A Novel Approach of Dynamic Slicing and Debugging of Object Oriented Programs

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

In

Software Engineering

Submitted By

Paritosh Jain

(800931015)

Under the supervision of:

Dr. Rajesh Bhatia
Professor, CSED
DCRUST,Murthal(Sonepat)

Mr. Karun Verma
Assistant Professor
CSED,Thapar University

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

June 2011

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"A Novel Approach of Dynamic Slicing of Object Oriented Programs"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Rajesh Bhatia and Mr. Karun Verma** and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Signature:

Paritosh Jain

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Dr. Rajesh Bhatia
Professor
CSED, DCRUST Murthal (Sonepat)

Mr. Karun Verma
Assistant Professor
CSED,Thapar University

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life.

This work would not have been possible without the encouragement and able guidance of my supervisor Dr. Rajesh Bhatia and Mr. Karun Verma. I thank my supervisors for their time, patience, discussions and valuable comments. Their enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to Dr. Maninder Singh, Associate Professor & Head, Computer Science & Engineering Department, a nice person, an excellent teacher and a well – credited researcher, who always encouraged me to keep going with work and always advised me with his invaluable suggestions.

I will be failing in my duty if I don’t express my gratitude to **‘**Dr. Abhijit Mukherjee’, Director of the University**,** for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my family whom I dearly miss and without whose blessings none of this would have been possible. To my parents, I own thanks for their wonderful love and encouragement. I would also like to thank my brother, since he insisted that I should do so. I would also like to thank my close friends for their constant support.

Paritosh Jain

(800931015)

ABSTRACT

Software testing and debugging is the necessary phase of software development, lots of effort has been deployed to detect the bugs and debug the program. Program slicing is a technique to extract program parts with respect to some special computation. Program Slicing is to remove the irrelevant statements from the program code. Irrelevant statements are those statements to which the buggy statement is neither data dependent nor control dependent. Program slicing aids the programmer to reduce the debugging effort.

Computation of slices needs an intermediate representation of the program, and then this intermediate representation is used to compute the slices. Various dependence graphs have been proposed to represent the program like program dependence graph. Since these graphs are very complex it is very difficult to convert the program into these intermediate graphs. The proposed approach introduces another intermediate representation of the programs whose space complexity is lesser.

Detecting and locating bugs is the major task of the debugging process. Here we detect most possible bugs in the program with polymorphic behavior. The debugger also generates the suggestion messages for the detected bugs. These suggestion messages can help the programmer in debugging the program.

Table of Content

Certificate i

Acknowledgement ii

Abstract iii

Table of Content iv

List of Figures vii

List of Tables viii

List of Abbreviations ix

Chapter 1 Introduction **1**

1.1 Background 1
1.2 Motivation for our Project 2
1.3 Basic Concepts 2
1.3.1 Slicing 2
1.3.2 Slicing Criteria 3
1.3.3 Types of Slicing 3
1.3.4 Static Slicing 3
1.3.5 Dynamic Slicing 3
1.3.6 Statement Dependencies 4
1.3.7 Data Dependency 4
1.3.8 Control Dependency 5
1.3.9 Forward Slice 5
1.3.10 Backward Slice 6
1.3.11 Control Flow Graph 7
1.3.12 Data Dependence Graph 7
1.3.13 Program Dependence Graph 8
1.3.14 System Dependence Graph 9
1.3.15 Object Oriented Program Dependence Graph 10
1.3.16 Extended System Dependence Graph 12
1.3.17 Execution Trace 12

1.3.18 Defined-Used Chain 12
1.3.19 PSPO 13
1.3.20 GPSPO 13
Chapter 2 Literature Survey 14
2.1. Call Stack-Sensitive Slicing 15
2.2. Slicing of Object Oriented Programs 17
2.3. Dynamic Slicing of Object Oriented Programs 19
2.4. Debugging of Object Oriented Programs in Case of Polymorphism 23
2.4.1. State Definition Anomaly 24
2.4.2. State Definition Inconsistency 24
2.4.3. State Defined Incorrectly 24
Chapter 3 Problem Statement 25
3.1. Gap Analysis 25
3.2. Concise Problem Statement 26
3.3. Justification 26
Chapter 4 Proposed Methodology 27
4.1. System Overview 27
4.2. Experimental Setup 28
4.2.1. Defined-used Info 28
4.2.2. ClassInfo 28
4.2.3. ObjectInfo 28
4.3. Proposed Algorithm for Slice Computation 29
Chapter 5 Experimental Results 31
5.1. Slice Computation 31
5.2. Slice Computation of Bugged Program 34
5.2.1. Input Program with Bug State Definition Anomaly 34
5.2.2. Input program with Bug State Definition Inconsistency 35
5.2.3. Input program with Bug State Defined Incorrectly 36
5.2.4. Input Program with Parameter Mismatch bug 37
Chapter 6 Conclusion and Future Scope 38
6.1. Conclusion 38

6.2. Future Scope 38

References **39**

List of Papers Presented **42**

List of Figures

Fig 1.1 The dependency graph of the example program	6
Fig 1.2 Control Flow Graph of the sample Program	7
Fig 1.3 Data Dependence Graph of the example program	8
Fig 1.4 Program Dependence Graph of the example program	9
Fig 1.5 System Dependence Graph of the Example Program	10
Fig 1.6 Object Oriented Program Dependence Graph of the example Program	11
Fig 2.1 An execution trace of the example program on input argv[1] = 3.	21
Fig 2.2 example class diagram of program having bugs	23
Fig 4.1 Workflow of the proposed approach	27
Fig 4.2 Software Debugger	29
Fig 5.1 Debugger generating the warning message	32
Fig 5.2 Debugger asking the user whether the polymorphic call is correct	33
Fig 5.3 Debugger Showing the bugged lines in RED	33
Fig 5.4a Debugger showing details of the bug SDA present in the input program	34
Fig 5.4b Suggestion messages for bug SDA	34
Fig 5.5a Debugger showing details of the bug SDIH present in the input program	35
Fig 5.5b Suggestion messages for bug SDIH	35
Fig 5.6a Debugger showing details of the bug SDI present in the input program	36
Fig 5.6b Suggestion messages for bug SDI	36
Fig 5.7a Debugger showing details of the bug Parameter Mismatch	37
Fig 5.7b Suggestion messages for bug and the bugged lines in red	37

List of Tables

Table 3.1: Gap Analysis between the proposed approach and existing approaches 26

List of Abbreviations

OOPs:- Object Oriented Programs

CFG:- Control Flow Graph.

DDG:- Data Dependence Graph.

PDG:- Program Dependence Graph.

SDG:- System Dependence Graph.

OPDG:- Object Oriented Program Dependence Graph.

ESDG:- Extended System Dependence Graph.

CLDG:- Class Dependence Graph.

d-u:- Deifned-Used.

PSPO:- Path Slice Per Object.

GPSPO:- Generator Path Slice Per Object.

SDA :- State Definition Anomaly.

SDIH :- State Definition Inconsistency.

SDI :- State Defined Incorrectly.

Chapter 1

Introduction

1.1 Background

Most of the transactions performed in today's world use different types of software solutions. These software solutions are becoming quite complex and their quality have been primarily bounded by the cost and time factors. Also, the focus of building software has seen a dramatic drift from using traditional procedural techniques to object-oriented techniques. Object oriented technique, no doubt modularizes the program, but at the same time, it is very complex and difficult to debug and test for errors. It has been found that almost 50% of the softwares built today go unused because of their inability to meet the above mentioned constraints, which in turn results in a huge loss of time, money and manpower. Software testing activities are hence very essential for the construction of reliable software.

Various methods have been developed to test softwares for errors. These methods apply different approaches toward software testing which use various intermediate forms. Intermediate graph representation of a program is one such convenient representation. It includes various graphs like control flow graph, data dependence graph, control dependence graph, program dependence graph, system dependence graph, etc to represent the program structure and the relations between different program constructs. This representation can be further used in different areas of software engineering that includes activities like slicing, program debugging, software testing, regression testing, etc.

Slicing is an important technique which has a wide range of applications in software testing. Basically, slicing is a technique for simplifying programs by focusing on selected aspects of semantics. It is method of program analysis which is used to extract a set of statements in a program which is relevant for a particular computation. This set of statements is called a program slice. Various type of slicing strategies exist such as forward slicing, backward slicing, static slicing, dynamic slicing, etc. These different slicing techniques have different application domains such as software maintenance, software optimization, program analysis, information flow control, etc.

1.2 Motivation for our project

Usually testing of the software products is carried out in various levels to identify all defects existing in the software product. However, for most practical systems, even after satisfactorily carrying out the testing process, we cannot guarantee that a software product is error free. This situation is caused by the fact that input data domain of most software products is very large. Hence, it is practically impossible to test the software exhaustively with all the sample test cases. It is quite obvious that not all the lines in the source code contribute to the error at a particular location. We therefore need not consider the whole source code in the testing process and only focus on those areas that are more likely to have caused the error. In order to find these high-risk areas, we need to find the dependencies between the program statements and then slice the program.

We mainly concentrated on Object Oriented Programs; the slicing of object oriented programs is more complicated than slicing procedural programs. We need to consider the features such as classes, inheritance and polymorphism. Though, inheritance and polymorphism are great strengths of OOPs they pose special challenges in slicing.

1.3 Basic concepts

1.3.1 Slicing

Program slicing is a method of program analysis which is used to extract a set of statements in a program which is relevant for a particular computation. This set of statements is called a program slice. It therefore, computes the statements which affect the value of a variable at a particular point in the program. Program slicing was originally introduced by Mark Weiser as "a method for automatically decomposing programs by analyzing their data flow and control flow starting from a subset of a program's behavior, slicing reduces that program to a minimal form that still produces that behavior. The reduced program called a slice is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of the behavior." The input to the program slicing algorithm is the slicing criteria, and the output is program slice. [1]

1.3.2 Slicing criterion

Slicing is always carried out or computed with reference to a slicing criterion. The slicing criterion is represented as $< S, V >$. S is the statement whose slice is to be computed and V is the variable for which we need to compute the slice and that has been used or defined at S [1].

1.3.3 Types of slicing

Program slicing is broadly categorized into the following types

1.3.4 Static slicing

Static slice consists of all the statements in the program that affects the value of a variable at a particular point of interest [6]. A static slice is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point [3]. Given a variable v and a point of interest n, slice will be constructed for v at n. Here an example program to be sliced is given and the variable is p and the slice point is the end of the program.

```
void main()
{
scanf("%d",&n);
s=0;
p=0;
while (n>0)
{
s=s+n
p=p*n;
n=n-l;
}
printf("%d %d",p,s);
}
```

```
Slice obtained:
void main()
{
scanf("%d",&n);
p=0;
while (n>0)
{
p=p*n;
n=n-l;
}
}
```

1.3.5 Dynamic Slicing

Static slice may contain statements which have no influence on the value of the variables of interest for the particular execution in which the anomalous behavior of the program was discovered.

Dynamic slice identify all and only those statements that affect the variables of interest on the particular anomalous execution trace.

In dynamic slicing we have to consider three parameters. First one is the variable, second one is the point of interest within the program and the third one is the sequence of input values for which the program was executed. These together are called as the 'dynamic slicing criterion' [4, 5]. We can say that a dynamic slice for a variable v, at a point n, on an input i can be constructed. Here an example program to be sliced is given. Variable is p, slicing point is end of the program and input is $n = 0$.

```
void main()
{
scanf("%d",&n);
s=0;
p=0;
while (n>0)
{
s=s+n
p=p*n;
n=n-l;
}
printf("%d %d",p,s);
}
```

```
Slice obtained:

void main()

{

p=0;

}
```

1.3.6 Statement Dependencies

We can construct dependencies between statements, showing how they influence each other. We distinguish the following two types of dependencies [7].

1.3.7 Data Dependency

A statement B is data dependent on a statement A if

■ A writes some variable V (or more generally, part of the program state) that is being read by B, and

■ there is at least one path in the control flow graph from A to B in which V is not being written by some other statement.

1.3.8 Control Dependency

A statement B is control dependent on a statement A if $B's$ execution is potentially controlled by A.

Consider the example

```
1 #include<stdio.h>
2 int fib(int n)
3 {
4     int f;
5     int f0=1;
6     int f1=1;
7  while(n>1) {
8     n=n-1;
9     f=f0+f1;
10   f0=f1;
11   f1=f;
12  }
13  return f;
14 }
```

We can use dependencies to check for specific defect patterns and focus on specific subsets of the program being debugged; such a subset is called a slice. The data and control dependencies of the example program are shown in fig 1.1.

1.3.9 Forward Slice

By following all dependencies from a given statement A, we can reaches all statements of which the read variables or execution could ever be influenced by A. This set of statements is called the forward slice [7].

For example in above sample code forward slice for statement 9 ($f0 = 1$) includes {2,6,7,8,9}.

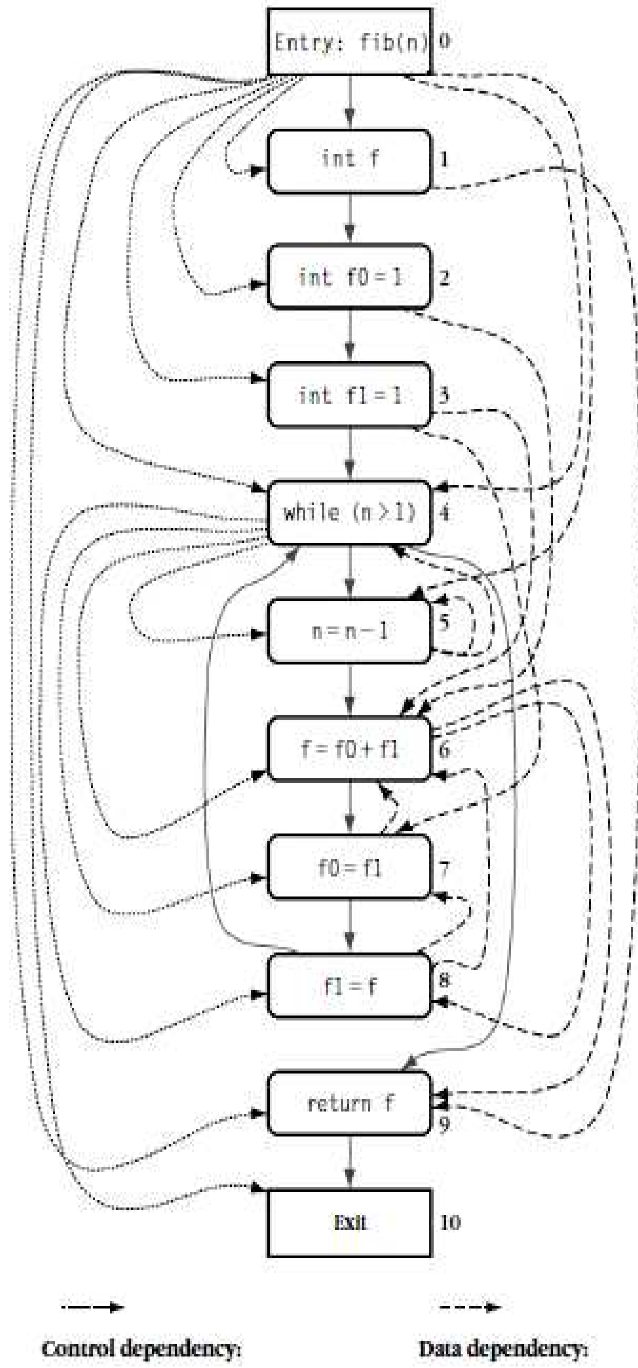


Fig 1.1 The dependency graph of the example program [7].

1.3.10 Backward Slice

By proceeding backward along the dependencies. we can determine all statements that could have influenced any statement.[7].

For example the backward slice of statement 13($return\ f$),includes statements {0,1,2,3,4,5,6,7,8,9}

1.3.11 Control Flow Graph

A Control Flow Graph is a directed graph with a unique entry node START and a unique exit node STOP, where each node is a statement in the program. There is a directed edge from node P to node Q in the control flow graph if control may flow from block P directly to block Q. Edges in a CFG are of two types. An edge is called a T edge, if control flows along that edge when the predicate at the origin evaluate to true and vice versa [1].

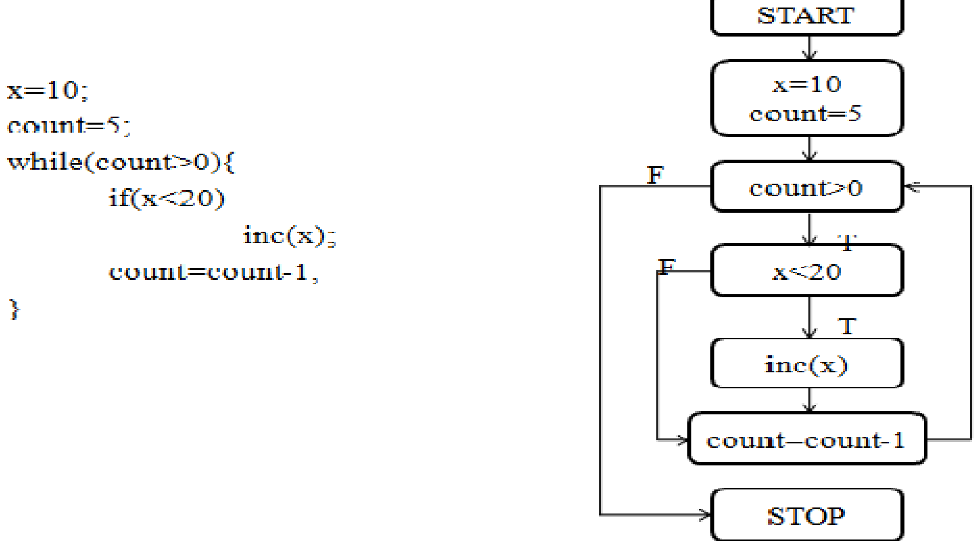


Fig 1.2 Control Flow Graph of the sample Program [1]

1.3.12 Data Dependence Graph

Data dependence graph added data dependence edge to a control flow graph. Data dependence edge exists from node X to Y if the following conditions are satisfied [1]

- Node X defines variable, say V
- Node Y uses the variable V for computation
- Control can flow from X to Y and along the flow path and there should not be any intervening definition of the variable V.

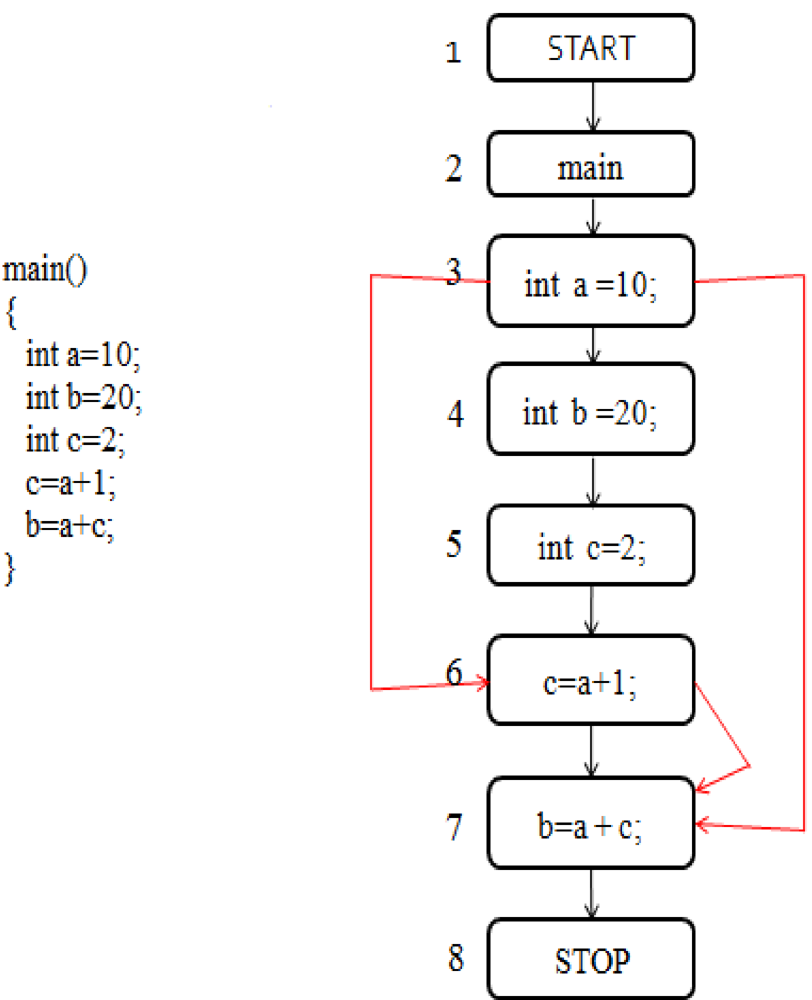


Fig1. 3 Data Dependence Graph of the example program [1]

1.3.13 Program Dependence Graph

The program dependence graph G [2, 13, 14] of a program P is the graph $G = (N, E)$, where each node n belonging to N represents a statement of the program P. The graph contains two kinds of directed edges: control dependence edges and data dependence edges. A control (or data) dependence edges (m, n) indicates that n is control (or data) dependent on m. Note that the PDG of a program P is the union of a pair of graphs: Data dependence graph and control flow graph of P [9].

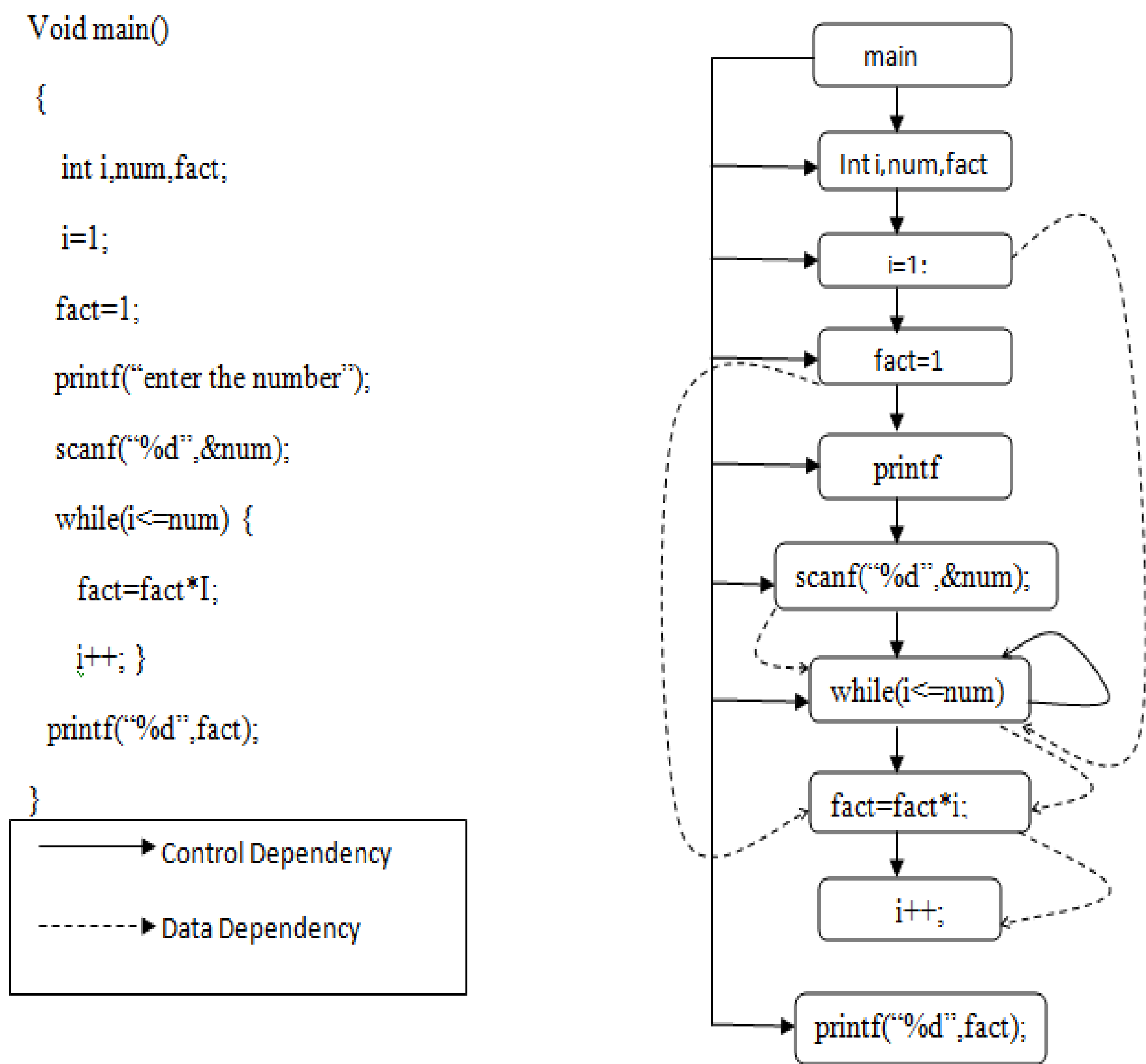


Fig1. 4 Program Dependence Graph of the example program [7]

1.3.14 System Dependence Graph

The PDG cannot handle procedure calls. Horwitz et al [2, 13, 14] introduced the System Dependence Graph (SDG) representation which models the main program together with all associated procedures. SDG is actually a collection of PDGs. For programs without procedure calls, the PDGs and SDGs are similar. For construction of an SDG, first the PDGs of all the procedures are constructed individually and then the SDG is constructed by integrating all the PDGs [9].

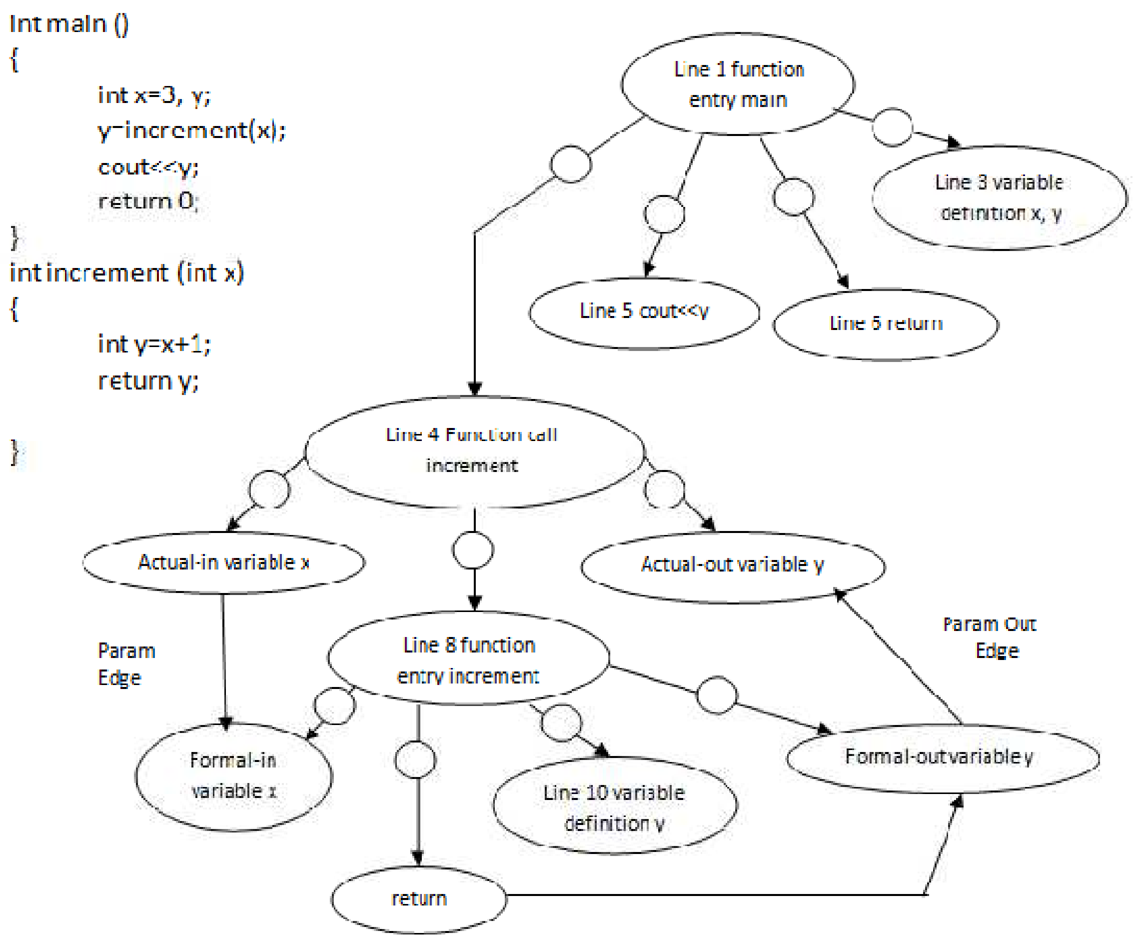


Fig 1.5 System Dependence Graph of the Example Program [9]

1.3.15 Object-Oriented Program Dependence Graph

OPDG adds some new kinds of node and edge to demonstrate object-oriented characteristics such as inheritance and polymorphism [12]. Class header node represents a collection of all the methods and the data members in a class, it also describes the inheritance relationship between classes. Use the member edge to connect the class header node with the method header nodes which belong to the class or inherited from the super class. Use the inheritance edge to connect the class header node with the subclass header nodes and the super class header nodes. Class header node also stores the information of the data members. Method header node is the method entry node, which indicates that class who define the method. There are three kinds of method call in object-oriented program, method calls, class instantiation call, polymorphism call. Simple call edge describes general method call; the call can be determined the called object by static analysis. Instantiation edge describes the instantiation of the class, it calls the class constructor function. Polymorphic call edge

describes the dynamic binding and the polymorphism. The calling to the virtual methods can only be determined at run time. Use of the polymorphic call edge to connect the call node with the class header nodes can ensure that slicing algorithm can find all the possible called methods through the polymorphism calling edges and the class inheritance edges. Polymorphism choice edge connects a virtual method header node in the super class with the virtual method header nodes in the subclasses which have the same specification. Inheritance edges used to describe inheritance. Use the inheritance edge to connect the super class header node with the subclass header node. Membership edges used to describe membership of the class and its methods, data members. Use the membership edge to connect the member header node with the class header node who defines the method. Subclass inherits the methods defined in the super class. Use inheritance edge to connect the method header node that is inherited with the subclass header node [12]. The inheritance edge shows the sub-class's implied membership. Figure 1.6 is an example of OPDG.

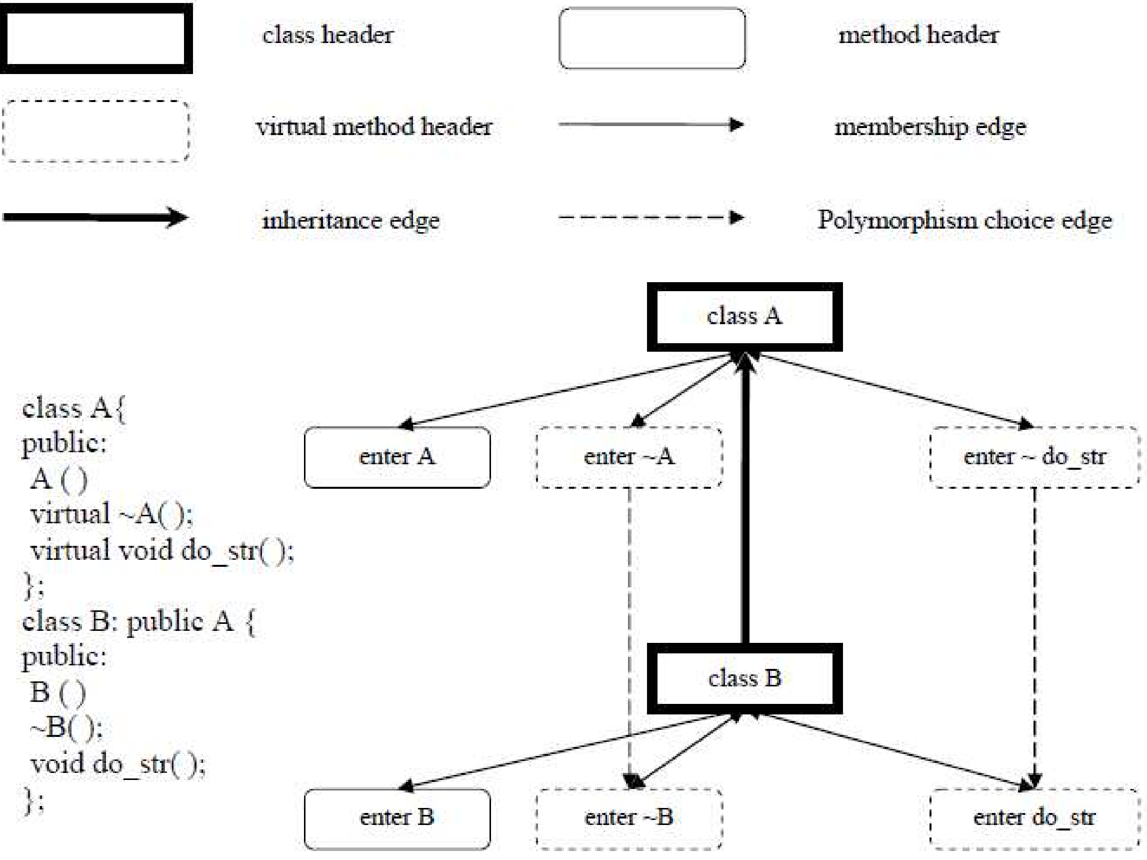


Fig 1.6 Object Oriented Program Dependence Graph of the example Program [12]

1.3.16 Extended System Dependence Graph

The extended system dependence graph is used to represent the programs with object oriented features that include data hiding, inheritance, polymorphism, etc. It is also called as a Class dependence graph (CLDG) [10].

A CLDG captures the control and data dependence relationships that can be determined about a class without the knowledge of calling environments. Each method in a CLDG is represented by a procedure dependence graph. Each method has a method entry vertex that represents the entry into the method. A CLDG also contains a class entry vertex that determines the entry into the class. The class entry vertex is connected to the method entry vertex for each method in the class by a class member edge. Class entry vertices and class member edges let us quickly access the method information when a class is combined with another class or system [10].

In a CLDG, each method entry is expanded by adding formal-in and formal-out vertices. Formal-in vertices are used for each formal parameter that is added and formal-out vertices for each formal reference parameter that is modified by the method. Additionally, formal-in and formal-out vertices are also added for global variables referenced in the method. Since the class's instance variables are accessible to all methods in the class, we treat them as global to methods in the class and we add formal-in and formal-out vertices for all reference variables referenced in the method. However, the exception to this representation for instance variable is that formal-in vertices for the instance variables in the class constructor and formal-out vertices for the instance variables in the class destructor are omitted [10].

1.3.17 Execution Trace

It is a specialized use of logging to record information about a program's execution. An execution trace (or simply a trace) of an object-oriented program is the sequence of methods invoked during the execution [8]. For each method, the trace records the object on which the method is called, the values of the arguments that are passed, the value that is returned, and the trace generated by the body of the method.

1.3.18 Defined-Used Chain

When a statement assigns value to some variable we say the statement defines the variable, and when it references some variable we say it uses the variable. If a variable reference plays role in the computing of the new value of another variable in

the same statement we say the definition is influenced by the use. Note that when a statement uses and defines several variables (multi-definition statement) it may occur that different uses influence a single definition, on the other hand the same use may influence different definitions. For example in the following C language instruction [11].

$$x = y + v, z = v;$$

Use of variables y and v influence definition x, furthermore use v influences definition z too in the same statement [11].

A subpath from node n to node m is called to be definition-clear with respect to variable v, if none of the nodes of the subpath (ignoring n and m) contain definition to v. We say that the definition of variable v in node n reaches another node m, if there exists a definition-clear subpath with respect to variable v from n to m. When the definition of variable v at node n reaches a node m in which the same variable v is referenced, the definition of v in node n and the data-dependent node m is called a definition-use(d-u) pair for v and denoted by (nv, m). The sequence of connected du pairs, in which each adjacent pair corresponds to a du pair, is referred to as a du chain [11].

1.3.19 PSPO

Path Slice Per Object, it is a slice of the given execution trace for the given object such that:-

- The sequence of public methods invoked on the object in the trace is same as the sequence of public methods invoked on the object in the slice.
- Given a method invocation in the slice, the state of all objects accessed by the method is same in both the trace and slice.

1.3.20 GPSPO

Generator Path Slice Per Object, it is the executable program whose execution trace is the PSPO. [26]

Chapter 2

Literature survey

Program Slicing was proposed by Mark Weiser. Program Slicing is to remove the irrelevant statements from the program code. Irrelevant statements are those statements to which the buggy statement is neither data dependent nor control dependent. Weiser found that the output of a particular execution of the program depends on the small portion of the source code and change in the rest of the program does not affect the output of the program. Weiser used Control Flow Graph (CFG) as an intermediate representation for the computation of Program Slicing [15] [16].

CFG can only represent the control dependency between the program statements. Ottenstein proposed a new graphical representation of the program called Program Dependency Graph (PDG) which represented both Control as well as Data dependency. One disadvantage of PDG was that it can only be applied on single-process procedures [17].

Horwitz introduced the System Dependence Graph (SDG) for computing Slice of the inter procedure program with procedure call. Later Horwitz proposed the better algorithm to compute slices of inter procedure program with procedure call. Horwitz proposed two-phase graph reachablity algorithm to compute the precise inter-procedural slice, it divide traversal process of the system dependence graph into two phases [18].

Phase 1: at the slicing criterion, traversal the SDG along with the data dependence edges, control dependence edges, parameter input edges, called edges, summary edges, mark all reachable nodes.

Phase 2: at all the marked nodes in the phase 1, traversal the SDG along with the data dependence edges, control dependence edges, parameter output edges, summary edges, mark all reachable nodes.

Horwitz later proposed a new technique for slicing of programs with many procedure calls, referred as call stack sensitive slicing [21].

2.1 Call Stack-Sensitive Slicing

Consider the example

```
1  void print {char *msg,int val)  {
2    printf("%s %d\n",msg,val);
3    }
4    int get choice(char *ch) {
5      if (strcmp(ch,"sum")==0)
6          return 1;
7       else return 0;
8     }
9    void main(int argc,char *argv[])  {
10   int sum=0;
11  int prod =1;
12  int k=1;
13   int ch=getchoice(argv[1]);
14   if(ch==0)  {
15     while(k<11)  {
16       sum+=k;
17       k++;
18    }
19   print("val:",sum);
20   }
21  else {
22    while(k<11)  {
23     prod *=k;
24     k++;
25    }
26    print("val:",prod);
27   }
28   }
```

Here the point of failure for input sum is line 4. Unfortunately, in this case, the backward slice from line 4 is the entire program because function print is called to print both the sum and the product. Backward slicing of a program statement S involves a backward traversal of the edges of the System Dependence Graph representation of the program. The important difference arises when statement S is in a procedure P other than main. In that case, the full slice follows edges back to all call sites that call P, while the call stack-sensitive slice only follows edges back to the call site that is at the top of the given call stack. If P was reached by a sequence of calls from main, e.g., call P_1, call P_2,. . . , call P_n, call P, the full slice continues to follow edges back to all call sites that call P_n, P_{n-1}, etc., while the call stack-sensitive slice only follows edges back to the call sites in the call stack. In the example program, the

full slice back from line 4 includes both lines 25 and 31 (the two call sites that call function print). The call stack-sensitive slice, given the call stack that contains the call to print made on line 25, only follows edges back to the call site on line 25, and thus, does not include the call to print made on line 31 [21].

Call stack-sensitive slicing can be even more helpful when a single bug causes different test inputs to trigger failures at different points or at the same point but with different sequences of active function calls. For example, if the example program is tested with the command line argument sum and then prod, bad output will be produced at line 25 and then at line 19. If both errors are due to the same bug, the erroneous code must be in the slices back from both lines and thus it can be even more easier for the programmer to locate the bug by displaying the intersection of those two slices: the boxed indexes lines of code in the example program [21] shows the intersection of the two slices.

Although lots of graphical representation of the program has been proposed but generating the graph from program is in itself a very difficult task and then following dependencies to compute slice makes it more tedious. Arpad Beszedes and Tamas Gergely[11] proposed a new method to compute slices based on du chains. Since graphical representation of the program is not required so the method proposed is quite useful for computing slices. Arpad Beszedes and Tamas Gergely [11] uses execution trace for computing the dynamic slices of the program execution trace will contain actions denoted by i^j , where i is the serial number of an instruction in the program, while j is the serial number of an execution step in the execution trace. Arpad Beszedes and Tamas Gergely [11] proposed forward and backward slicing algorithms. The algorithm use execution trace and def-use information of the program statement for computing slices.

i	defined:used
1 read(a)	a:ϕ
2 y=0	y: ϕ
3 x=1	x: ϕ
4 while(a>0)	p:{a}
5 y=x	y:{x,p}
6 x=2	x:{p}

7 a=a-1 a:{a,p}

8 z=y z:{y}

Execution trace for this program with a=1 is ET={1^1, 2^3, 3^3, 4^4, 5^5, 6^6, 7^7, 4^8, 8^9}

2.2 Slicing of object oriented programs

Slicing object-oriented programs presents new challenges which are not encountered in traditional program slicing. To slice an object-oriented program, features such as classes, dynamic binding, encapsulation, inheritance, message passing and polymorphism need to be considered carefully. Although the concepts of inheritance and polymorphism are strengths of object-oriented programming languages, they pose special challenges in program slicing. Due to inheritance and dynamic binding in object-oriented programs, the process of tracing dependencies becomes more complex than that in a procedural program. Larson and Harrold were the first to consider these aspects in their work [20]. Larson and Harrold [20] extended the SDG of Horwitz et al. [22] to represent object-oriented programs. They have constructed Class Dependence Graphs (CLDG) for each class in an object-oriented program. A CLDG captures the control and data dependence relationships that can be determined about a class without knowledge of calling environments. After constructing the CLDG for a complete object oriented program, they have used the two-pass graph reachability algorithm [22] for computing slices. Since Larson and Harrold [20] have computed the static slice, so all most all of the statements in the example program are included in the slice. One limitation of this approach is that the data dependencies obtained using the approach for creating the individual procedure dependence graphs are imprecise: by treating data members declared in a class as if they were global to the methods of that class, the approach fails to consider the fact that in different method invocations, the data members used by the methods might belong to different objects. A second limitation of the approach is that it does not handle cases in which an object is used as a parameter or as a data member of another object.

Krishnaswamy [23] proposed a different approach to slicing object-oriented programs. He used another dependence-based representation called the object oriented program dependency graph (OPDG) to represent the object-oriented programs. The OPDG of an object oriented program represents control flow, data dependencies and control dependencies. The OPDG of an object-oriented program is the union of three

sub graphs: Class Hierarchy Subgraph (CHS), Control Dependence Subgraph (CDS), and Data Dependence Subgraph (DDS).The CHS represents inheritance relationship between classes, and the composition of methods into a class. A CDS represents the static control dependence relationships that exist within and among the different methods of a class. The DDS represents the data dependence relationship among the statements and predicates of the program. Slices can be computed using OPDG as a graph reachability problem. He also computed the polymorphic slices of object-oriented programs based on the OPDG.

The OPDG of an object-oriented program is constructed as the classes are compiled and hence it captures the complete class representations. The main advantage of OPDG representation over other representations is that the representation has to be generated only once during the entire life of the class. It does not need to be changed as long as the class definition remains unchanged.

Steindl [24] has developed a fully operational program slicing tool, Oberon Slicing Tool, for the programming language Oberon-2. It generates state-of-the-art algorithms and applies them to a strongly-typed object-oriented programming language. It extends them to support intermodular slicing of object-oriented programs. Control and data flow analysis considers inheritance, dynamic binding and polymorphism, as well as side-effects of functions, short circuit evaluation of Boolean expressions and aliases due to reference parameters and pointers. The algorithm for alias analysis is fast but effective by taking into account information about the type of variables and the place of their declaration. The result of static program analysis is visualized with active text elements: hypertext links connect the call sites with the possible call destinations; parameter information elements indicate the direction of data flow at calls. Since static program analysis must make conservative assumptions about actual program executions, the sets of possible aliases and call destinations due to dynamic binding are more general than necessary. Steindl [24] has visualized these sets and allowed the programmer to restrict them via user interaction. These restrictions are then used to compute more precise control and data flow information. In this way, the programmer can limit the effects of aliases and dynamic binding and bring in his knowledge about the program into the analysis. The disadvantages of this technique are:

- The layout of the original source code is lost.

- The front-end of the compiler skips all comments, so they are lost and cannot be displayed.
- The front-end of the compiler performs some simple optimizations such as constant folding, transformation of IF statements with constant conditions, replacement of integer multiplication by a power of two by arithmetic shift, etc. These optimizations cannot be undone and the results are presented to the user. This may give insights, but may also confuse.

2.3 Dynamic Slicing of Object-oriented Programs

Agrawal and Horgan [27] were the first to present algorithms for finding dynamic program slices using program dependence graphs. They proposed a dynamic slicing method by marking nodes on a static program dependence graph. The computed slice is not always precise, because some dependencies might not hold in dynamic execution. They also proposed a precise method based on the dynamic dependence graph (DDG) [27]. Zhao [30] extended the DDG of Agrawal and Horgan [27], known as dynamic object-oriented dependence graph (DODG) to represent various dynamic dependencies between statement instances for a particular execution of an object-oriented program. The DODG is an arc-classified diagraph (V, A), where V is the multi-set of flow-graph vertices, and A is the set of arcs representing dynamic control dependencies and data dependencies between vertices. Zhao's construction of DODG is based on dynamic analysis of control flow and data flow of the program, and similar to those for constructing dynamic dependence graphs for procedural programs [25]. Zhao constructed the DODG by creating a new node for each occurrence of a statement in the execution trace, and creating all the dependence edges associated with the occurrence at run-time. Zhao [30] has considered the specific features of object oriented programs such as method calls, inheritance, polymorphism and dynamic binding etc. Zhao has adopted the following concepts for dynamic slicing of object-oriented programs:

- A slicing criterion for an object-oriented program is of the form (s, v, t, i), where s is a statement in the program, v is a variable used at s, and t is an execution trace of the program with input i.

- A dynamic slice of an object-oriented program on a given slicing criterion (s, v, t, i) consists of all statements in the program that actually affected the value of a variable v at statement s.

Based on the DODG, Zhao has used a two-phase algorithm to compute dynamic slices of object-oriented programs. Computation of dynamic slices using the DODG is carried out as a graph-reachability problem. The two phases of the algorithm are:

- Computing a dynamic slice over the DODG of the object-oriented program. (This can be done by using a usual depth-first or breadth-first graph traversal algorithm to traverse the DODG of the program by taking the vertex corresponding to the statement of interest as the start point of traversal.)
- Mapping the slice over the DODG to the source code to obtain a dynamic slice of the program.(This can be done by simply defining a mapping function.)

Consider the example

```
1: class Elevator {
        Public:
2:    Elevator(int 1_top_floor)
3:        { current_floor = 1;
4:          current_direction = UP;
5:          top_floor = 1_top_floor; }
6:     virtual ~Elevator() { }
7:     void up()
8:        { current_direction = UP; }
9:     void down()
10:       { current_direction = DOWN; }
11:    int which_floor()
12:       { return current_floor; }
13:    Direction direction()
14:       { return current_direction; }
15:    virtual void go(int floor)
16:       { if (current_direction = UP )
17:          {   while   (current_floor   !=
floor)
&& (current_floor <= top_floor)
18:             add(current_floor, 1); }
         else
19:              { while (current_floor !=
floor) && (current_floor > 0 )
20:             add(current_floor, −1); }
     };
        Private:
21:    add(int &a, const int &b)
22:       { a = a+b; } ;
        protected:
            int current_floor;
            Direction current_direction;
            int top_floor;
          };
23:   class   AlarmElevator:   public
Elevator
24:       AlarmElevator(int top_floor);
25:       Elevator(top_floor)
26:         {alarm_on = 0; }
27:       void set_alarm()
28:         {alarm_on = 1; }
29:        void reset_alarm()
30:         {alarm_on = 0 }
31:       void go(int floor)
32:         { if (! alarm_on)
33: Elevator :: go(floor)
          };
        protected:
          int alarm_on;
        };
34: main(int argc, char **argv) {
          Elevator *e_ptr;
35:        if (argv[1])
36:           e_ptr = new Elevator(10);
          else
37  e_ptr = new AlarmElevator(10);
38:     e_ptr –> go(3);
39:      cout << "\n currently on floor:"
<<            e_ptr –> which_floor();
```

The lines with line number bold and within square boundary shows the dynamic slice of the example program with respect to the slicing criterion (39, current_floor, t, argv[1] = 3), where t is the execution trace. Execution trace of the example program is shown in fig 7. The graph is the directed graph where the direction of edges shows the execution flow of the program.

The disadvantage of Zhao's approach is that the number of nodes in a DODG is equal to the number of executed statements, which may be unbounded for programs having many loops. Further, Zhao has used trace files to store the execution history which is expensive. The space complexity and the time complexity of this dynamic slicing algorithm are of $O(S)$ and $O(S^2)$, respectively, where S is the length of execution of the program.

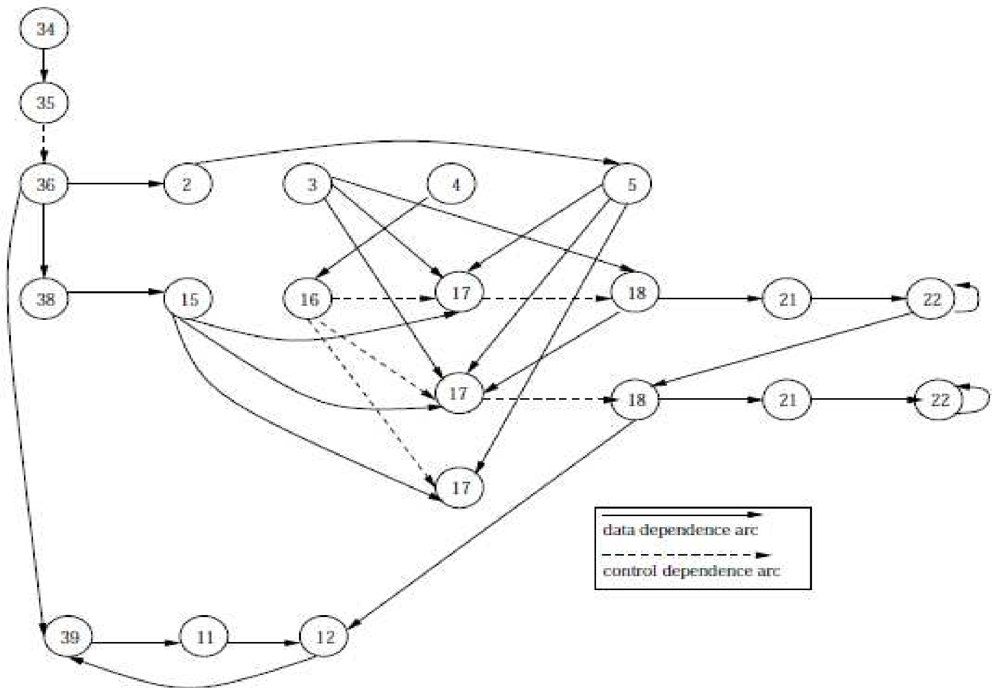


Fig 2.1. An execution trace of the example program on input argv[1] = 3 [30].

Rajib mall proposed An Edge Marking Dynamic Slicing Technique for slicing Object-Oriented Programs [28]. Rajib mall proposes the use of ESDG as the intermediate representation of the program. Edge Marking Dynamic Slicing algorithm is as follows:

The algorithm first mark all the nodes corresponding to the lines executed, and then the corresponding edges. Algorithm then computes the dynamic slice at the mentioned line in slicing criteria by following all the marked edges backwardly.

This proposed algorithm reduces the time and space complexity of the Zhao's approach, this algorithm also does not require saving the execution trace in file.
There are two key challenges in slicing of OOPs:

- a method called on an object could be private and cannot be directly included in the GPSPO,
- a slice may include a method on an object, but the object's definition is subsumed by another method present in the program.

Sudeep Juvekar,Jacob Burnim and Koushik Sen[26] proposed and implemented a technique for computation of slices of OOPs. He compute slices for the particular object, which is called PSPO,and this PSPO is generated through a generator which is called GPSPO[26].
Consider a Java Program

```
1 public class Example1 {
2   public static void main (String[]) {
3     TreeSet set = new TreeSet();
4     LinkedList list = new LinkedList();
5     Integer I;
6     for( int i=0; i<3; i++) {
7       int j = (i * 7) % 11;
8       I = new Integer(j);
9       list.addLast(I);
10      set.add(I);
11    }
12  }
13 }
```

A GPSPO of the execution trace of the above program with respect to the LinkedList object created at line 4 is shown below

```
public class GPSPO1 {
  public static void main (String[] args) {
    LinkedList X1 = new LinkedList();
```

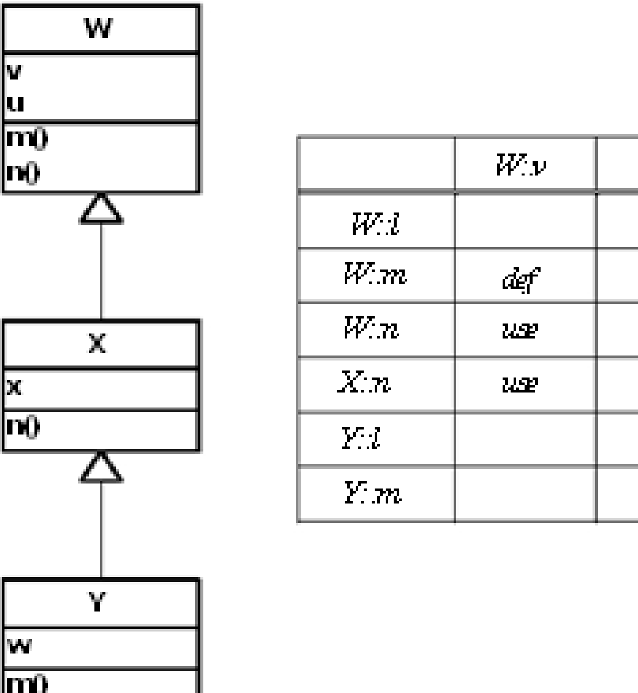
```
Integer X2 = new Integer(0);
X1.addLast(X2);
Integer X3 = new Integer(7);
X1.addLast(X3);
Integer X4 = new Integer(3);
X1.addLast(X4);} }
```

The execution trace of this GPSPO is the PSPO of the execution trace of the original program in with respect to the LinkedList object. Note that the GPSPO completely eliminates the method invocations on the TreeSet object.

Sudeep Juvekar,Jacob Burnim and Koushik Sen [26] solves the above two challenges through their implementation in java using the execution trace as a input.

2.4 Debugging of OOPs in case of Polymorphism

There are some well defined bugs that may present in the program. Many researchers detect the bugs by various techniques. Some possible Bugs are:-



	W::v	W::u	X::x	Y::w
W::l		def		
W::m	def			
W::n	use	use		
X::n	use		def	
Y::l		def		
Y::m				def

Fig 2.2 example class diagram of program having bugs [29]

2.4.1 State Definition Anomaly (SDA)

X extends W, and X overrides some methods. The overriding methods in X fail to define some variables that the overridden methods in W defined. $W::m$ ( ) defines v and $W::n()$ uses v. $X::n$ ( ) uses. $Y::m$ ( ) does not define .For an object of type Y, a data flow anomaly exists[29].

2.4.2 State Definition Inconsistency (SDIH)

Overriding a variable, possibly accidentally. If the descendant's version of the variable is defined, the ancestor's version may not be. Y overrides $W's$ version of v. $Y::m()$ defines $Y::v$. $X::n()$ uses v. For an object of type Y, a data flow anomaly exists and results in a fault if $m()$ is called, then $n()$[29].

2.4.3 State Defined Incorrectly (SDI)

Overriding a method $m()$ that defines a variable v. The overriding method may define v incorrectly. $W::n()$ defines $v.X::n()$ also defines v, but incorrectly.For an object of type X, a behavioral problem occurs if $W::m()$ uses v and assumes it has a value as given in $W::n()$.

All these bugs are detected by the researchers but no one actually debugs these bugs.

Chapter 3

Problem Statement

3.1 Gap analysis

The brief comparison of the proposed methodology with the existing work is given in form of following table 3.1. The comparative study below helps to find out certain shortcomings in the existing methodologies with comparison of the proposed work.

Slicing of object oriented programs using dependence graphs	The past approaches compute the slices for OOPs using various dependence graphs such as object oriented data dependence graph [12]. Computation of slices using these dependence graphs is very difficult, since we first generate the intermediate representation of the program i.e. the dependence graphs, and then compute the slices by following the dependencies. The proposed approach computes the slices for OOPs by generating the d-u chains of the variables and object used in the program.
Time and space complexity of computing slices	The time and space complexity of the algorithms computing the slices using the dependence graphs is much more, so that they are not practically applicable to large programs. Since the proposed approach is not using any dependence graphs, the space complexity is much less and time complexity is slightly lesser.
Challenges for computing slices of OOPs	The challenges for computing slices of OOPs include polymorphism inheritance and constructor. The past approaches solve the constructor and inheritance challenges like private constructor and many more [26, 28, 11]. The proposed approach handles all the possible cases of polymorphism and solves all challenges of polymorphism.
Run time behavior of objects in case of polymorphism	It is very difficult to predict the run time behavior of pointer objects. The past approaches predict the run time behavior of the objects by recording execution

	trace or by marking edges in the dependence graphs [28]. The proposed approach computes the dynamic slices of the program without recording any execution trace.
Debugging of programs in case of polymorphism	There are some well defined possible bugs in the programs having polymorphic behavior. The past approaches detect these bugs by various methods. The proposed approach detects and debugs these bugs.

Table 1: Gap Analysis between the proposed approach and existing approaches

The gaps between the proposed approach and existing approaches are the motivation factors of proposed work.

3.2 Concise Problem Statement

From the above explained gaps our work consist of the following activities

- Computation of dynamic slices of the OOPs especially in case of polymorphism.
- Detection of possible polymorphic bugs in the program.
- Generation of suggestion messages to debug the bugs.

3.3 Justification

The proposed approach not only computes the slices of the program but also detects and debugs the possible bugs in the program. The proposed approach need not require generating any complex dependence graph instead, d-u chains, can be used as intermediate representation of the program, so the proposed approach reduces the time and space complexity of computing slices. The gap analysis in the previous section shows that the gaps in the existing work are removed by this approach. As per existing literature there is no such approach, which computes the dynamic slices of OOPs using d-u chains.

Chapter 4

Proposed Methodology

Debugging is the necessary phase of software development, lots of effort has been deployed to detect the bug and debug the program. Program slicing aids the programmer and reduces the debugging effort. The proposed approach computes the dynamic slicing of OOPs, detecting the bug and also reduces the time required to compute the slices.

4.1 System Overview

The system developed is called "Software Debugger". The main works performed by the system is as follows:-

- Compute the dynamic slices of the OOPs.
- Detect the possible bugs in case of program with polymorphism behavior.
- Generate suggestion messages which may remove the bug.

First the debugger compute the slices of the program after that if the program is having polymorphic behavior the system detects the possible bugs and generate the suggestion messages.

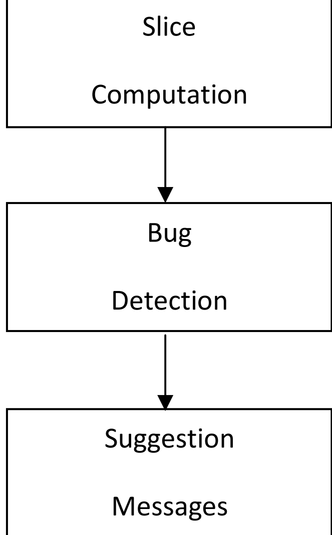


Fig 4.1 Workflow of the proposed approach

4.2 Experimental Setup

The software debugger is developed in c#.net. The input program is in C++ language. The structure of the system is shown in fig 4.2. The key feature of the system lies in slice computation which uses d-u chains and other data structure.

4.2.1 Defined-Used info

This data structure contains the information about all variable and objects, defined and used at each line in main function.

d-u info
string defined; string used;

4.2.2 ClassInfo

This data structure contains information about all the functions defined in the class, their arguments and also whether they are virtual or not.

ClassInfo
string ClassName; string MethodName; string MethodNature;

4.2.3 ObjectInfo

This data structure contains information about all the objects defined in the program, it also have the information about whether the object is pointer object.

ObjectInfo
string ClassName; string ObjectName; string ObjectNature;

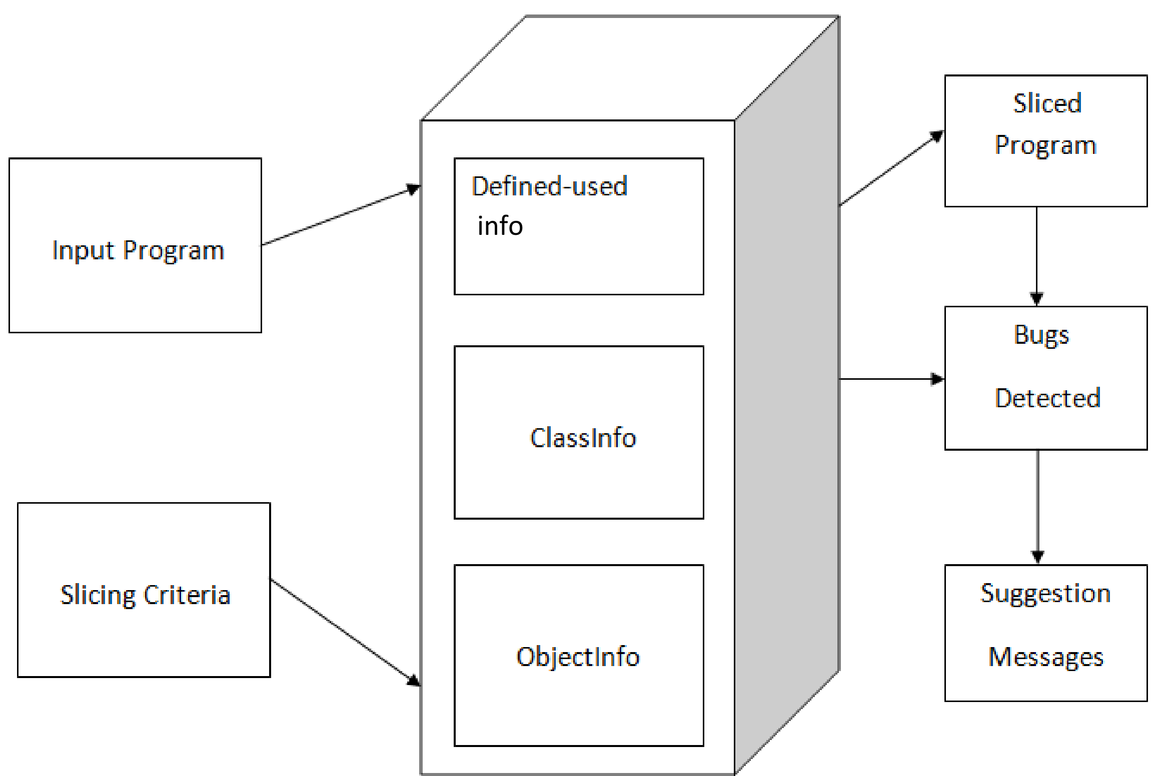


Fig 4.2 Overview of "Software Debugger"

4.3 Proposed Algorithm for Slice Computation

Begin:

$for\ \forall\ a \in\ duInfo$

$if\ (a(d) = O\ ||a(u) = O)$

$S = S \cup a(l)$

$endfor$

$for\ \forall\ b \in\ ObjInfo$

$if\ (b = O)$

$S = S \cup a(l)$

$Include_class_structure(b(c))$

$endfor$

$for\ \forall\ c \in\ duInfo$

$if(c(u)\ is\ method\ call\ \&\&\ e(l)\ is\ in\ S)$

$CP = check\ whether\ the\ object\ used\ is\ \ Pointer$

$CN = find\ class\ name\ of\ object\ used$

$if(!CP \mid\mid (CP \&\& address\ assigned\ to\ the\ object\ \ is\ of\ its\ own\ class))$

$for\ \ \forall\ f\ \epsilon\ \ methodlist$

$if(f(c) = CN \&\& f(d) = c(u))$

$S = S \cup \{all\ lines\ of\ this\ method\}$

$endfor$

$else\ \ if(function\ is\ defined\ in\ super\ class\ \ and\ is\ non\ virtual)$

$S\ \ = S \cup \{all\ lines\ of\ this\ method\ in\ superclass\}$

$else\ if(\ sub\ class\ does\ not\ have\ definition\ of\ that\ method)$

$S = S \cup \{all\ lines\ of\ this\ method\ in\ superclass\}$

$else$

$S = S \cup \{all\ lines\ of\ this\ method\ in\ subclass\}$

$endfor$

$\boldsymbol{Include_class_structure}(\boldsymbol{str})$

$for\ \ \forall\ c1\ \epsilon\ \ clslist$

$if(c1(d) = str)$

$S = S \cup c1(l), line\ numbers\ of\ opening\ and\ closing\ brace\ of\ class\ c1(d)$

$if(c1(next)! = NULL\ in\ IC)$

$Include_class_structure(c1(next))$

Chapter 5

Experimental Results

Software testing and debugging is the most time consuming phase of the software development. Therefore, there is need of technology to support the debugging phase. This approach helps the programmer finding bugs in the program easily and faster. To support the proposed approach a GUI tool is developed in c#.net.

5.1 Slice Computation

The program is browsed and slicing criteria is entered, the input program is:-

```
1. #include <iostream>
2. using namespace std;
3. class CPolygon {
4.   protected:
5.    int width, height;
6.  public:
7.    void set_values(int a,int b){
8.        width=a;
9.        height=b;}
10.    virtual int area(){
11.       return(0);} };
12. class CRectangle: public CPolygon
13. {
14.   public:
15.     int area(){
16.       int area;
17.       area=width*height;
18.       return(area); } };
19. class CTriangle: public CPolygon {
20.   public:
21.     int area(){
22.       int area;
23.       area=width*height/2;
24.       return(area);}};
25. int main()
26. {
27.   CRectangle rect;
28.   CTriangle trgl;
29.  CPolygon poly;
30.  CPolygon *ppoly1;
31.  ppoly1=&rect;
32.  CPolygon *ppoly2;
33.  ppoly2=&trgl;
34.  CPolygon *ppoly3;
35.  ppoly3=&poly;
36.  ppoly1->set_values(4,5);
37.  ppoly2->set_values(4,5);
38. ppoly3->set_values(4,5);
39.  ppoly1->area();
```

40. ppoly2->area(); 42. return 0;

41. ppoly3->area(); 43. }

If the slicing criteria is wrong, then debugger will generate the warning message, otherwise the slice is computed, in fig 5.1 user entered the wrong slicing criteria, and system generate a warning message. At every polymorphic function call, the debugger will tell the user about which function is actually called and ask whether it is correct. In fig 5.2 the debugger is confirming the validity of polymorphic call.

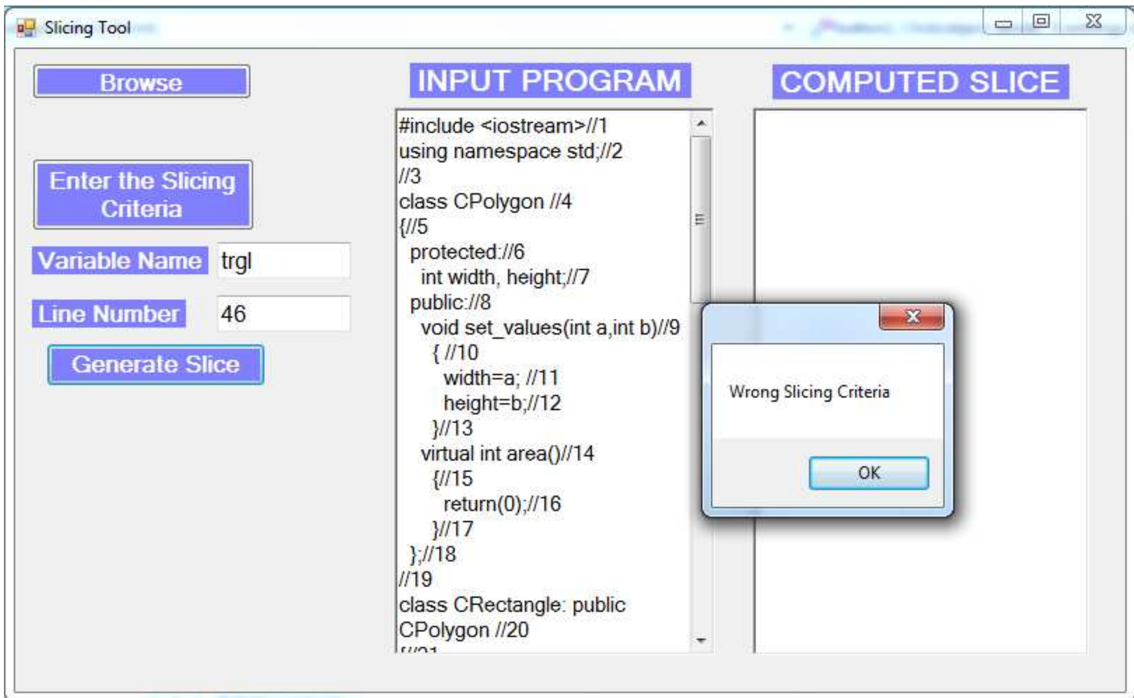


Fig 5.1 Debugger generating the warning message

Here user can analyze the run time polymorphic call, if user answered in “yes”, then system will continue computing the slice without further warning, and if user come to know that this is not the function which is actually required to called here then he will answer in “No”, then debugger will find the bug and display that bugged line in red Fig 5.3 shows the computed slice for the input program with the bugged lines shown in red.

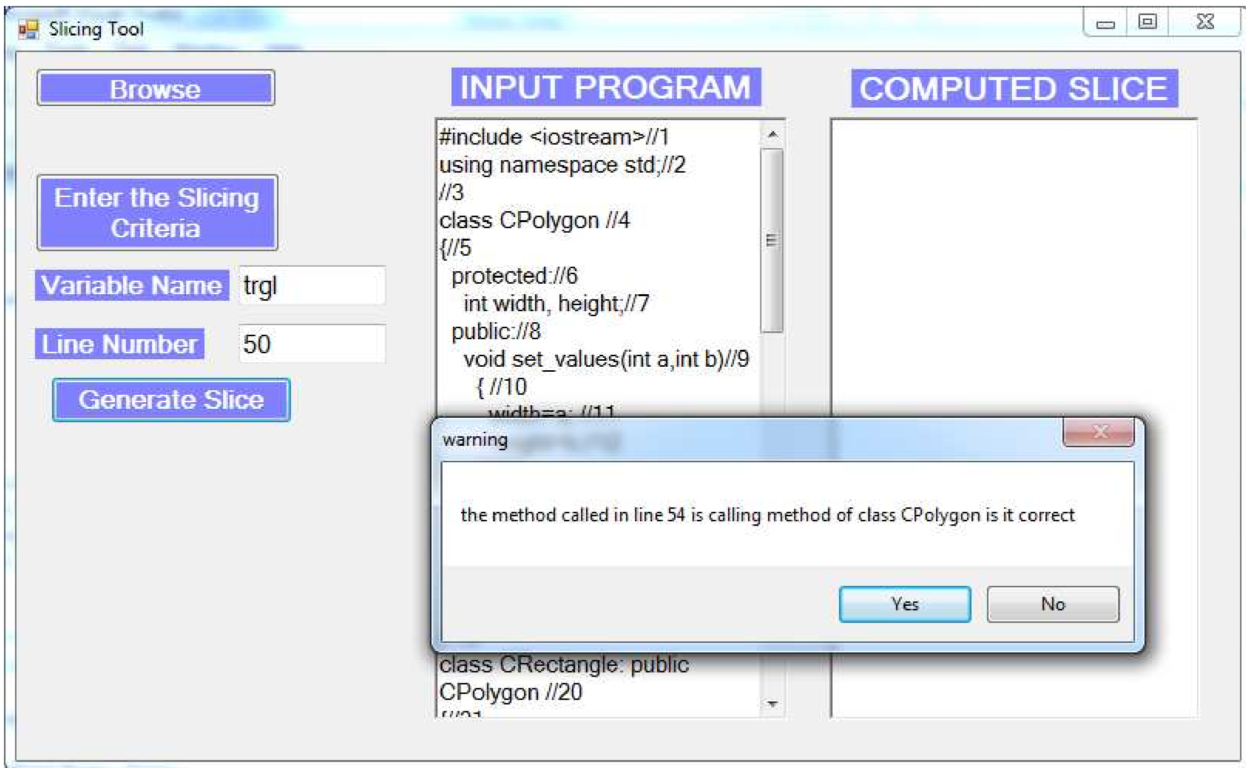


Fig 5.2 Debugger asking the user whether the polymorphic call is correct

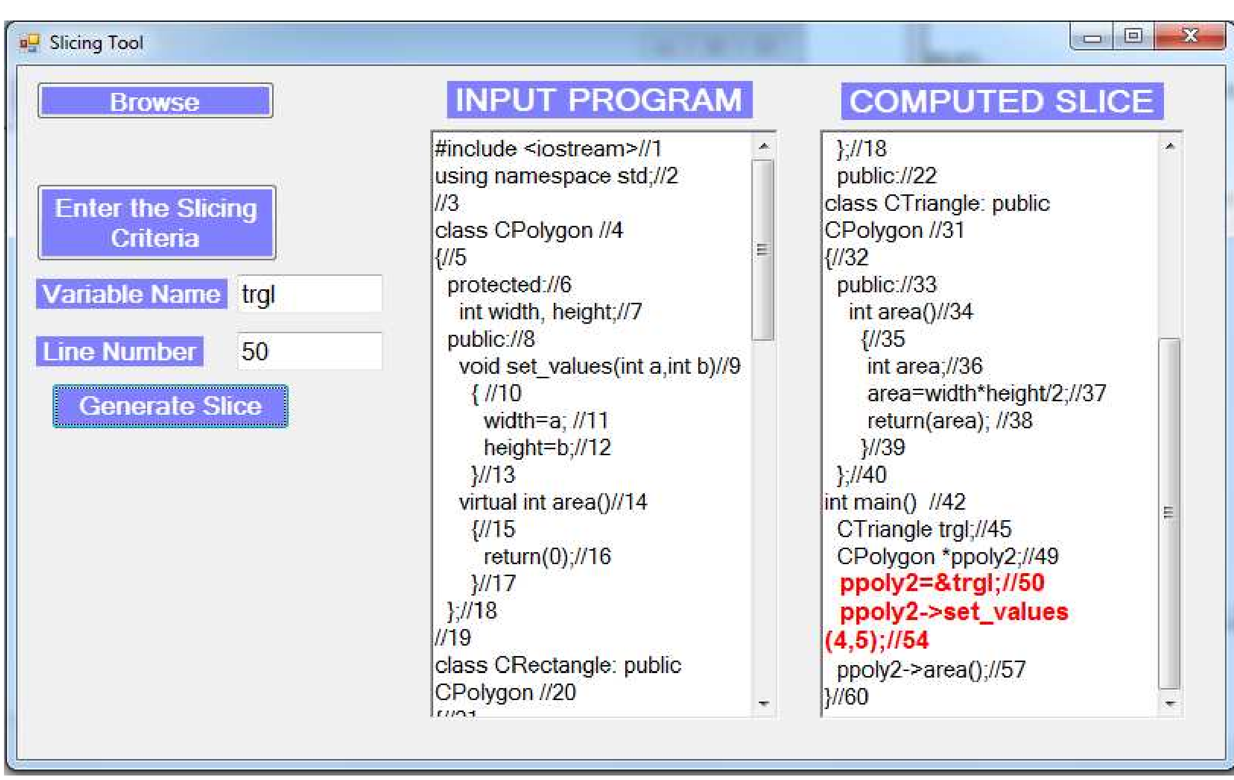


Fig 5.3 Debugger showing the bugged lines in red

5.2 Slice Computation of bugged Program

Whenever Debugger detects the bug in the input program debugger generate some suggestion messages which can help the programmer to remove those bugs.

5.2.1 Input Program with Bug State Definition Anomaly (SDA)

Here in fig 5.4a Debugger generates the message showing the details of the bug present in the input program.fig 5.4b shows the suggestion messages generated by the Debugger.

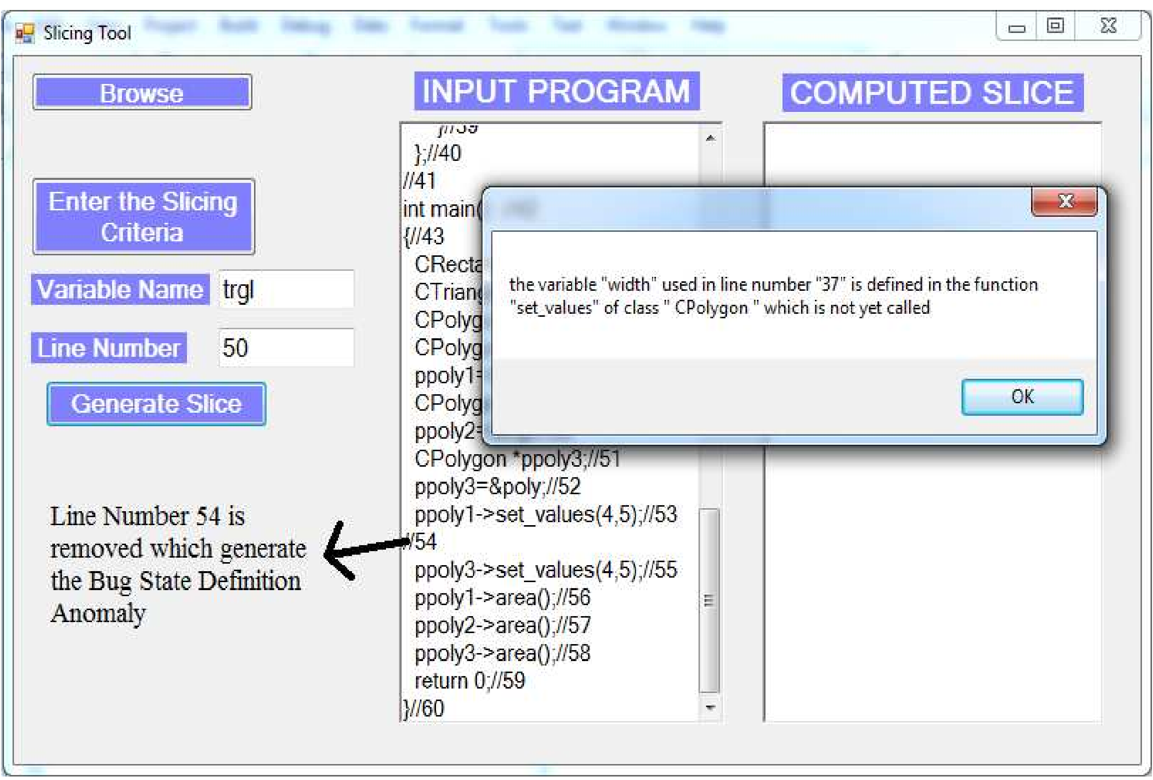


Fig 5.4a Debugger showing details of the bug SDA present in the input program

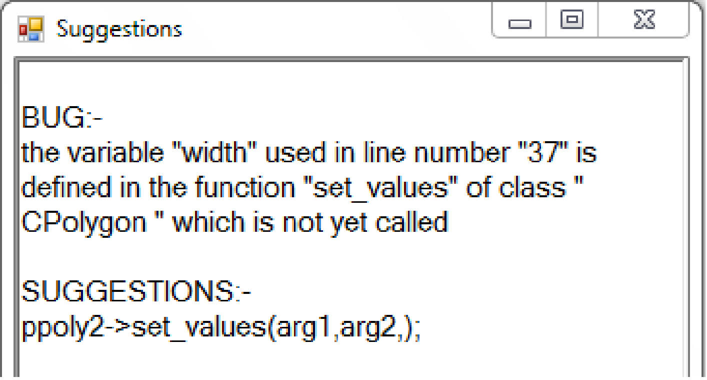


Fig 5.4b Suggestion messages for bug SDA

5.2.2 Input program with Bug State Definition Inconsistency (SDIH)

Here in Fig 5.5a Debugger displays the details of the bug present in the input program, and also conforming whether it is bug or the user willingly want to declare overridden variable locally.

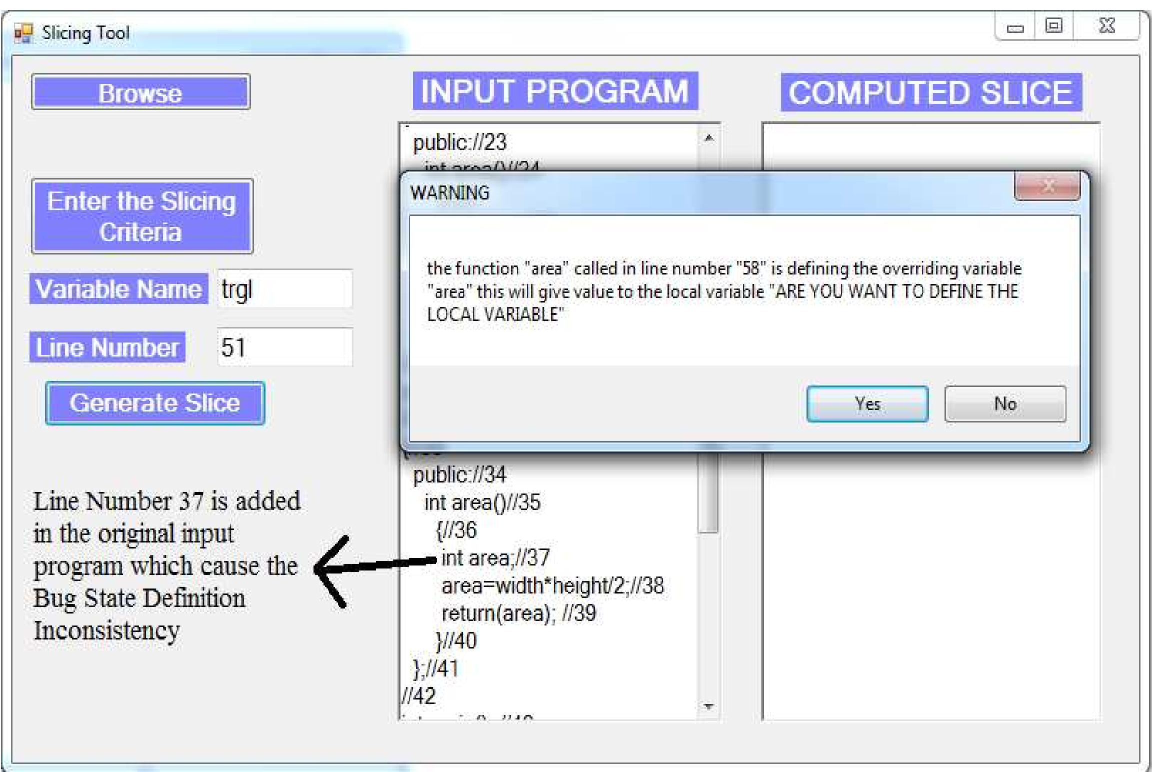


Fig 5.5a Debugger showing details of the bug SDIH present in the input program

Fig 5.5b shows the suggestion messages generated by the debugger which may help in removing the state definition inconsistency.

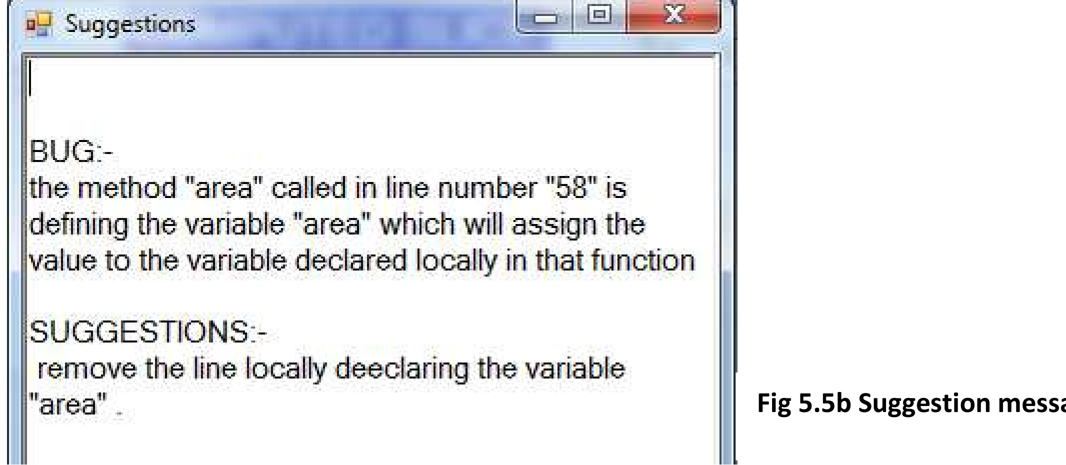


Fig 5.5b Suggestion messages for bug SDIH

5.2.3 Input program with Bug State Defined Incorrectly (SDI)

Here in Fig 5.6a Debugger displays the details of the bug present in the input program, and also confirms whether it is bug or the user willingly want to define overridden variable differently in both classes.

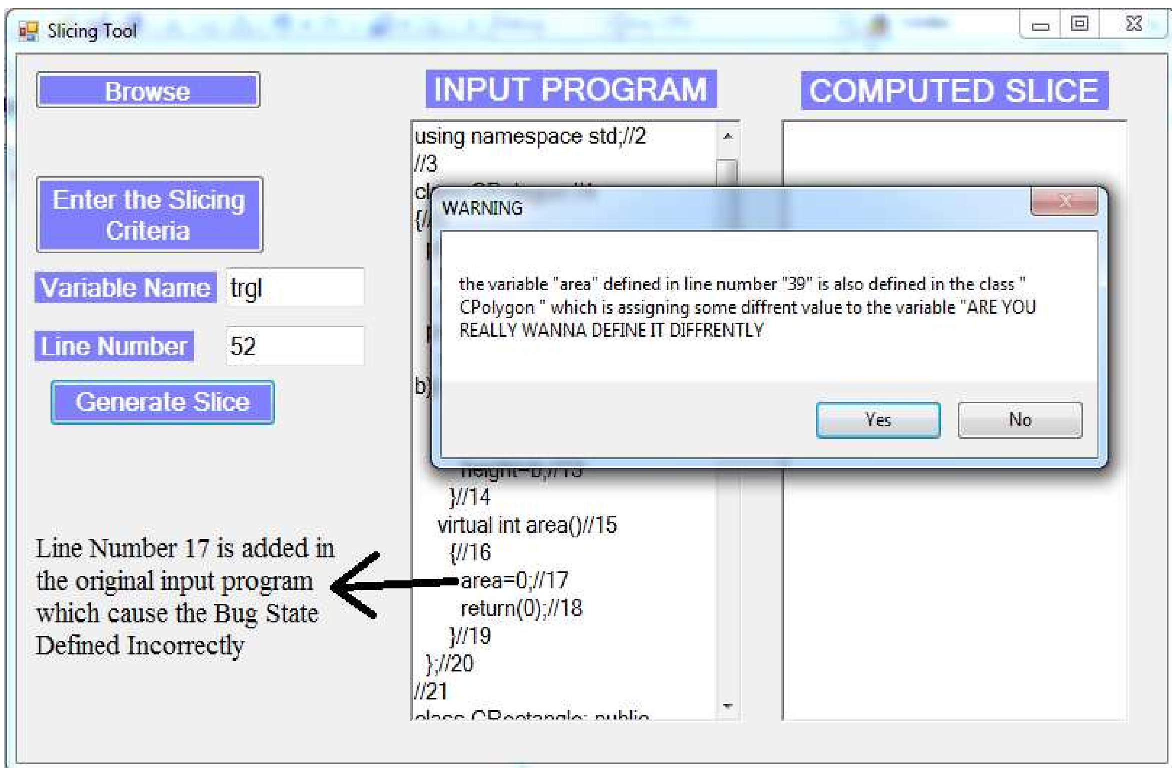


Fig 5.6a Debugger showing details of the bug SDI present in the input program

Fig 5.5b shows the suggestion messages generated by the debugger which may help in removing the bug

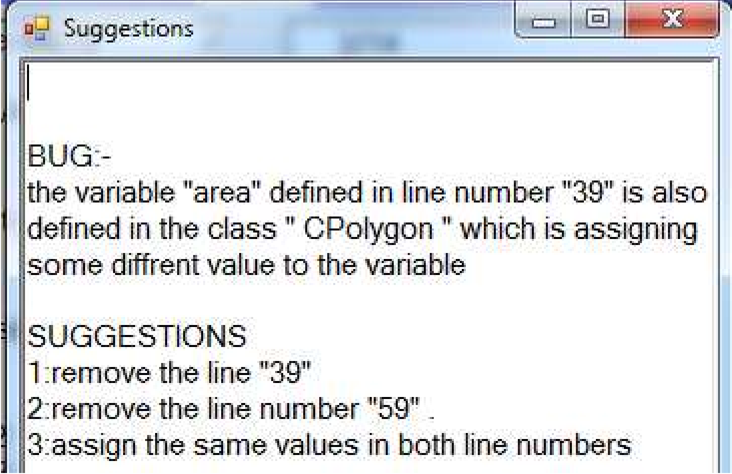


Fig 5.6b Suggestion messages for bug SDI

5.2.4 Input program with Parameter mismatch Bug

It may be the case that user passed the wrong arguments at run time which can result in the parameter mismatch bug, Fig 5.7a presents one such example.

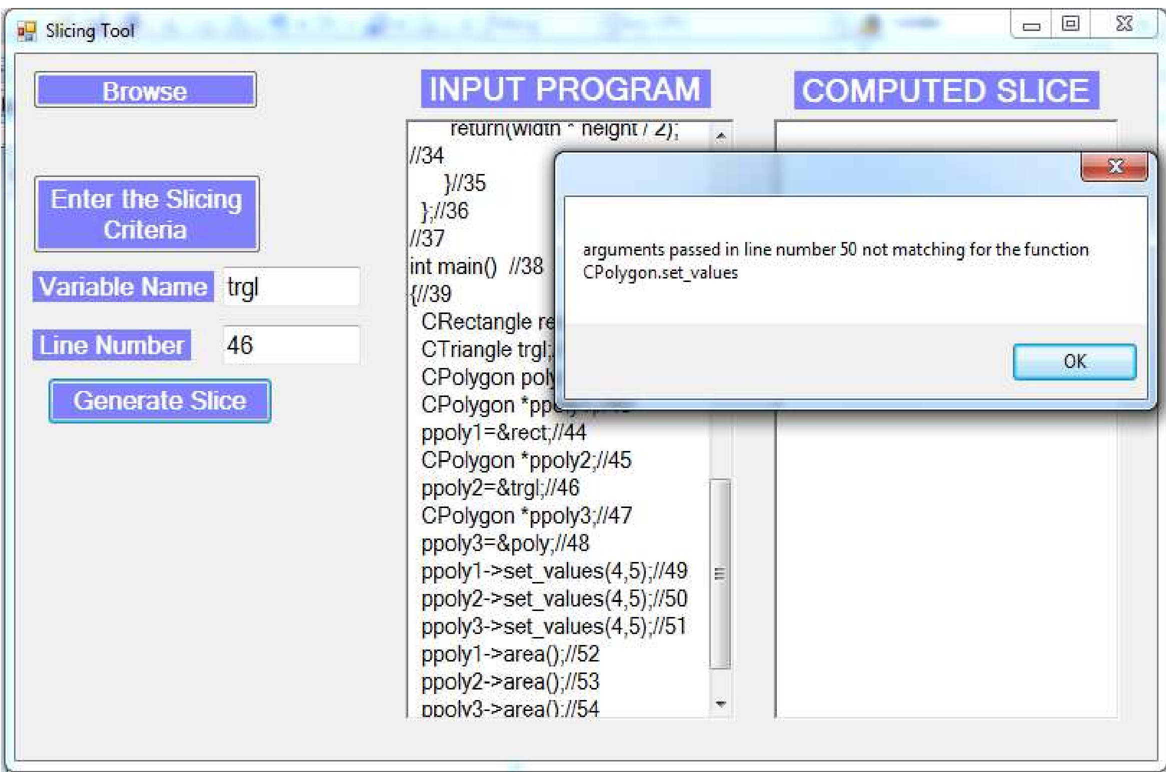


Fig 5.7a Debugger showing details of the bug Parameter Mismatch

Fig 5.7b shows the suggestion messages generated by the debugger which may help in removing the bug

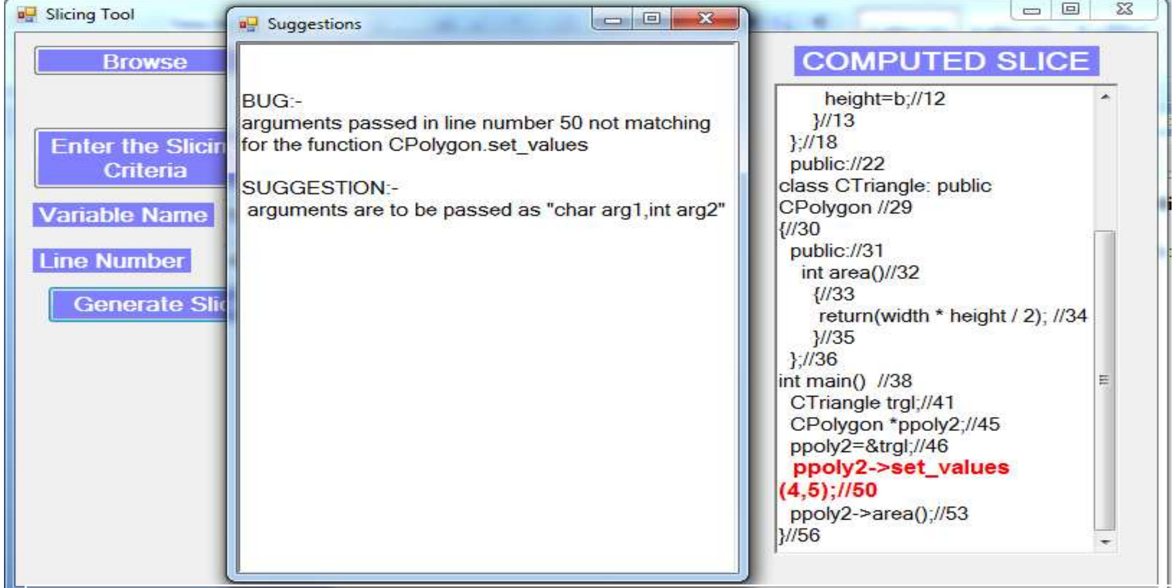


Fig 5.7b Suggestion messages for bug and the bugged lines in red

Chapter 5

Conclusion and Future Scope

Software testing and debugging is the most expensive and time consuming task of the software development. The proposed method computes the dynamic slicing of the object oriented programs which can reduce the time consumed by debugging. The main contribution of this work is computation of dynamic slices of OOPs, detecting Bugs and generating suggestion messages. The major contribution of this work is given below:-

5.1 Conclusion

- The past approaches use various dependence graphs for the intermediate representation of the program. These dependence graphs are complex to generate, the proposed approach generate d-u chains as the intermediate representation of program. The space complexity of the proposed data structure is lesser than the dependence graphs.
- The past approaches identify and detect bugs present in the program but no one actually locates and debugs the bugs. The proposed approach detects and locates the bugged line and also generates the suggestion messages which can actually debug the program. As the tool automatically detects the bugs present in the program, it reduces the effort required for locating the bugs

5.2 Future Scope

- This technique further can be extended to solve more challenges of OOPs like private constructor and destructor etc.
- Further work can be done in the direction of detecting more possible bugs and generating suggestion messages which can help the programmer to debug the program faster.
- This approach can further be extended for other object oriented languages like java.

References

[1] C. Steindl, “Program Slicing for Object Oriented Programming Languages”, PhD thesis, Johannes Kepler University, Linz, 1999

[2] J. Ferrante, K. J. Ottenstein, J. D. Warren, “The Program Dependence Graph and its use in Optimization”, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July 1987

[3] M. Youxin, W. Dafa, D. Junwei, “Research on Framework of Test Case Generation of Web Applications Based on Z Specification”, Proceedings of International Forum on Information Technology and Applications, pp. 555-558, May 2009

[4] P. Samuel, R. Mall, “A Novel Test Case Design Technique Using Dynamic Slicing of UML Diagrams”, e-Informatica Software Engineering Journal, vol 2, no I, 2008

[5] B.Korel, “Computation of Dynamic Program Slices for Unstructured Programs”, IEEE Transactions on Software Engineering, vol. 23, no.1,pp. 17-34, January 1997

[6] P. Samuel, A. Surendran, “Forward Slicing Algorithm based test data generation”,IEEE Transactions on Software Engg.vol 10 no. 978-1-4244-5540-9/10,march 2010

[7] A. Zeller,” Tracking Dependencies ” in Why Program Fails,2th edition, Elsevier Inc. , 2009

[8] S. Juvekar, J. Burnim, K. Sen, “Path Slicing per Object for Better Testing Debugging and Usage Discovery”, Technical Report No. UCB/EECS-2009-132,2009

[9] K. L. Kumawat, “Prioritization of Program Elements based on their Testing Requirements”, Btech thesis, National Institute of Technology, Rourkela, May, 2009

[10] L. Larsen, M. J. Harrold, “Slicing Object-Oriented Software”, Proceedings of the 18th international conference on Software Engineering, Pages 495-505, 1996.

[11] A Beszedes, T. Gergely and T. Gyimothy, “Graph-Less DynamicDependence Based Dynamic Slicing Algorithms”, In Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM’06),2006

[12] B. Panda and V. Prakash, “Slicing of Object Oriented Programs”, Btech Thesis,National Institute of technology,Rourkela,may 2010.

[13] T. Ball, “The Use of Control Flow and Control Dependence in Software Tools”, PhD thesis, Computer Science Department, University of Wisconsin-Madison, 1993

[14] Y. Song and D. Huynh, “Forward Dynamic Object-Oriented Slicing”, Application Specific Systems and Software Engineering and Technology (ASSET“99), IEEE CS Press, 1999

[15] M. Weiser, “Program Slicing”, IEEE Trans. Software Engineering, vol 16(5), pp 498-509,1984

[16] M. Weiser, “Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method”, Ph.D thesis, The University of Michigan, Ann Arbor, Michigan, 1979

[17] K. J. Ottenstein, L.M. Ottenstein, “The program dependence graph in a software development environment”, ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp.177-184, April 1984

[18] S. Horwitz, T. Reps and D.Binkley, “Interprocedural slicing using dependence graphs”, ACM Transaction on Programming Languages and System, Volume:12, Issue: 1, pp.26-60, Jan 1990

[20] L. D. Larson and M. J. Harrold, “Slicing object oriented software”, In Proceedings of the 18th International Conference on Software Engineering, German, March 1996

[21] S. Horwitz, B. Libit and M. Polishchuk, “Better Debugging via Output Tracing and Callstack-Sensitive Slicing”, IEEE Transactions on Software Engineering, Vol. 36, JANUARY/FEBRUARY 2010.

[22] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural slicing using dependence graphs". ACM Transactions on Programming Languages and Systems, vol no 12, pp. 26 – 61, 1990

[23] A. Krishnaswamy, "Program slicing: An application of program dependency graphs", Technical Report, Department of Computer Science, Clemson University, August 1994

[24] C. Steindl, 'Program slicing for object-oriented programming languages", PhD Thesis, Johannes Kepler University Linz, 1999

[25] H. Agrawal, R. A. DeMillo and E. H. Spafford, "Dynamic slicing in the presence of unconstrained pointers", In Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification (TAV4), pages 60 – 73, 1991

[26] S. Juvekar, J. Burnim, K. Sen, "Path Slicing per Object for Better Testing Debugging and Usage Discovery", Technical Report No. UCB/EECS-2009-132,2009.

[27] H. Agrawal and J. Horgan, 'Dynamic program slicing", In Proceedings of the ACM SIGPLAN'90 Conference on Programmimg Lanuages Design and Implementation,SIGPLAN Notices, Analysis and Verification, volume 25, pages 246 – 256, White Plains, NewYork, 1990

[28] D.P. Mohapatra, R Mall, R Kumar, "Computer Software and Applications", Conference, COMPSAC 2004, Proceedings of the 28th Annual International, vol 1 pages 60-65,2004

[29]R. T. Alexander, "Testing the Polymorphic Relationships of Object-oriented Programs", Dissertation, George Mason University, 2001

[30] J. Zhao, "Dynamic slicing of object-oriented programs", Technical report, Information Processing Society of Japan, May 1998

List of Papers Presented

[1] Paritosh Jain, Mr. Karun Verma and Dr. Rajesh Bhatia, “A Novel approach for slicing of object oriented programs”, In ACM SigSoft Software Engineering Notes, Sept Issue 2011 (Communicated)