') {
                line_total += current_number;
                current_number = 0;
            }
            printf(" < total: %d\nLine %d> ",
                line_total, next_line);
            line_total = 0;
            next_line++;
        } else {

```

```

    if(character >= '0' && character <= '9') {
        current_number = current_number * 10
            + character - '0';
    } else {
        if(last_char >= '0' && last_char <= '9') {
            line_total += current_number;
            current_number = 0;
        }
    }
    last_char = character;
    scanf("%c",&character);
}
printf("< total: %d\n", line_total);
}

```

---

Program 2 (P2) Source: [Nanja 90]

```

/*****
*                               TEST PROGRAM 3                               *
*   Program to print histogram of word lengths.                             *
*****/
#include <stdio.h>
#define MAXSIZE 32
#define WIDTH 50
main(argc,argv)
int argc;
char *argv[];
{
    int i,n,maxlen,maxcount,tick;
    long lengths[MAXSIZE], total;
    char buffer[BUFSIZ], *gets();
    for(i = 0; i < MAXSIZE; i++) {
        lengths[i] = 0;
    }
    maxlen = 0;
    while(gets(buffer) != (char *) NULL) {
        n = strlen(buffer);
        if(n >= MAXSIZE) {
            lengths[0]++;
        }
        else {
            lengths[n]++;
            if(n > maxlen) {
                maxlen = n;
            }
        }
    }
    maxcount = 0;
    for(i = 0; i <= maxlen; i++) {
        if(lengths[i] > maxcount) {
            maxcount = lengths[i];
        }
    }
    printf("length\t|");
    for( i = 0; i <= WIDTH; i++) {
        printf("-");
    }
    printf("|count\n");
    tick = (maxcount + (WIDTH - 1)) / WIDTH;
    total = 0;
    if(lengths[0]) {
        n = lengths[0] / tick;
        printf("%5d+\t|*s*s%6d\n",i,n+1,"",
            (WIDTH - n + 1),"|",lengths[0]);
        total += lengths[0];
    }
    for(i = maxlen; i > 0; i--) {
        n = lengths[i] / tick;

```

```

printf("%5d+\t|*s*s*s%6d\n",i,n+1,"",
      (WIDTH - n + 1),"|",lengths[0]);
total += lengths[i];
}
printf("TOTAL\t|");
for(i = 0; i <= WIDTH; i++) {
printf("-");
}
printf("|%6d\n",total);
exit(0);
}

```

---

Program 3 (P3) Source: [Nanja 90]

```

/*****
*                               TEST PROGRAM 4                               *
*   Program to generate multiple coin toss samples.                       *
*****/
#include <stdio.h>
#define MAX RAND 2000
#define MODULUS 327681
#define SEMI_MOD (MODULUS %2)
main()
{
int index,start,nr_trials,nr_iter;
int head,tail,h_lead,t_lead,iter,curr_seed;
int mult1,mult2,incr1,incr2;
double ratio, lead_sum, side_sum;
double d_vals[MAX_RAND];

head = tail = h_lead = t_lead = 0;
lead_sum = side_sum = 0.0;
printf("\n Starting seed?");
scanf("%d",&curr_seed);
printf("\n Sample size?");
scanf("%d",&nr_trials);
printf("\n Number of samples to generate?");
scanf("%d",&nr_iter);
printf("\n First multiplier?");
scanf("%d",&mult1);
printf("\n First increment?");
scanf("%d",&incr1);
printf("\n Second multiplier?");
scanf("%d",&mult2);
printf("\n Second increment?");
scanf("%d",&incr2);
printf("Starting seed = %d\n\n",curr_seed);
printf("generating random values.....\n");
for(iter = 0; iter < nr_iter; iter++) {
head = 0;
tail = 0;
h_lead = 0;
t_lead = 0;
for( index = 0; index < nr_trials; index++) {
if(curr_seed >= SEMI_MOD)
start = (mult1 * curr_seed + incr1) % MODULUS;
else
start = (mult2 * curr_seed + incr2) % MODULUS;
if(start)
head++;
else
tail++;
if(head > tail)
h_lead++;
else if(tail > head)
t_lead++;
}
printf("%3d heads; %3d tails;",head,tail);
printf("H leads = %3d; T leads = %3d", h_lead, t_lead);
}
}

```

```

if(h_lead > t_lead)
    ratio = (double) h_lead / (h_lead + t_lead);
else
    ratio = (double) t_lead / (h_lead + t_lead);
d_vals[iter] = ratio;
lead_sum += ratio;
if(head > tail)
    side_sum += (double) head / nr_trials;
else
    side_sum += (double) tail / nr_trials;
printf("ratio = %.4lf\n",ratio);
}
printf("\n DONE \n");
printf("side_sum == %.4lf; mean side lead == %.4lf\n",
    side_sum,side_sum / nr_iter);
printf("lead_sum == %.4lf; mean lead == %.4lf\n",
    lead_sum,lead_sum / nr_iter);
}

```

---

Program 4 (P4) Source: [Nanja 90]

```

/*****
*                               TEST PROGRAM 5                               *
*   Program to compute correlation coefficients.                               *
*****/
#include <stdio.h>
#define MAX_VALS 50
#define MAX_STR 100
main()
{
    float c_vals[MAX_VALS];
    float d_vals[MAX_VALS];
    float sum1,sum2,var1,var2;
    float coeff,co_vari,numer,denom;
    int index ,n1,n2;
    char *null_str = "";
    char info[MAX_STR];

    printf(" Enter values for group 1.\n");
    printf("?");
    gets(info);
    index = 0;
    while( strcmp(info, null_str) != 0) {
        c_vals[index] = atoi(info);
        ++index;
        printf("?");
        gets(info);
    }
    n1 = index;
    printf(" Enter values for group 2.\n");
    printf("?");
    gets(info);
    index = 0;
    while( strcmp(info, null_str) != 0) {
        d_vals[index] = atoi(info);
        ++index;
        printf("?");
        gets(info);
    }
    n2 = index;
    if (n1 = n2) {
        sum1 = 0.0;
        for(index = 0; index < n1; index++)
            sum1 += c_vals[index];
        sum2 = 0.0;
        for(index = 0; index < n1; index++)
            sum2 += c_vals[index];
        co_vari = 0.0;
        for(index = 0; index < n1; index++)

```

```
    co_vari += c_vals[index] * d_vals[index];
numer = co_vari - (sum1 * sum2);
var1 = 0.0;
for (index = 0; index < n1; index++)
    var1 += c_vals[index] * c_vals[index];
for (index = 0; index < n1; index++)
    var2 += d_vals[index] * d_vals[index];
denom = (var1 - sum1 * sum2) * (var2 - sum2 * sum2);
denom = sqrt(denom);
if(denom != 0)
    coeff = numer / denom;
printf("r == %7.311f\n",coeff);
}
else
    printf("Arrays must be the same size.\n");
}
```

---

Program 5 (P5) Source: [Nanja 90]



## APPENDIX G

### SAMPLE C++DEBUG SOURCE CODE LISTING

C++Debug is comprised of 24 files. The following files contain code written in the C++ programming language. Some sample code segments are shown as well.

Makefile	// used to compile the programs
CDebug_Global.h	// used to declare all global constants and variables
main.h	// start the program
main.cpp	//
Menu.h	// manage the menu
Menu.cpp	//
CPPtrace.h	// create 'cpptrace' file
CPPtrace.cpp	//
SourceLine.h	// store all information of each line such as program line, sets, i.e., D, U, DU,
SourceLine.cpp	// etc.
SymbolTable.h	// keep track of all variables, constants, types, classes, templates, etc.
SymbolTable.cpp	//
LexYaccInitialize.h	// initialize some variables before using 'Lex' and 'Yacc'
Token.h	// defines 'Token'
l.l	// generate 'Lexer' to scan the string
y.y	// generate 'Parser' to analyze C++ grammar
Parser.h	// determine D-set, U-set, function prototype, called functions,
Parser.cpp	// calling functions
Utilities.cpp	// contains utility functions that are used to create C++Debug
Utilities.h	//
Slice.h	// to compute a slice
Slice.cpp	//
Dice.h	// to compute a dice
Dice.cpp	//



```

        C.cppttrace_usage();
    }
    argv[1][0] = '0';    // clear variable for lnnnn
    argv[1][1] = '0';    // clear variable for lnnnn
    if(C.NNNN<0) {
        C.NNNN = G_NNNN;
    } else {
        C.NNNN = atoi(argv[1]);
    }
    P.TRACE_ON = false;    // off grammar
                        // to set flag show on screen in cppttrace file
    if( ! P.CheckFileNotFound("_cppttrace_breakpoint.dat")) {
        system(" rm _cppttrace_breakpoint.dat ");
    }
    P.Parsing(argv[2]);
    system("g++ -o _cppttrace_ _cppttrace_.cpp");
    system("_cppttrace_");
    system("rm _cppttrace_*");
    exit(0);
    break;

    default:
        // 'C++Debug' manual
        C.cppttrace_usage();
        break;
    }
}
}
if(argc == 4) {
    if((argv[2][0] == '-') && (argv[2][1] == '1') ) {
        argv[2][0] = '0';    // clear variable for lnnnn
        argv[2][1] = '0';    // clear variable for lnnnn
        C.NNNN = atoi(argv[2]);
        if(C.NNNN<0) C.NNNN = G_NNNN;
    } else {
        printf("#### Filename not found\n");
        // 'C++Debug' manual
        C.cppttrace_usage();
    }
}

// argv[1] = FileName
Menu.MainMenu(argv[1]);
}

void MainInitialize() {

    // main program initialize

    // # of loop to show
    // set default to G_NNNN
    C.NNNN = G_NNNN;
    // set default not to show grammar
    P.TRACE_ON = false;

    // use EMACS by default
    P.VI = false;

    P.EMACS_WIN = false; // no windows

    // use Level 1 by default
    P.LEVEL2 = false;

    strcpy(P.LIBRARY, " -lGLU -lGL -lX11 -lm -lglut -lXext -lXi -lXmu " );

    P.FlagInfo = false;
}

#include " CDbg_Global.h "
#include <iostream>

```

```

#include <fstream>
#include <string>
#include <vector>
#include <map>
using namespace std;

// *****
//
// Class Name:   MapSingle
//
// Description:  Create one dimension arrays
//
// Data:
//   - LineNo   Arrays of type T_Elm
//   - Size     size of arrays of LineNo
//
// Methods:
//   - T_LineNo GetSize(void) return Size of arrays
//   - PutDat(LineNo, Dat)   Put data, Dat, at line LineNo
//   - GetDat(LineNo)       Get data, Dat, at line LineNo
//   - ReadFile(FileName)   Read data from file into an array
//   - Print()              Print data in the arrays on the screen
//
// *****

template <class T_LineNo, class T_Elm>
class MapSingle {
protected:
    T_Elm   *LineNo; // create one dimension arrays of size Size
    T_LineNo Size;
public:
    MapSingle(const T_LineNo &SizeIn) : Size(SizeIn) {LineNo = new T_Elm[Size+1];};
    T_LineNo GetSize(void) { return Size; }
    void     PutDat(const T_LineNo &line_no, const T_Elm &v) {LineNo[line_no] = v;};
    T_Elm    GetDat(const T_LineNo &line_no) { return LineNo[line_no]; }
    int      ReadFile(char *FileName);
    void     Print(void);

    ~MapSingle() { delete [] LineNo; }; // destructor
};

template <class T_LineNo, class T_Elm>
int MapSingle <T_LineNo, T_Elm>
::ReadFile(char *FileName) { // read data from file and put it
                            // into the arrays.
    ifstream in(FileName); // file name
    if(!in) {
        cout<<"### Cannot open "<<FileName<<" input file.\n";
        return 1;
    }

    T_LineNo i = 1; T_Elm temp;

    in>>temp;
    while((!in.eof()) && (i<Size)) { // read until EOF and < Size
        LineNo[i++] = temp;
        in>>temp;
    }
    in.close();
}

template <class T_LineNo, class T_Elm>
void MapSingle<T_LineNo, T_Elm>
::Print(void) { // show data in the arrays on the screen

    for(T_LineNo i=1; i<Size; i++) {
        cout<<i<<"{"<<LineNo[i]<<endl;
    }
}

```

```

// *****
//
// Class Name:  MapPair
//
// Description: Create arrays of type MAP, Standard Template Library
//
// Data:
// - LineNo    Type MAP supported by Standard Template Library
// - Size      Size of dynamic arrays
//
// Methods:
// - GetSize(void)          return Size of arrays
// - PutPair(line_no, v1, v2) put data v1, v2 at line_no
// - GetPair(line_no, v1)  return data v2 at line_no with key v1
// - Find(line_no, v1)     return TRUE if find v1 at line_no
// - LineUnion(LineNo1, LineNo2) Set union between LineNo 1 and 2
// - ReadFile(FileName)   read data from file and put in the arrays
// - PrintLine(line_no)   print data at line_no
// - PrintVar(void)       print pure data in the MAP
// - PrintSet(Act)        print data including Action set
// - Addr(line_no)        return address of LineNo[line_no]
//
// *****

template <class T_LineNo, class T_Elm1>
class MapSingle;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class MapPair {
protected:
    map <T_Elm1, T_Elm2> *LineNo; // create arrays of map pairs
    T_LineNo          Size;     // size of arrays
public:
    MapPair(const T_LineNo &Size);
    T_Elm2  GetSize(void) { return Size; };
    void    PutPair(const T_LineNo &line_no, const T_Elm1 &v1, const T_Elm2 &v2);
    T_Elm2  GetPair(const T_LineNo &line_no, const T_Elm1 &v1);
    bool    Find(const T_LineNo &line_no, const T_Elm1 &v1);
    void    LineUnion(const T_LineNo &LineNo1, const T_LineNo &LineNo2);
    int     ReadFile(char *FileName);
    void    PrintLine(const T_LineNo &line_no);
    void    PrintVar(void);
    void    PrintSet(MapSingle <T_LineNo, T_Elm1> &Act);
    map <T_Elm1, T_Elm2> Addr(const T_LineNo &line_no) { return LineNo[line_no];};

    ~MapPair(){delete []LineNo;}; // destructor
};

template < class T_LineNo, class T_Elm1, class T_Elm2>
MapPair<T_LineNo, T_Elm1, T_Elm2>
    ::MapPair(const T_LineNo &SizeIn) {

    Size = SizeIn;
    LineNo = new map <T_Elm1, T_Elm2> [Size+1];
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void MapPair< T_LineNo, T_Elm1, T_Elm2>
    ::PutPair(const T_LineNo &line_no, const T_Elm1 &v1, const T_Elm2 &v2) {

    LineNo[line_no].insert(pair< T_Elm1, T_Elm2>(v1 , v2));
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
T_Elm2 MapPair< T_LineNo, T_Elm1, T_Elm2>
    ::GetPair(const T_LineNo &line_no, const T_Elm1 &v1) {

    map<T_Elm1, T_Elm2>::iterator p;

    p = LineNo[line_no].find(v1);
    if(p!= LineNo[line_no].end()) return p->second; // if found
}

```

```

    else return (0); // if cannot find
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
bool MapPair< T_LineNo, T_Elm1, T_Elm2>
    ::Find(const T_LineNo &line_no, const T_Elm1 &v1) {

    map<T_Elm1, T_Elm2>::iterator p;

    p = LineNo[line_no].find(v1);
    if(p!= LineNo[line_no].end()) return true; // if found
    else return false; // if cannot find
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void MapPair<T_LineNo, T_Elm1, T_Elm2>
    :: LineUnion(const T_LineNo &LineNo1, const T_LineNo &LineNo2) {

    map<T_Elm1, T_Elm2>::iterator p; // set Union between two lines
    // and store in the first line

    p = LineNo[LineNo2].begin();

    while(p != LineNo[LineNo2].end()) {
        PutPair(LineNo1, p->first,p->second);
        p++;
    }
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void MapPair< T_LineNo, T_Elm1, T_Elm2>
    ::PrintLine(const T_LineNo &line_no) {

    map<T_Elm1, T_Elm2>::iterator p; // print data at line_no

    p = LineNo[line_no].begin();
    cout<<"Line No. "<<line_no<<": ";
    while(p != LineNo[line_no].end()) {
        cout<<"|"<<p->first<<","<<p->second<<"| ";
        p++;
    }
    cout<<endl;
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void MapPair< T_LineNo, T_Elm1, T_Elm2>
    ::PrintVar(void) {

    for(T_LineNo i = 0; i<Size; i++) // print all data
        PrintLine(i);
    cout<<endl;
}

template <class T_LineNo, class T_Elm1, class T_Elm2>
int MapPair<T_LineNo, T_Elm1, T_Elm2>
    ::ReadFile(char *FileName) {

    ifstream in(FileName); // D data input
    if(!in) {
        cout<<"### Cannot open "<<FileName<<" input file.\n";
        return 1;
    }

    T_Elm1 line, dat; // read data from file and put in the map_pair

    in>>line;
    while((!in.eof()) && (line < Size)) { // read until EOF and < Size
        in>>dat;
        while((!in.eof()) && (dat != 0)) { // read until EOF and < Size
            PutPair(line,dat,0);
            in>>dat;
        }
    }
}

```

```

        in>>line;
    }
    in.close();
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void MapPair< T_LineNo, T_Elm1, T_Elm2>
::PrintSet(MapSingle <T_LineNo, T_Elm1> &Act) {

    map<T_Elm1, T_Elm2>::iterator p; // print data with v1 = line number
                                   // v2 = trajectory
    for(T_LineNo i = 0; i<Size; i++) {
        p = LineNo[i].begin();
        cout<<"Line No. "<<i<<" ";
        while(p != LineNo[i].end()) {
            cout<<"{"<<Act.GetDat(p->first)<<","<<p->first<<"} ";
            p++;
        }
        cout<<endl;
    }
    cout<<endl;
}

// *****
//
// Class Name: Action
//
// Description: A trajectory will be illustrated in terms of
// a pair(instruction, its position in the trajectory), rather
// than the instruction itself, so as to distinguish between
// multiple occurrences of the same instruction in the trajectory.
// For example, instruction X at position P in T will be represented
// by pair(x,p). For ease of understanding, pair(x,p) will
// be replaced by Xp, and will be referred to as an action.
//
// Data:
// - Act Arrays of data class MapSingle
//
// Methods:
// - Action(Size) create object Act size Size
// - void PrintAction() print action sets
//
// *****

template <class T_LineNo, class T_Elm>
class MapSingle;

template <class T_LineNo, class T_Elm>
class Action {
protected: // data
    MapSingle <T_LineNo, T_Elm> Act; // class of one dimension arrays
public: // method
    Action(const T_LineNo &Size) : Act(Size) {}; // create object Act size Size
    void PrintAction(void) { // print action sets
        cout<<endl<<"<<< Action Set >>>"<<endl;
        Act.Print();
    }
};

// *****
//
// Class Name: Control
//
// Description: Test Action, an action Xp is a test action if X
// is a test instruction. Where Test Instruction statement is
// a control instruction such as an if-then-else or a
// while statement.
//
// Data:

```

```

// - Crt   Arrays of type MapSingle
//
// Methodes:
// - Control(Size)   create object Crt size Size
// - PrintControl() print control sets
//
// *****

template <class T_LineNo, class T_Elm>
class MapSingle;

template <class T_LineNo, class T_Elm>
class Control {
protected:
    MapSingle <T_LineNo, T_Elm> Ctr; // class of one dimension arrays
public:
    Control(const T_LineNo &Size) : Ctr(Size) {}; // create object Crt size Size
    void PrintControl(void) { // print control sets
        cout<<endl<<"<<<  Test-Control Set >>>"<<endl;
        Ctr.Print();
    }
};

// *****
//
// Class Name:   MapSingle       Single Arrays
// - PutDat(LineNo, Dat)   Put data, Dat, at line LineNo
// - GetDat(LineNo)       Get data, Dat, at line LineNo
// - ReadFile(FileName)   Read data from file
// - Print()              Print data in the arrays
//
// *****

template <class T_LineNo>
class ActionSize {
private:
    T_LineNo ActSize;
public:
    ActionSize(T_LineNo Size) { ActSize = Size; };
    T_LineNo GetActionSize(void) { return ActSize; };
};

// *****
//
// Class Name: Dset
//
// Description: D(Xp) Define, the set of variables that are
//              defined in action Xp.
// Data:
// - D         Defined sets, Arrays of data typ MapPair
//
// Methods:
// - Dset(Size)   Create D with class MapPair of size Size
// - PrintD()     Print D set
//
// *****

template <class T_LineNo, class T_Elm1, class T_Elm2>
class MapPair;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class Dset {
protected:
    MapPair <T_LineNo, T_Elm1, T_Elm2> D;
public:
    Dset(const T_LineNo &Size): D(Size) {};
    void PrintD(void);
};

template < class T_LineNo, class T_Elm1, class T_Elm2>

```



```

void Dset< T_LineNo, T_Elm1, T_Elm2>
  ::PrintD(void) {

    cout<<endl<<"<<< D sets >>>"<<endl;
    D.PrintVar();
}

// *****
//
// Class Name: Uset
//
// Description: U(Xp) The set of variables that are used in Xp.
//
// Data:
//   - U   a data of class MapPair
//
// Methods:
//   - Uset(Size) Create U of size Size
//   - PrintU()   Print U sets
//
// *****

template <class T_LineNo, class T_Elm1, class T_Elm2>
class MapPair;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class Uset {
protected:
    MapPair <T_Elm1, T_Elm1, T_Elm2> U;
public:
    Uset(const T_LineNo &Size): U(Size) { };
    void PrintU(void);
};

template < class T_LineNo, class T_Elm1, class T_Elm2>
void Uset< T_LineNo, T_Elm1, T_Elm2>
  ::PrintU(void) {

    cout<<endl<<"<<< U sets >>>"<<endl;
    U.PrintVar();
}

// *****
//
// Class Name: DUset
//
// Based classes:
//   - Dset D(Xp) Define, the set of variables that are defined
//     in action Xp.
//   - Uset U(Xp) The set of variables that are used in Xp.
//   - Action Used to find source line number from a trajectory
//
// Description: DU(Xp) Definition-Use Relation, a binary relation
// on M(T) in which one action assigns a value to an item of data
// and the other action uses that value.
//
// Data:
//   - DU   Data of class MapPair
//
// Methods:
//   - DUset(Size) Create DU of size = Size
//   - ComputeDU() Compute DU set from Dset and Uset by using
//     [Korel 90] Alg. Time Complexity O(N^2)
//   - PrintDU()   Print DU sets
//
// *****

template <class T_LineNo, class T_Elm1, class T_Elm2>
class MapPair;

```

```

template <class T_LineNo, class T_Elm1>
class Action;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class Dset;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class Uset;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class DUset : virtual public Action <T_LineNo, T_Elm1>,
              public Dset <T_LineNo, T_Elm1, T_Elm2>,
              public Uset <T_LineNo, T_Elm1, T_Elm2> {
protected:
    MapPair <T_LineNo, T_Elm1, T_Elm2> DU;
public:
    DUset(const T_LineNo &Size) : DU(Size), Action(Size), Dset(Size), Uset(Size) {};
    void    ComputedU(void);
    void    PrintDU(void);
};

template <class T_LineNo, class T_Elm1, class T_Elm2>
void DUset <T_LineNo, T_Elm1, T_Elm2>
    ::PrintDU(void) {

    cout<<"<<< DU sets >>>"<<endl;
    DU.PrintSet(Act);
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void DUset< T_LineNo, T_Elm1, T_Elm2>
    ::ComputeDU(void) {

    map<T_Elm1, T_Elm2>::iterator p;
    map<T_Elm1, T_Elm2> x;

    T_LineNo Size = DU.GetSize();

    for(T_LineNo i = 1; i<Size; i++) {
        x = D.Addr(i); // read all D sets at line i
        p = x.begin();

        while(p != x.end()) {
            for(T_LineNo j = i+1; j<Size; j++) {
                if(U.Find(j,p->first)) { // if find v in Uset at line j
                    DU.PutPair(i,j, p->first); // put v in Uset
                }
                if(D.Find(j,p->first)) { // break if v is re-defined
                    break;
                }
            }
            p++;
        }
    }
}

// *****
//
// Class name: TCset
//
// Based classes:
// - Action Used to find a source line number from a trajectory
// - Control Used to find which line is a test-action such as
//   if, while, switch, etc.
//
// Description: Test-Control relation, a binary relation on M(T),
// captures the effect between test actions and actions that have
// to be chosen to execute by these test actions.
//
// Data:

```

```

// - TC Data of class MapPair
//
// Methods:
// - TCset(Size) Create TC of size = Size
// - ComputeTC(void) Compute TC relation set by using [Korel 90]
// Alg. Time complexity = O(N^2)
// - PrintTC() Print all IR sets
//
// *****

template <class T_LineNo, class T_Elm1>
class Action;

template <class T_LineNo, class T_Elm1>
class Control;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class MapPair;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class TCset : virtual public Action <T_LineNo, T_Elm2>,
              virtual public Control <T_LineNo, T_Elm2> {
protected:
    MapPair <T_LineNo, T_Elm1, T_Elm2> TC;
public:
    TCset(const T_LineNo &Size) : TC(Size), Action(Size), Control(Size) {};
    void ComputeTC(void);
    void PrintTC(void);
};

template <class T_LineNo, class T_Elm1, class T_Elm2>
void TCset <T_LineNo, T_Elm1, T_Elm2>
    ::PrintTC(void) {

    cout<<"<<< TC sets >>>"<<endl;
    TC.PrintSet(Act);
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void TCset< T_LineNo, T_Elm1, T_Elm2>
    ::ComputeTC(void) {

    T_LineNo j, k;
    T_Elm2 Dat;
    T_Elm1 Size = TC.GetSize();

    for(T_LineNo i = 1; i<Size; i++) {
        if(Act.GetDat(i) != 0) { // looking for the test action line
            Dat = Act.GetDat(i);

            j = i+1; // if found, then looking for another that identity
            while((Dat != Act.GetDat(j)) && (j<Size)) {
                j++;
            }
            if(j<Size) { // if found, then put lines between them into TC sets
                for(k = i+1; k<j; k++)
                    TC.PutPair(i,k, Act.GetDat(k));
            }
        }
    }
}

// *****
//
// Class Name: IRset
//
// Based class:
// - Action Used to find a source line number from a trajectory
// - Control Used to find which line is a test-control such as
// if, while, switch, etc.

```

```

//
// Description: Let Xp IR Yt, iff X = Y is the identity relation
//             IR on M(Front(T,q)).
//
// Data:
// - IR Data of class MapPair
//
// Methods:
// - IRset(Size) Create IR of size = Size
// - ComputeIR(void) Compute IR relation set by using [Korel 90]
//   Alg. Time complexity = O(N^2)
// - PrintIR() Print all IR sets
//
// *****

template <class T_LineNo, class T_Elm1>
class Action;

template <class T_LineNo, class T_Elm1>
class Control;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class MapPair;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class IRset : virtual public Action <T_LineNo, T_Elm2>,
              virtual public Control <T_LineNo, T_Elm2> {
protected:
    MapPair <T_LineNo, T_Elm1, T_Elm2> IR;
public:
    IRset(const T_LineNo &Size) : IR(Size), Action(Size), Control(Size) {};
    void ComputeIR(void);
    void PrintIR(void);
};

template <class T_LineNo, class T_Elm1, class T_Elm2>
void IRset <T_LineNo, T_Elm1, T_Elm2>
::PrintIR(void) { // print all IR sets

    cout<<"<<< IR sets >>>"<<endl;
    IR.PrintSet(Act);
}

template <class T_LineNo, class T_Elm1, class T_Elm2>
void IRset< T_LineNo, T_Elm1, T_Elm2>
::ComputeIR(void) {

    T_LineNo j;
    T_Elm2 Dat;
    T_Elm1 Size = IR.GetSize();

    for(T_LineNo i = 1; i<Size; i++) {
        if(Act.GetDat(i) != 0) { // looking for line that is a test-control
            Dat = Act.GetDat(i); // such as if, while, switch, etc.

            j = i+1; // looking for a test-control line
            while((Dat != Act.GetDat(j)) && (j<Size)) {
                j++; // if line i is a test control, then looking for
            } // another line that identity
            if(j<Size) { // found the line
                IR.PutPair(i,j, Dat); // put line that identity for the i-j
                IR.PutPair(j,i, Dat); // put line that identity for the j-i
            }
        }
    }
}

// *****
//
// Class Name: Zset

```

```

//
// Based classes:
// - Action Used to find a source line number from a trajectory
// - Control Used to find which line is a test-control such as
//   if, while, switch, etc.
// - DUset DU(Xp) Definition-Use Relation
// - IRset Let Xp IR Yt, iff X = Y is the identity relation
//   IR on M(Front(T,q)).
// - TCset Test-Control relation
//
// Description: A Union set ofDUset, TCset, and IRset.
//
// Data:
// - Z Data of class MapPair
//
// Methods:
// - Zset(Size) Create DU of size = Size
//               Time complexity O(N^2)
// - ComputeZ() Compute Z sets from DUset, IRset, and TCset.
//               Time Complexity O(N^2)
// - PrintZ()   Print Z sets
//
// *****
template <class T_LineNo, class T_Elm1, class T_Elm2>
class MapPair;

template <class T_LineNo, class T_Elm1>
class Action;

template <class T_LineNo, class T_Elm1>
class Control;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class DUset;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class IRset;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class Zset : virtual public Action <T_LineNo, T_Elm1>,
             virtual public Control <T_LineNo, T_Elm1>,
             public DUset <T_LineNo, T_Elm1, T_Elm2>,
             public TCset <T_LineNo, T_Elm1, T_Elm2>,
             public IRset <T_LineNo, T_Elm1, T_Elm2> {
protected:
    MapPair <T_LineNo, T_Elm1, T_Elm2> Z; // create arrays of Z set
public:
    Zset(const T_LineNo &Size) : Z(Size), Action(Size), Control(Size),
                               DUset(Size), IRset(Size), TCset(Size) {} ;
    void ComputeZ(void);
    void PrintZ(void);
};

template < class T_LineNo, class T_Elm1, class T_Elm2>
void Zset< T_LineNo, T_Elm1, T_Elm2>
    ::PrintZ(void) {

    cout<<"<<< Z sets >>>"<<endl;
    Z.PrintSet(Act);
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void Zset< T_LineNo, T_Elm1, T_Elm2>
    ::ComputeZ(void) {

    map<T_Elm1, T_Elm2>::iterator pDU; // point to DU sets
    map<T_Elm1, T_Elm2> xDU;

    map<T_Elm1, T_Elm2>::iterator pIR; // point to IR sets
    map<T_Elm1, T_Elm2> xIR;

```

```

map<T_Elm1, T_Elm2>::iterator pTC; // point to TC sets
map<T_Elm1, T_Elm2> xTC;

for(T_LineNo i = 1; i<Z.GetSize(); i++) {

    xDU = DU.Addr(i); // put all DU sets into Z sets
    pDU = xDU.begin();
    while(pDU != xDU.end()) {
        Z.PutPair(i,pDU->first,pDU->second);
        pDU++;
    }

    xIR = IR.Addr(i); // put all IR sets into Z sets
    pIR = xIR.begin();
    while(pIR != xIR.end()) {
        Z.PutPair(i,pIR->first,pIR->second);
        pIR++;
    }

    xTC = TC.Addr(i); // put all TC sets into Z sets
    pTC = xTC.begin();
    while(pTC != xTC.end()) {
        Z.PutPair(i,pTC->first,pTC->second);
        pTC++;
    }
}
}

// *****
//
// Class Name: SliceCriterion
//
// Description: The specification that the behavior of interest can be
// expressed as the values of a set of the variables at a subset of
// the statements.
//
// *****

template <class T_LineNo, class T_Elm1>
class SliceCriterion {
private:
    T_LineNo q;
    T_Elm1 v;
public:
    SliceCriterion(void) { q = 0; v = 0; };
    SliceCriterion(const T_LineNo &Q, const T_Elm1 &V) { q = Q; v = V; };
    void PutQ(const T_Elm1 &Q) { q = Q; };
    void PutV(const T_Elm1 &V) { v = V; };
    T_Elm1 GetQ(void) { return q; };
    T_Elm1 GetV(void) { return v; };
};

// *****
//
// Class name: SliceSet
//
// Based class:
// - Action Used to find a source line number from a trajectory
//
// Description: Based on the premise that instead of localizing
// errors in the original program, which can be of a large size,
// one can locate such errors in a program of smaller size which
// is sliced from the original program but still preserves part
// of the original program's behavior for a particular input or
// relative to a particular variable.
//
// Data:
// - SN A number of a current program slice
// - Slice Set of program slices

```

```

//
// Methods:
// - SliceSet(Size) Create object Slice of class MapPair.
// - PutSN(v) Set value of SN
// - GetSN(void) return SN
// - GetFinalSlice() Transfar slice from S0 to SN
// - PrintSlice() Print slice sets
//
// *****

template <class T_LineNo, class T_Elm1>
class Action;

template <class T_LineNo, class T_Elm1>
class SliceCriterion;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class SliceSet : virtual public Action <T_LineNo, T_Elm1> {
protected:
    T_LineNo SN;
    MapPair <T_LineNo, T_Elm1, T_Elm2> Slice;
public:
    SliceSet(const T_LineNo &Size) : Slice(Size), Action(Size), SN(1) {};
    void PutSN(const T_LineNo &v) { SN = v; };
    T_LineNo GetSN(void) { return SN; };
    void PrintSlice(void);
    void GetFinalSlice(void);
};

template < class T_LineNo, class T_Elm1, class T_Elm2>
void SliceSet< T_LineNo, T_Elm1, T_Elm2>
::GetFinalSlice(void) {

    map<T_Elm1, T_Elm2>::iterator p;
    map<T_Elm1, T_Elm2> x;
    T_LineNo Size = Slice.GetSize();

    x = Slice.Addr(0);
    p = x.begin();

    while(p != x.end()) { // transfer slice from S0 to SN
        Slice.PutPair(SN,Act.GetDat(p->first),p->first);
        p++;
    }
    SN++;
    if(SN >= Size) SN = 1;
}

template <class T_LineNo, class T_Elm1, class T_Elm2>
void SliceSet< T_LineNo, T_Elm1, T_Elm2>
::PrintSlice(void) {
    cout<<"<<< Slice >>>"<<endl;
    Slice.PrintVar();
}

// *****
//
// Class name: ComputeSliceSet
//
// Based classes:
// - Action Used to find a source line number from a trajectory
// - Control Used to find which line is a test-control such as
// if, while, switch, etc.
// - Z class A union set of DUsset, IRset and TCset
// - SliceSet Sets of slice programs.
// - Dice class, Sets of pieaces of programs
//
// Description: Compute a slice set based on [Korel 90] Alg.
// Time complexity =
// Data:
// - A A set of all actions

```

```

// - S A set of all slices
//
// Methods:
// - ComputeSliceSet(SizeIn) Create A, S of size Size
//     Time complexity = O(N)
// - LastD(Crit) Compute Last-Defined with Slice criterion, Crit
//     Time complexity = O(N)
// - LastT(Crit) Compute Test-Control
//     Time complexity = O(N)
// - ComputeA0S0(Crit) Compute A0 and S0
// - ComputeAc(line_no, Crit) Compute Ac, c = {1,2,3, ... ,}
//     Time complexity = O(N)
// - ComputeSc(LineNo) Compute Sc, c = {1,2,3, ... ,}
// - ComputeSlice(Crit) Compute a program slice
//     Time complexity = O(N)
// - PrintLastD_T(Crit) Print Last Defined and Test-Control
// - PrintA(void) Print A sets
// - PrintS(void) Print S sets
// - TestProgramSlice() To check that program slice working
//     properly with the sample data
//
// *****

template <class T_LineNo, class T_Elm1>
class Action;

template <class T_LineNo, class T_Elm1>
class Control;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class Zset;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class SliceSet;

template <class T_LineNo, class T_Elm1, class T_Elm2>
class ComputeSliceSet : virtual public Action <T_LineNo, T_Elm1>,
                        virtual public Control <T_LineNo, T_Elm1>,
                        public Zset <T_LineNo, T_Elm1, T_Elm2>,
                        public SliceSet <T_LineNo, T_Elm1, T_Elm2> {
private:
    MapPair <T_LineNo, T_Elm1, T_Elm2> A; // create arrays of A sets
    MapPair <T_LineNo, T_Elm1, T_Elm2> S; // create arrays of S sets
public:
    ComputeSliceSet(const T_LineNo &SizeIn);
    void PrintA(void);
    void PrintS(void);
    void PrintLastD_T(SliceCriterion <T_LineNo, T_Elm1> &Crit);
    T_Elm2 LastD(SliceCriterion <T_LineNo, T_Elm1> &Crit);
    T_Elm2 LastT(SliceCriterion <T_LineNo, T_Elm1> &Crit);
    void ComputeA0S0(SliceCriterion <T_LineNo, T_Elm1> &Crit);
    bool ComputeAc(const T_LineNo &LineNo, SliceCriterion <T_LineNo, T_Elm2> &Crit);
    void ComputeSc(const T_LineNo &LineNo);
    void ComputeSlice(SliceCriterion <T_LineNo, T_Elm1> &Crit);
    void TestProgramSlice(void);
};

template < class T_LineNo, class T_Elm1, class T_Elm2>
ComputeSliceSet<T_LineNo, T_Elm1, T_Elm2>
    ::ComputeSliceSet(const T_LineNo &Size): A(Size), S(Size),
                                           Action(Size),
                                           Control(Size),
                                           Zset(Size),
                                           SliceSet(Size) { };

template < class T_LineNo, class T_Elm1, class T_Elm2>
void ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
    ::PrintLastD_T(SliceCriterion <T_LineNo, T_Elm1> &Crit) {

    cout<<"Slice Criterion at V = "<<Crit.GetV()<<" , Q = "<<Crit.GetQ()<<endl<<endl;
    cout<<"Last Def = "<<LastD(Crit)<<endl;

```



```

    cout<<"Last Test = "<<LastT(Crit)<<endl<<endl;
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
  ::PrintA(void) {

    cout<<"<<< A sets >>>"<<endl;
    A.PrintSet(Act);
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
  ::PrintS(void) {

    cout<<"<<< S sets >>>"<<endl;
    S.PrintSet(Act);
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
T_Elm2 ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
  ::LastD(SliceCriterion <T_LineNo, T_Elm1> &Crit) {

    T_Elm1 v = Crit.GetV();
    T_LineNo i = Crit.GetQ() - 1;

    while(i > 0) { // looking for v, last defined in Dset
        if(D.Find(i,v) == true)
            return i;
        i--;
    }
    return 0; // if cannot find
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
T_Elm2 ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
  ::LastT(SliceCriterion <T_LineNo, T_Elm1> &Crit) {

    T_Elm1 v = Crit.GetV();
    T_LineNo i = Crit.GetQ() - 1;

    while(i > 0) { // looking for its scope
        if(TC.Find(i,v) == true)
            return i;
        i--;
    }
    return 0; // if cannot find
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
bool ComputeSliceSet<T_LineNo, T_Elm1, T_Elm2>
  :: ComputeAc(const T_LineNo &LineNo, SliceCriterion <T_LineNo, T_Elm2> &Crit) {

    T_LineNo Line;
    Line = LineNo - 1;

    T_LineNo sn = 0; // Slice Number 0, compute slice in this number
                    // and translate it later into the number SN
    map<T_Elm1, T_Elm2>::iterator p;
    map<T_Elm1, T_Elm2> x;

    bool FlagDone = true;

    x = S.Addr(Line);
    p = x.begin();

    while(p != x.end() ) {
        for(T_LineNo i = Crit.GetQ(); i>0; i--) {
            if(Z.Find(i,p->first)) {
                if(!Slice.Find(sn, i)) {
                    Slice.PutPair(sn, i,0);
                }
            }
        }
    }
}

```

```

        A.PutPair(LineNo, i,0);
        FlagDone = false;
    }
}
}
p++; // if there is a new set, compute A(c+1)
}
return FlagDone; // nothing change in Ac set, end compute
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void ComputeSliceSet<T_LineNo, T_Elm1, T_Elm2>
:: ComputeSc(const T_LineNo &LineNo) {

    T_LineNo Line;

    if((Line = LineNo - 1) < 0)
        cout<<"### Error in compute SI"<<endl;

    map<T_Elm1, T_Elm2>::iterator p;
    map<T_Elm1, T_Elm2> x;

    x = S.Addr(Line);
    p = x.begin();

    // Sc = S(c-1) + Ac
    while(p != x.end()) { // combine with S(c-1)
        S.PutPair(LineNo, p->first,p->second);
        p++;
    }

    x = A.Addr(LineNo);
    p = x.begin();

    while(p != x.end()) { // combine with Ac
        S.PutPair(LineNo, p->first,p->second);
        p++;
    }
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
::ComputeA0S0 (SliceCriterion <T_LineNo, T_Elm1> &Crit) {

    T_Elm1 v = Crit.GetV();
    T_LineNo line_no = Crit.GetQ();

    T_LineNo LD = LastD(Crit);

    A.PutPair(0, LD, 0); // compute A0 and S0
    S.PutPair(0, LD, 0); // and then put them into A0 and S0 sets
    Slice.PutPair(0, LD, 0);

    T_LineNo LT = LastT(Crit);

    if(LT) { // put Last Test-Control, if has
        A.PutPair(0, LT, 0); // put them into A0 and S0 sets
        S.PutPair(0, LT, 0);
        Slice.PutPair(0, LT, 0);
    }
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
::ComputeSlice(SliceCriterion <T_LineNo, T_Elm1> &Crit) {

    T_LineNo Size = A.GetSize();

    ComputeA0S0(Crit);

    T_LineNo i = 1; // compute Ac, Sc, where c = {1,2,3, ... ,}
    while((i<Size) && (!ComputeAc(i,Crit))) {

```

```

        ComputeSc(i);
        i++;
    }
    Slice.PutPair(0,Crit.GetQ(),0);
}

template < class T_LineNo, class T_Elm1, class T_Elm2>
void ComputeSliceSet< T_LineNo, T_Elm1, T_Elm2>
    ::TestProgramSlice(void) {

    Act.ReadFile("action.dat"); // test action sets
    PrintAction();

    Ctr.ReadFile("control.dat"); // test control sets
    PrintControl();

    D.ReadFile("D.dat"); // test D sets
    PrintD();

    U.ReadFile("U.dat"); // test Usets
    PrintU();

    ComputeDU(); // test DU sets
    PrintDU();

    ComputeIR(); // test IR sets
    PrintIR();

    ComputeTC(); // test TC sets
    PrintTC();

    ComputeZ(); // test Z sets
    PrintZ();

    // assign Slice criterion
    SliceCriterion <NUMTYPE, NUMTYPE> Crit(15, 2);

    ComputeSlice(Crit); // test compute slice
    PrintLastD_T(Crit); // check data, lats define, Test-Control
    PrintA(); // check data, Ac sets
    PrintS(); // check data, Sc sets
    PrintSlice(); // check slice at line 0
    GetFinalSlice(); // move slice from line 0 to line SN
    PrintSlice(); // check slice at line SN
}

```

## VITA 2

Winai Wichaipanitch

Candidate for the Degree of

Doctor of Philosophy

Thesis: AN INTERACTIVE DEBUGGING TOOL FOR C++ BASED ON  
DYNAMIC SLICING AND DICING

Major Field: Computer Science

### Biographical:

Personal Data: Born in Lopburi, Thailand, October 23, 1958, the son of Arun and Tonghaw. Married to Cholada Singhasurasakdhi on March 9, 1984.

Education: Graduated from Lopburi Vocational College, Lopburi, Thailand, in May 1977; received the Bachelor of Science in Electrical Engineering degree with a Major in Electronics from the Rajamangala Institute of Technology, Bangkok, Thailand in April 1984; received the Master of Science degree in Computer Science at the Computer Science Department of Oklahoma State University in December 1992; completed the requirements for the Doctor of Philosophy degree in Computer Science at the Computer Science Department of Oklahoma State University in August 2003.

Professional Experience: Instructor, Department of Electrical Engineering, Rajamangala Institute of Technology, 1979 to present.