

ELISSAR : A Case Tool for Testing and Regression Testing

By
Diana M. Nasreddine

June 1997

ELISSAR : A Case Tool for Testing and Regression Testing

RT
234

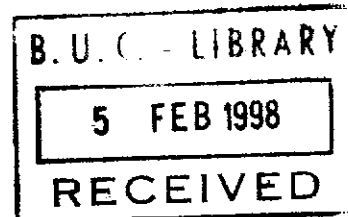
By
Diana M. Nasreddine
B.Sc., Lebanese American University

PROJECT

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
At the Lebanese American University
June 1997

[REDACTED]
Dr. Nashat Mansour (Advisor)
Assistant Professor of Computer Science
Lebanese American University

[REDACTED]
Dr. Issam Moughrabi
Assistant Professor of Computer Science
Lebanese American University



G. Pt.

ABSTRACT

Software testing is done within the implementation phase of the Software Development Life Cycle (SDLC) in order to provide confidence about the correctness of the software. Regression testing is done in the maintenance phase to verify that the modifications made have not caused unintended adverse side effects and that the modified system still meets the requirements. Testing and Regression testing are significant and costly parts of SDLC. Therefore, Computer Aided Software Engineering (CASE) tools are needed by software engineers to assist them for performing these activities. ELISSAR is such a CASE tool developed at LAU which takes a procedural program as an input and provides the following functionality :

- (a) It generates and displays control flow graph
- (b) It generates and displays def-use graph
- (c) It assists in dataflow testing
- (d) It provides test coverage information
- (e) It includes a few regression testing algorithms that suit different user requirements.

To My Dear Father and Mother

Acknowledgements

Developing this work would have been impossible without the support of many great people to whom I would like to express my sincere appreciation.

First of all, I would like to thank Professor Nashat Mansour, My advisor for his guidance, encouragement and support. Also I would like to thank my second reader Professor Issam Moughrabi for accepting to be on my committee. I greatly thank the chairman of the Natural Science Division at LAU, Dr. Ahmad Kabbani and the secretary of he Natural Science Division at LAU, Mrs. Hanan Naccach for the support and encouragement they offered.

Also I would like to thank all my friends, The A.C.C supervisor Mr. Tarek Dana, and the assistant A.C.C supervisor Mr. Ali Aleywan for their help.

Finally, special thanks go to my family for supporting and encouraging me in all the study period and for taking good care of me. Especially I would like to thank my brother Najib for all the helpful comments and discussions.

Table of Contents

Chapter 1	Introduction.....	1
Chapter 2	Tool architecture.....	6
	1. Tool Structure Chart.....	6
	2. Global Data Structures and Variables Declarations.....	7
Chapter 3	User Interface.....	9
	1. Menu Tree.....	9
	2. Menus.....	10
Chapter 4	Program Flow Graphs.....	33
	1. Control Flow Graphs.....	33
	2. Def-use graph.....	36
	3. Implementation of the Control Flow and the Def-use Graphs...	38
Chapter 5	Testing.....	46
	1. Data Flow Testing Criteria.....	46
	2. ASSET.....	50
	3. Implementation.....	53
Chapter 6	Experimental Results.....	64
Chapter 7	User Guidelines.....	77
	1. Syntax Limitations.....	77
	2. File Naming.....	77
Chapter 8	Conclusion.....	81
References.....		R-1

List of Figures

Chapter 2	Figure 2.1	The Structure Chart of ELISSAR	6
Chapter 3	Figure 3.1	The Menu Tree of ELISSAR	9
Chapter 4	Figure 4.1	An example of a C program	34
	Figure 4.2	The Control Flow Graph of Figure 4.1	35
	Figure 4.3	(a) The Tabular Form of the Def-Use Graph of Figure 4.1	36
		(b) The Graphical Form of the Def-Use Graph of Figure 4.1	37
	Figure 4.4	The Structure Chart of the Ctrllduse Module	39
Chapter 5	Figure 5.1	The structure Chart and Semantics of the Pascal Subset, the Input to ASSET	52
	Figure 5.2	The Modfile.c of Figure 4.1	54
	Figure 5.3	The Structure Chart of the Testcov Module	56
Chapter 6	Figure 6.1	An example of a C Program	64
	Figure 6.2	The Modfile.c of Figure 6.1	65
Chapter 7	Figure 7.1	The Variable Information File of Figure 4.1	80

CHAPTER 1

INTRODUCTION

Software testing is done within and after the implementation phase of the Software Development Life Cycle (SDLC) in order to produce working software. It has been claimed that testing is a demonstration that faults are not present. Despite the fact that some organizations spend up to 50% of their software budget on testing, delivered tested software are unreliable. The reason for this contradiction is simple. As Dijkstra puts it, “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence” [Dijkstra 1972]. What Dijkstra is saying is that if a product is executed with test data and the output is wrong, then the product definitely contains a fault. But if the output is correct, then there still may be a fault in the product; all that particular test has shown is that the product runs correctly on that particular set of test data. So testing is the process that gives more confidence in the software reliability and correctness, but it never shows that the software is free of bugs.

Software Testing is time consuming. A significant part of software cost is the cost of bugs: The cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.

Another problem of testing is that many programmers appear to be unaware that tests themselves must be designed and tested. Too often, test cases are attempted without prior

Analysis of the program's requirements or structure. Such test design is just a haphazard series of cases that are not documented either before or after the tests are executed. Because they were not formally designed, they cannot be precisely repeated, and no one is sure whether there was a bug or not. After the bug has been corrected, no one is sure that the retest was identical to the test that found the bug.

In order to produce designed test cases many testing strategies were proposed. Those testing strategies produce designed tests that aim at detecting errors, documenting and correcting them. Each testing strategy is designed to detect and correct some kind of errors in order to increase the programmer confidence in the reliability of the tested program. Testing strategies can be grouped into two classes, The Black box and the Glass box testing strategies.

The black box testing strategies, or functional testing strategies, totally ignore the structure of the program or code unit under test. The software element is considered a black box, which when presented with the input parameters produces an output similar to that defined in the specification. Testing is a comparison of the actual output with the specified output [Beizer 1990].

The Glass box testing strategies, or structural testing strategies looks at the code and the structure of that code. There are a number of different forms of glass box testing, including statement, branch, and path testing strategies [Beizer 1990].

Path testing is the name given to a family of testing strategies based on judiciously selecting a set of test paths through the program. Path testing strategies are the oldest of all the structural testing techniques and are applicable for unit testing. However, path testing requires complete knowledge of the program's structure. It is most often used by programmers to unit test their own code. The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases. Path testing is rarely, if ever, used for system testing. The assumption for path testing strategy is that something has gone wrong with the software that makes it take a different path than intended. It is also assumed in path testing that specifications are correct and achievable, and that there are no processing bugs other than those that affect the control flow. This is why path testing usually utilizes the program's control flow graph which is a graphical representation of a program's control structure.

Using the path testing strategy as the basis for test data selection presents two distinct problems. The first problem is that a bug could create unwanted paths or make mandatory paths unexecutable. Further, just because all paths are right doesn't mean that the routine is doing the required processing along these paths.

The second problem is that programs with loops may have an infinite number of paths, so path testing strategy can never lead to complete testing.

Rapps and Weyuker have proposed a family of test data selection criteria, the data flow criteria, which address the problems of the path testing strategies. The data flow criteria are based on the idea that rather than selecting program paths based on the control structure of a program, we track input variables through the program, following them as they are modified, until they are ultimately used to produce output values [Rapps, Weyuker 1985].

The data flow criteria are constructed so that associations between the definition of a variable and its uses are examined using the program def-use graph. Just as one would not feel confident about the correctness of a portion of a program which has never been executed, one has no reason to believe that the correct computation has been performed, if the result of some computation has never been used.

In this project, we have developed a Computer Aided Software Engineering (CASE) tool for testing based on data flow criteria proposed by Rapps and Weyuker [Rapps, Weyuker 1985]. The tool takes a C program as input and produces the control flow graph and the def-use graph for the input program. Also, it assists the tester in designing test cases and provides information about test coverage. This tool is a part of a larger project undertaken at LAU, leading for the development of the ELISSAR a CASE tool for testing and regression testing that provides the following functionality :

- (a) It generates and displays control flow graph

- (b) It generates and displays def-use graph
- (c) It assists in dataflow testing
- (d) It provides test coverage information
- (e) It includes regression testing algorithms that suit different user requirements

[Baradhi 1996, Arabi 1997].

This report is organized as follows. Chapter 2 gives the tool architecture and global data structures. Chapter 3 presents the user interface. Chapter 4 explains the generation of the program flow graphs. Chapter 5 presents the testing part of the tool. Chapter 6 gives the experimental results. Chapter 7 presents the user guidelines. Chapter 8 concludes the report.

Tool Architecture

```

graph TD
    Elissar --> Testing
    Elissar --> Regression_Testing[Regression Testing]
    Testing --> CtrlDuse
    Testing --> FGCD
    Testing --> Testcov
    Regression_Testing --> Reduction
    Regression_Testing --> Slicing
    Regression_Testing --> Incremental
    Regression_Testing --> Genetic
    Incremental --> Simulated_Annealing[Simulated Annealing]
    Incremental --> Firewall

```

Figure 2.1 contains the structure chart for ELISSAR: a CASE tool for testing and regression testing. The structure chart displays all the important modules for ELISSAR.

The complete structure chart of the `ctrlfuse` module is presented in section 4.3.1 and the complete structure chart of the `testcov` module is presented in section 5.3.1. The `FGTCD` and `Reduction` modules were built by Mr. Nidal Araby [Araby 1997]. The `Incremental`, `Firewall`, `Slicing` modules were built by Miss. Ghinwa Baradhi [Baradhi 1996]. The `Genetic` module was built by Mr. Khaled Fakih [Fakih 1996]. The `Simulated Annealing` module was built by Dr. Nashaat Mansour [Mansour 1994].

2.2. Global data structures and variables declaration

In this section we will give the declarations for global data structures and variables which were used in the implementation of `ELLISAR`.

```
typedef struct node{
    int con1,con2,tag,nest,lev,e_flg,nodenum;
    struct node *link;
    struct node *prev;
    int expl1,expl2;
}stack;
```

```
typedef struct list {
    int cnt,else_brac;
    struct list *next;
    struct list *before;
}pointer;
```

```
typedef struct list2 {
    int ndnum;
    char varname[15];
    char vtype;
    int i_o,loop,type ;
    struct list2 *after;
}varptr;
```

```
typedef struct list3{
```

```
        int e1,e2 ;
        char ename[15];
        char etype ;
        struct list3 *enext;
    }edgptr;

typedef struct list4{
    int node ;
    struct list4 *plink;
}path;

typedef struct list5{
    char dcuvvar[15];
    int i;
    path *dcuptr;
    struct list5 *dculink;
}dcu;

typedef struct list6{
    int node1,node2 ;
    struct list6 *pnext;
}tdpu;

typedef struct list7{
    char dpuvar[15];
    int i;
    tdpu *dpuptr;
    struct list7 *dpulink;
}dpu;

typedef struct list8{
    char adefvar[15];
    int n1,n2,n3;
    struct list8 *adeflink ;
}adef;

stack *flowlist;
varptr *varlist;
edgptr *edglist;
dcu *dculist;
dpu *dpulist;
dcu *dulist;
char filename4[15];
```

CHAPTER 3

USER INTERFACE

3.1. Menu tree

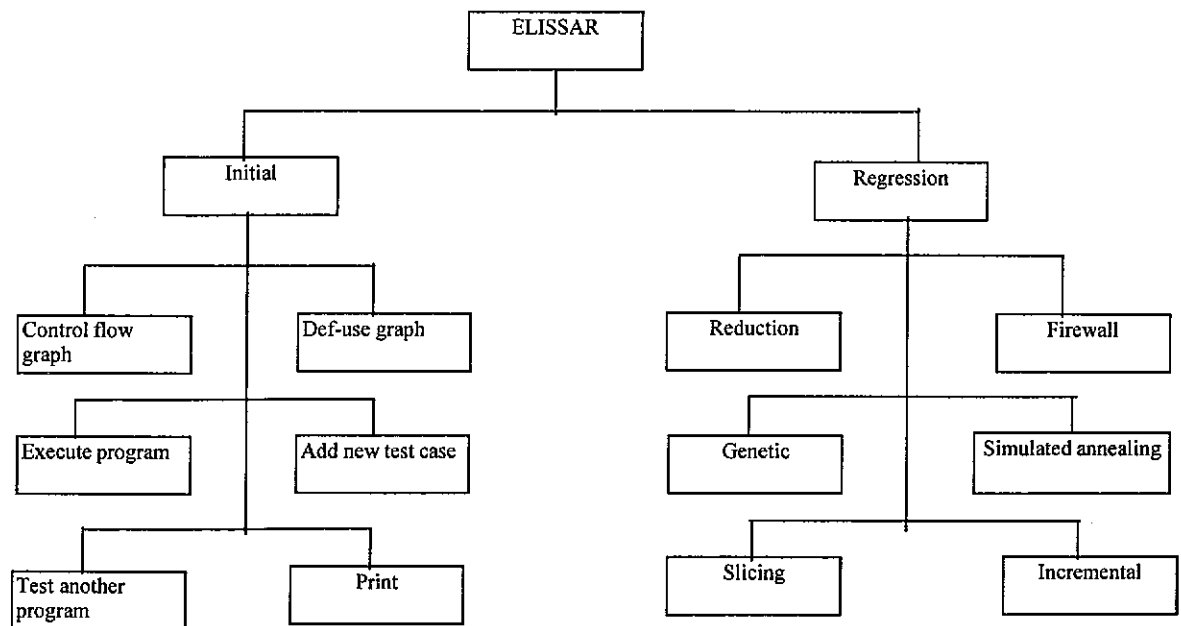


Figure 3.1. Menu tree

Figure 3.1 contains the menu tree of ELISSAR.

3.2. Menus

This section contains the ELISSAR's menus. The menus found in Figures 3.1, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.22 and 3.23 were implemented by Mr. Nidal Araby [Araby 1997].

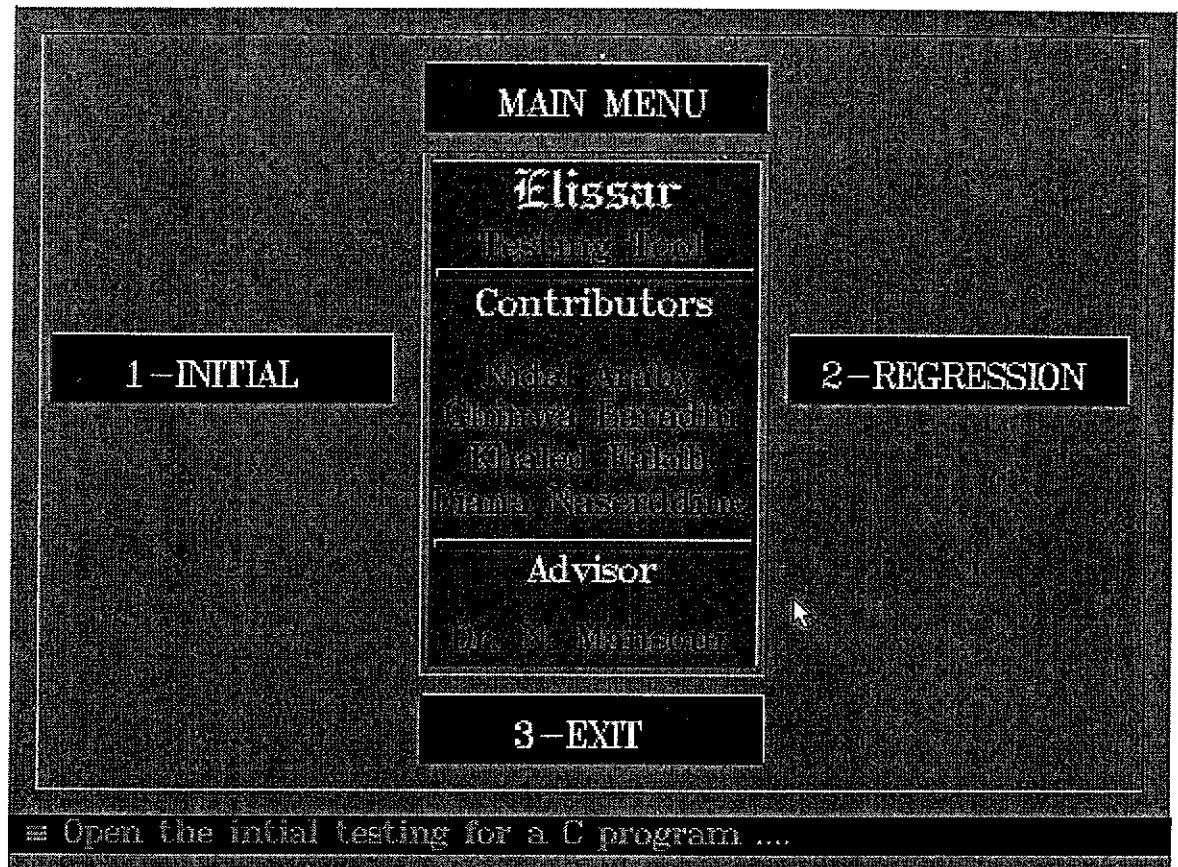


Figure 3.2. ELISSAR Main menu.

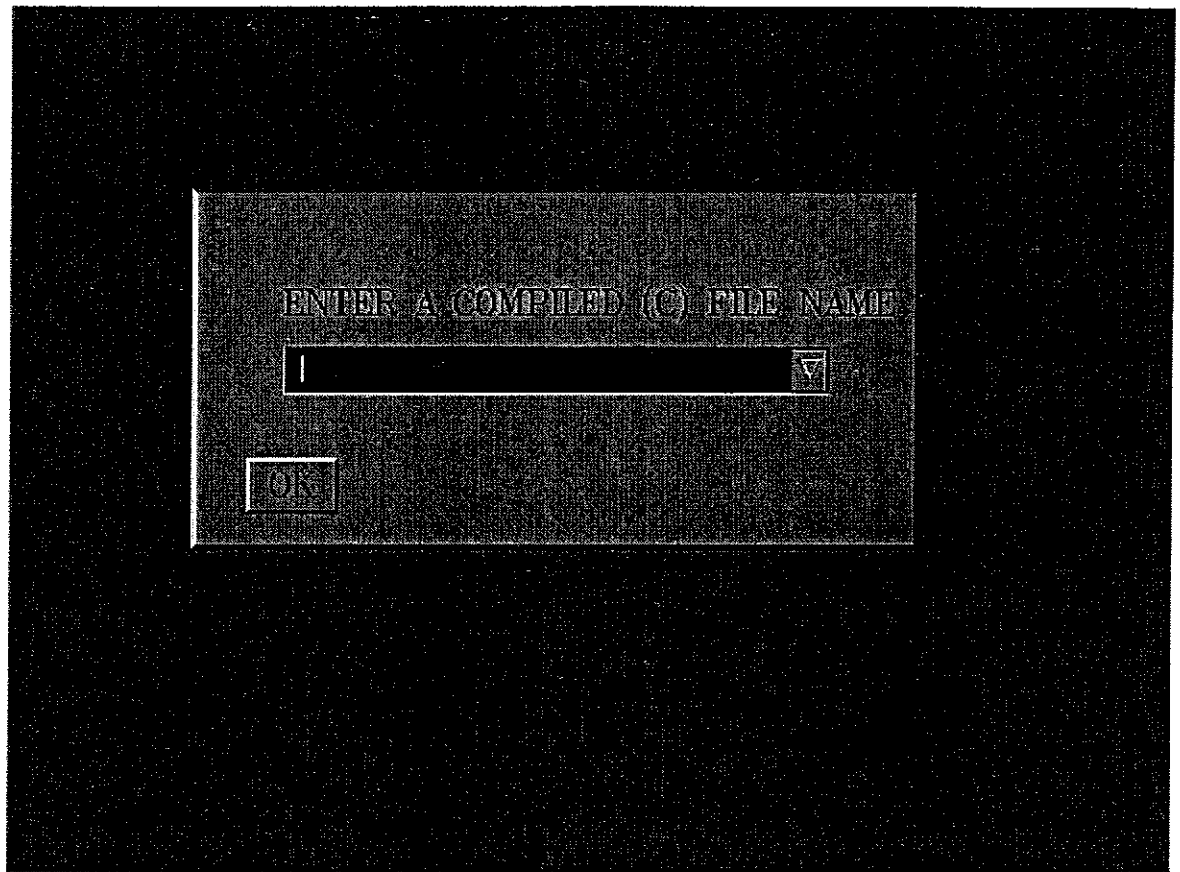


Figure 3.3. Initial Testing Input Menu.

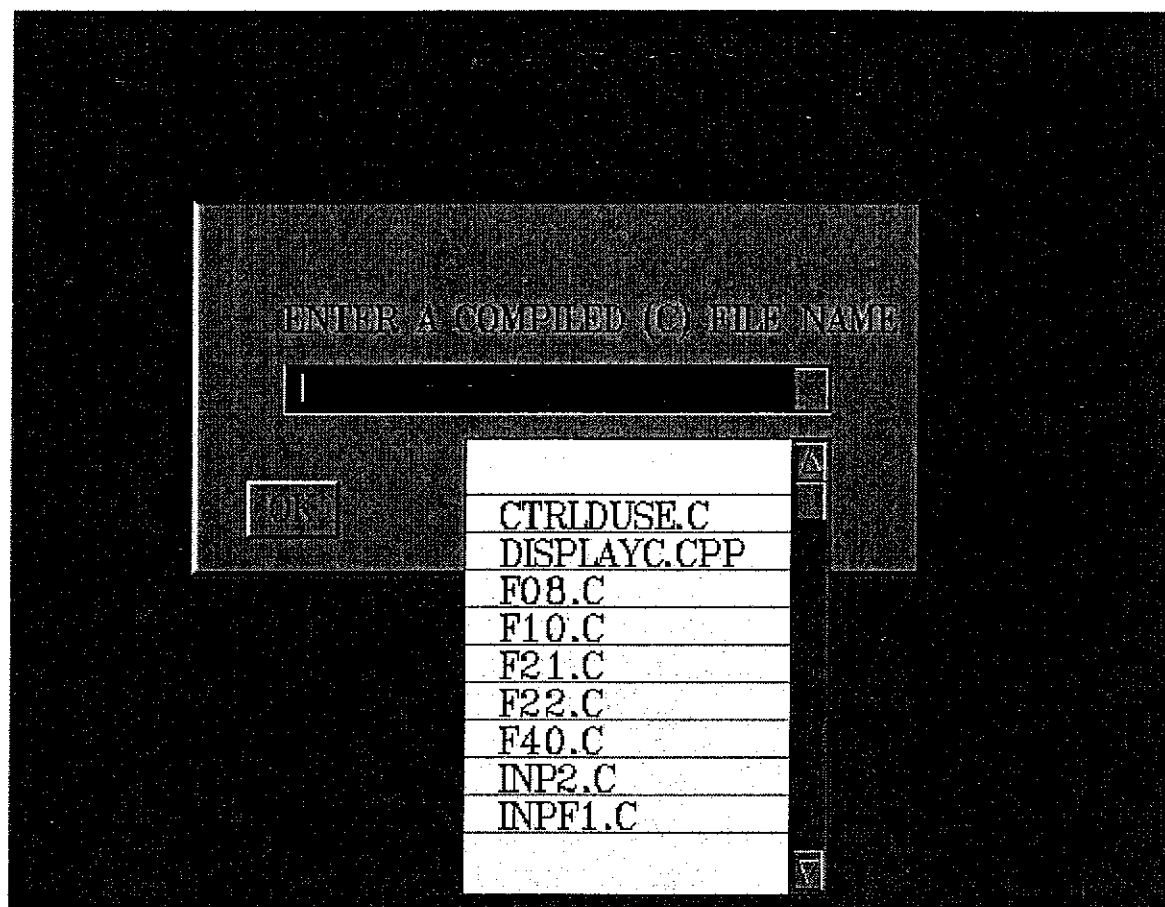


Figure 3.4. Initial Testing Input File Selection Main Menu.

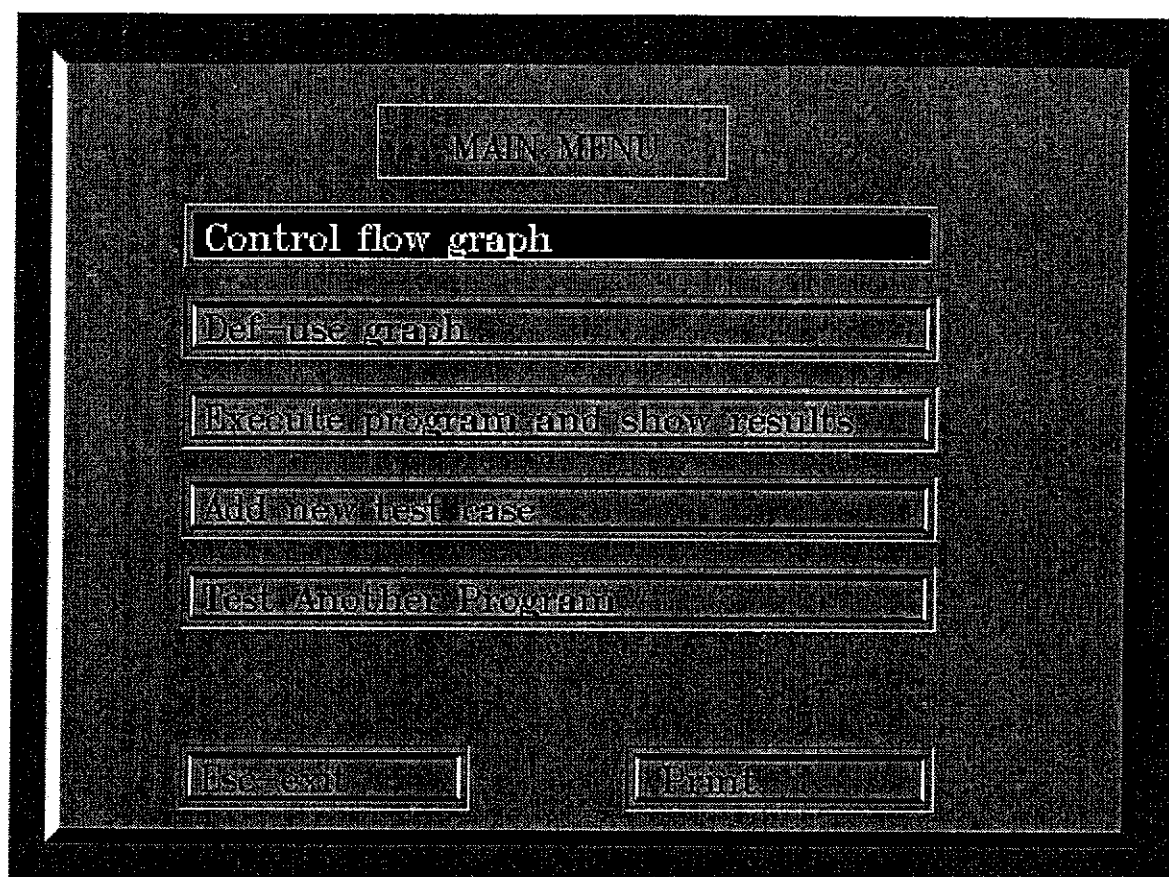


Figure 3.5. Initial testing Main menu.

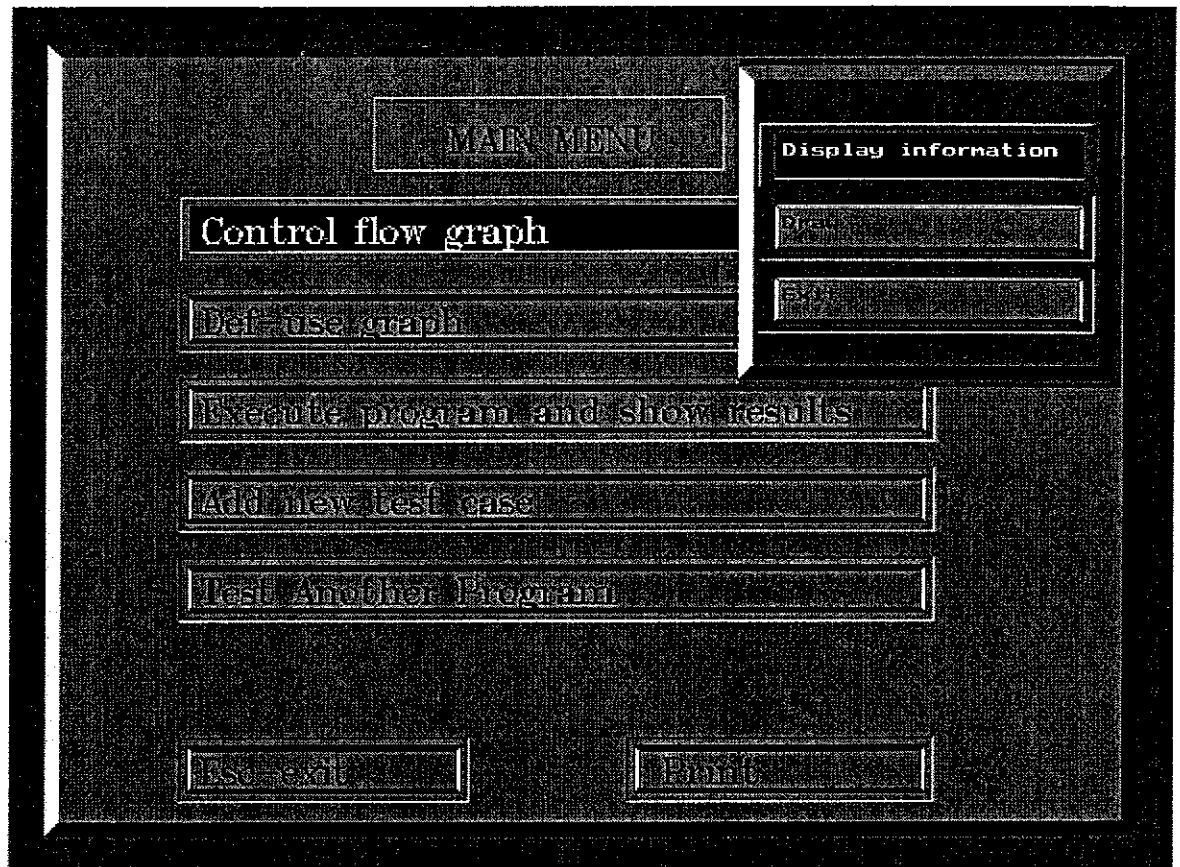


Figure 3.6. Control Flow Graph Menu.

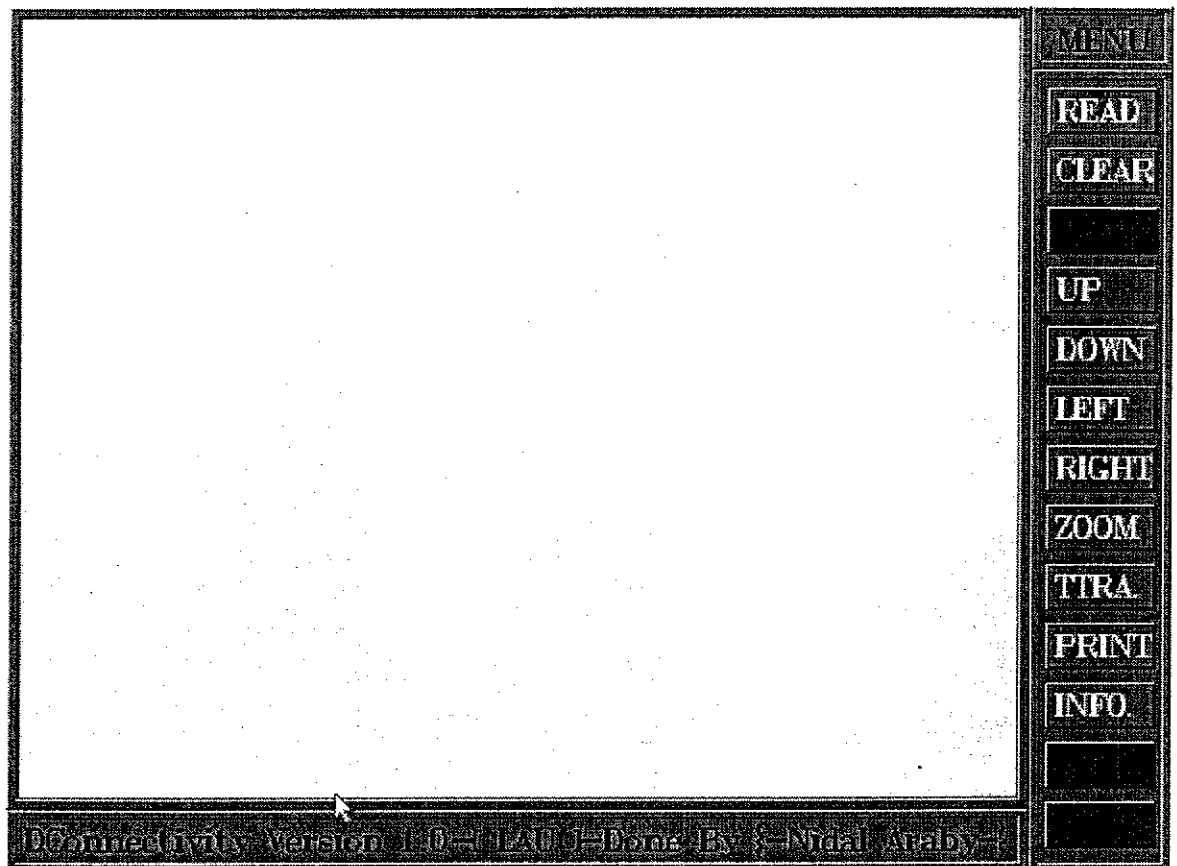


Figure 3.8. The FGTC D Main Menu.

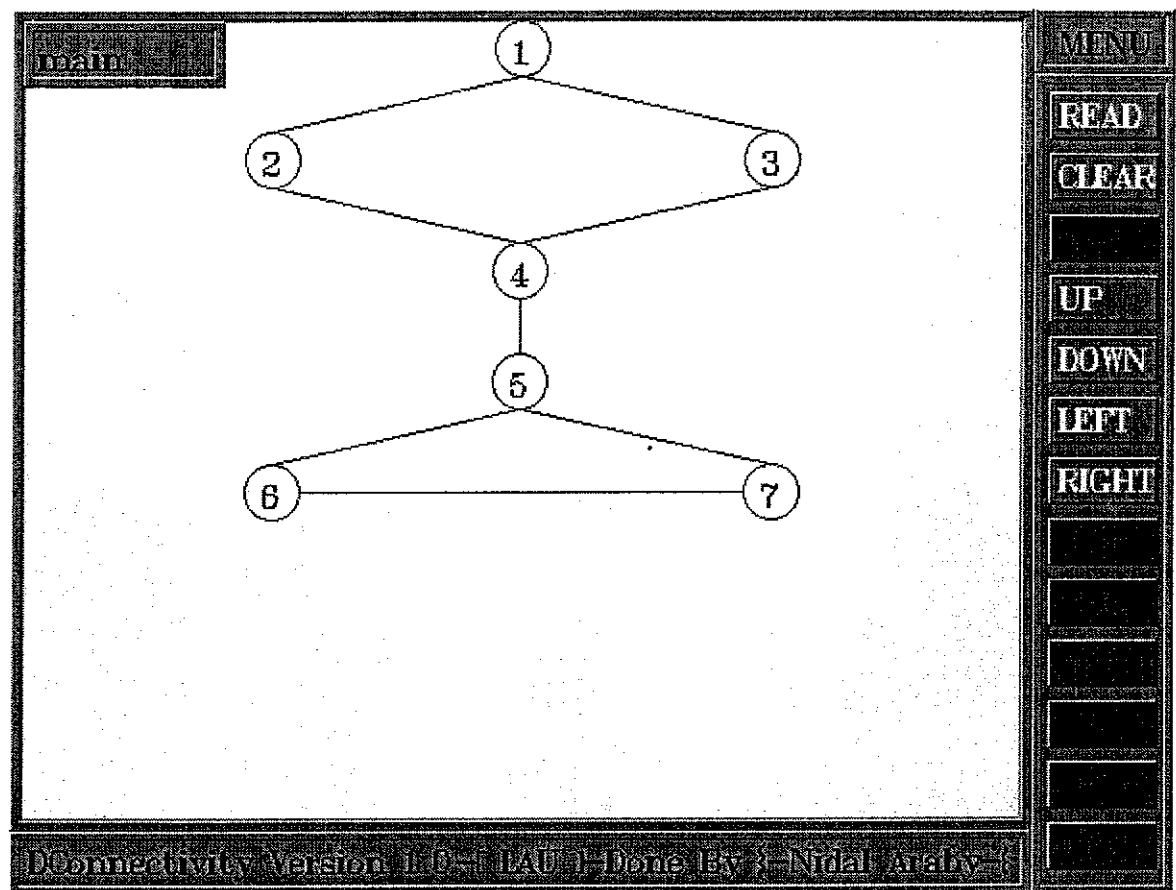


Figure 3.9. The Graphical Draw of the FGTCD.

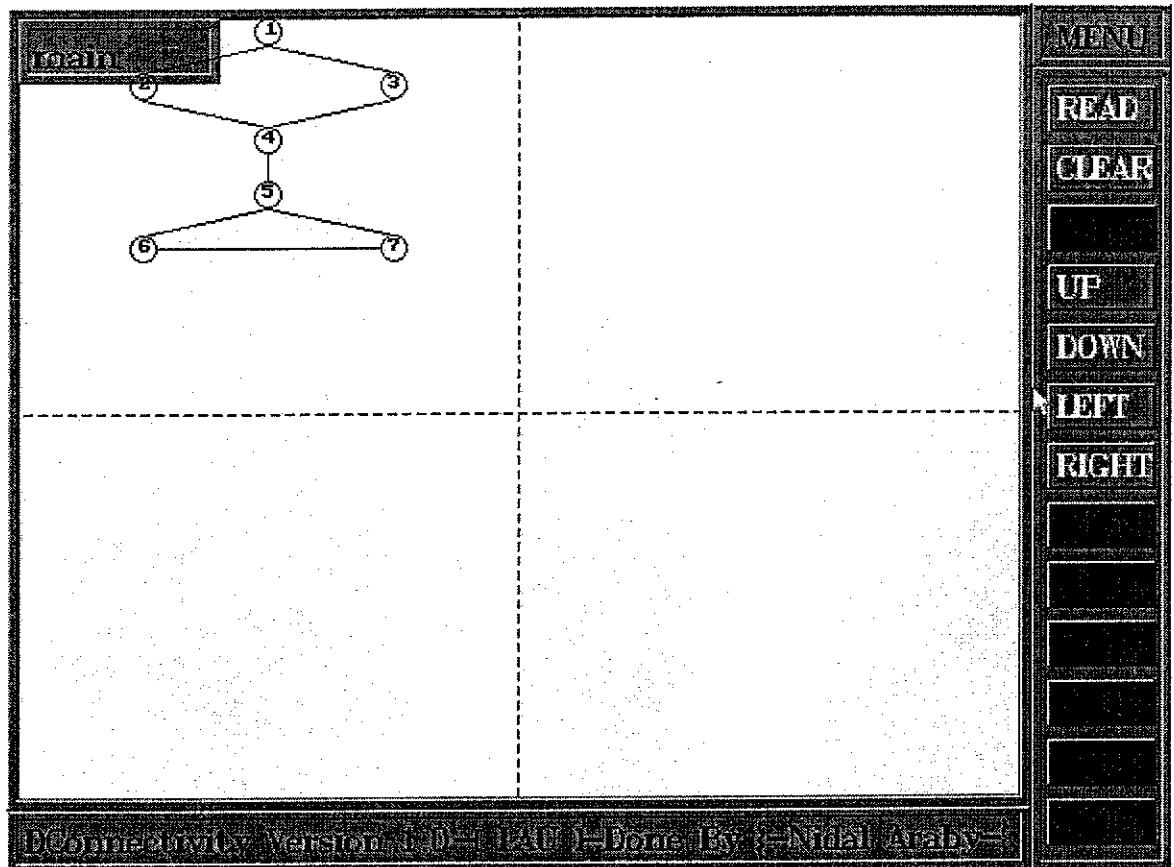


Figure 3.10. A Zoomed Draw of the Control Flow Graph.

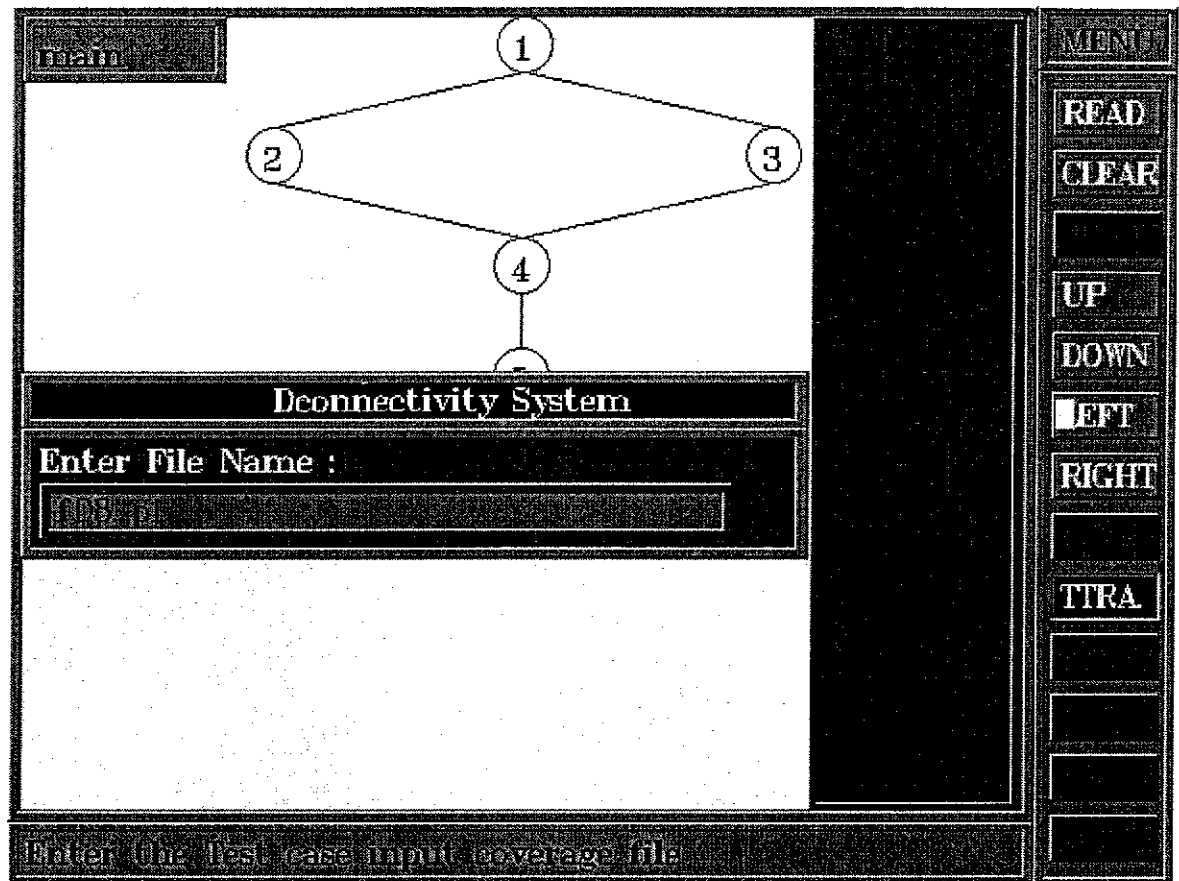


Figure 3.11. The test Traversal Input Screen Menu.

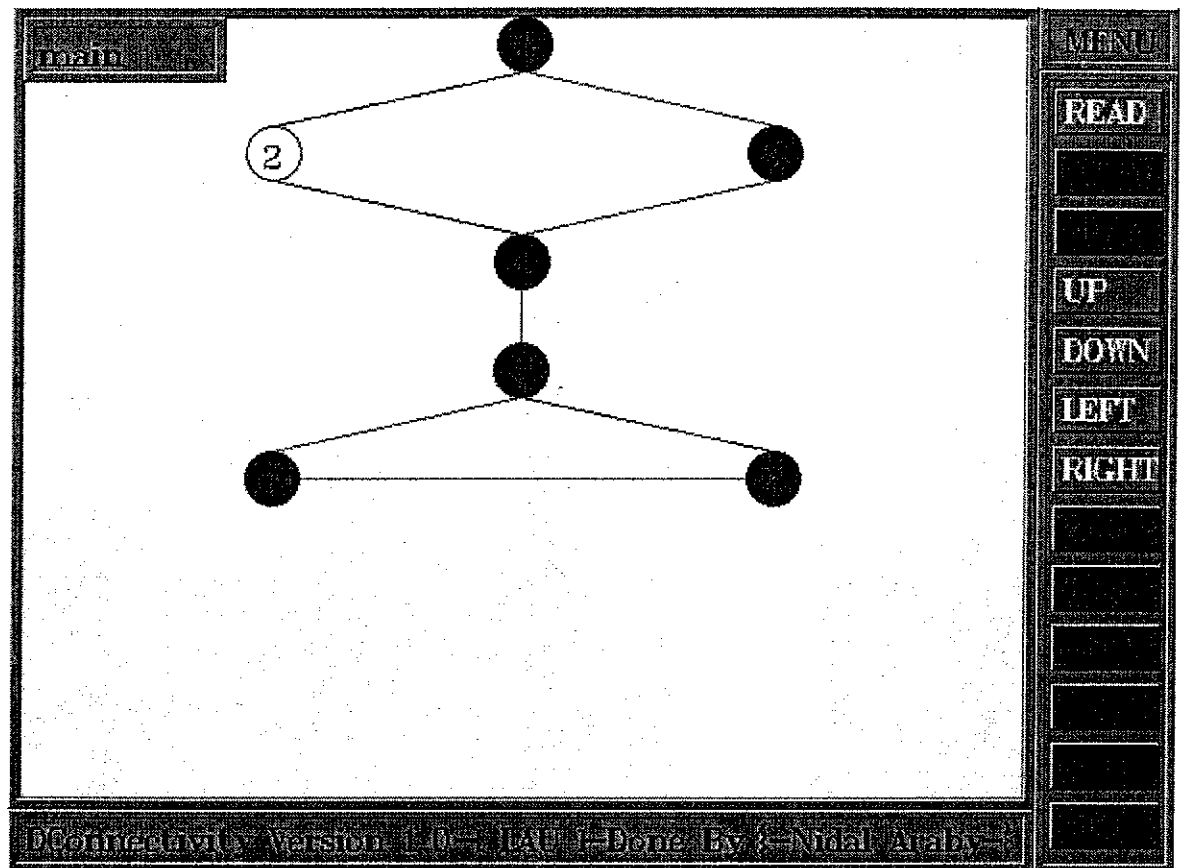


Figure 3.12. The test Traversal Path Graphical Draw.

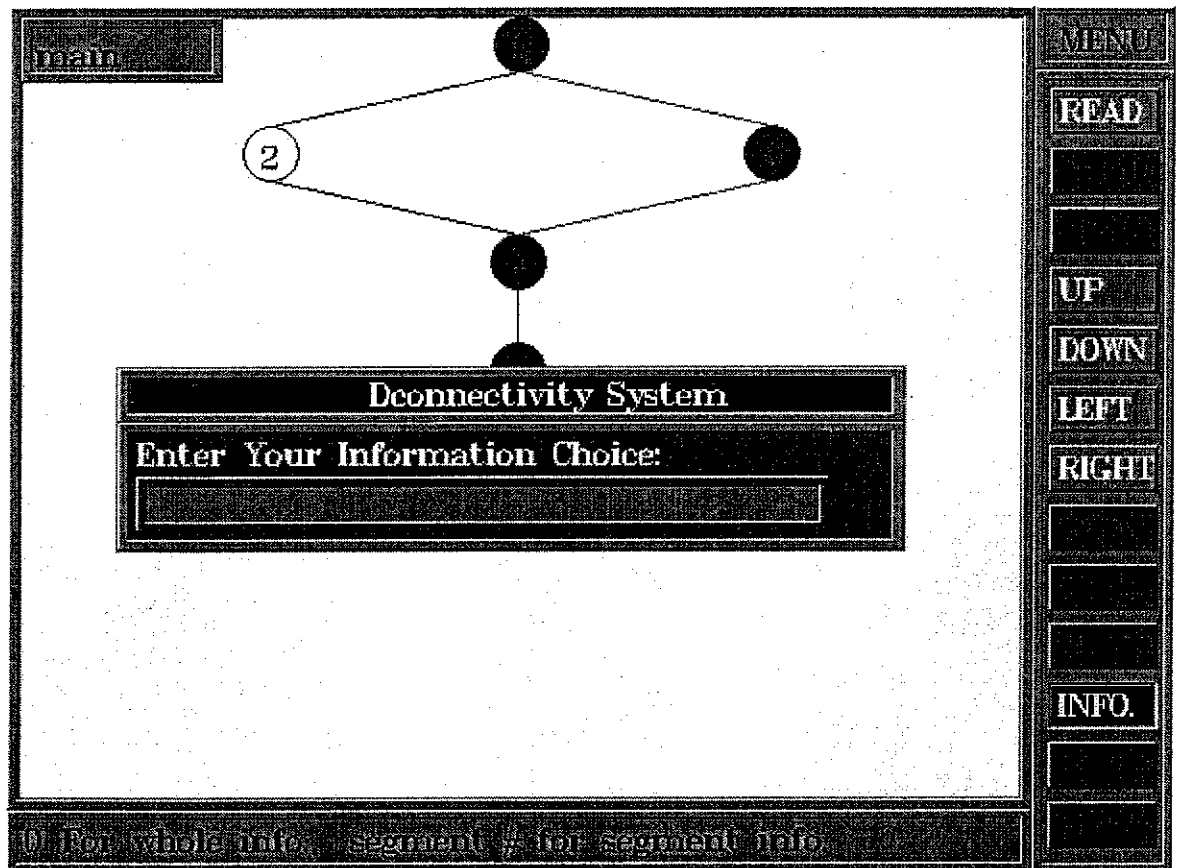


Figure 3.13. The Data Flow Graph Draw Menu.

Dconnectivity - Variable Screen			MENU
Function Name: Variable			READ
Segment: 0 Variable: onpt Definition: Use			
SEGMENT	VARIABLE	USE	
0	inpt	Definition Use	UP
0	onpt	Definition Use	DOWN
0	A	Definition Use	LEFT
0	B	Definition Use	RIGHT
0	X	Definition Use	
2	X	Definition Use	
3	A	Definition Use	
3	B	Computation Use	
4	X	Definition Use	INFO.
Hitachi Data Engineering Inc. Confidential			
Dconnectivity Version 1.0 © 1997 Hitachi Data Engineering Inc.			

Figure 3.14. The Data Flow Graph Tabular Screen.

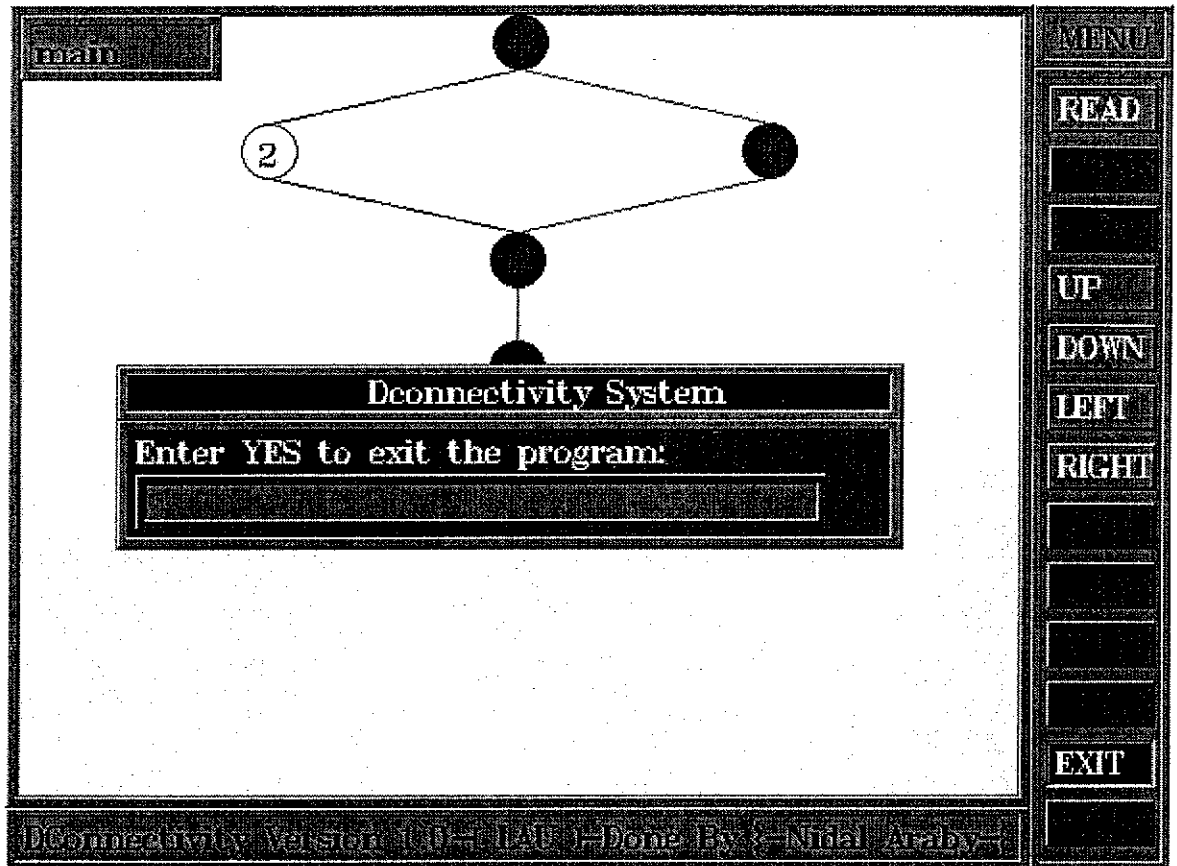


Figure 3.15. The FGTC D Exit Menu.

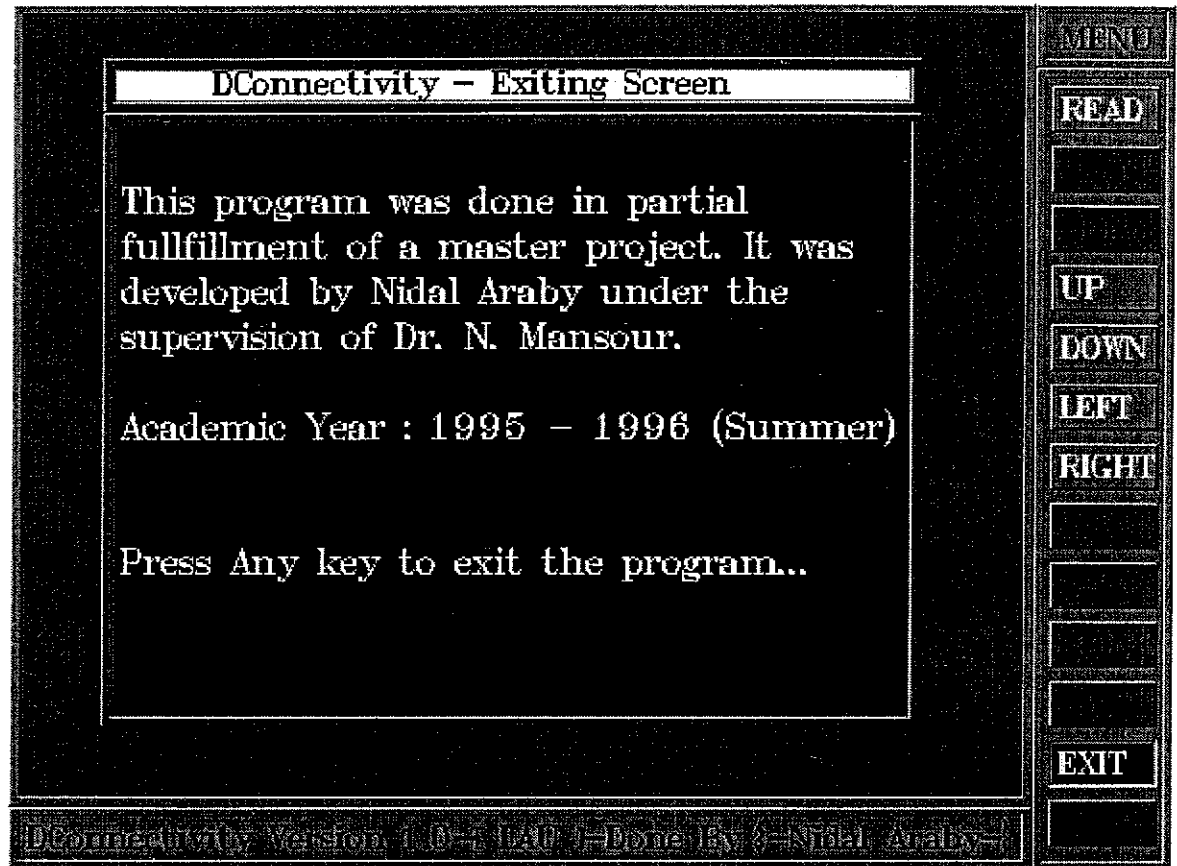


Figure 3.16. The FGTC D Exit Screen.

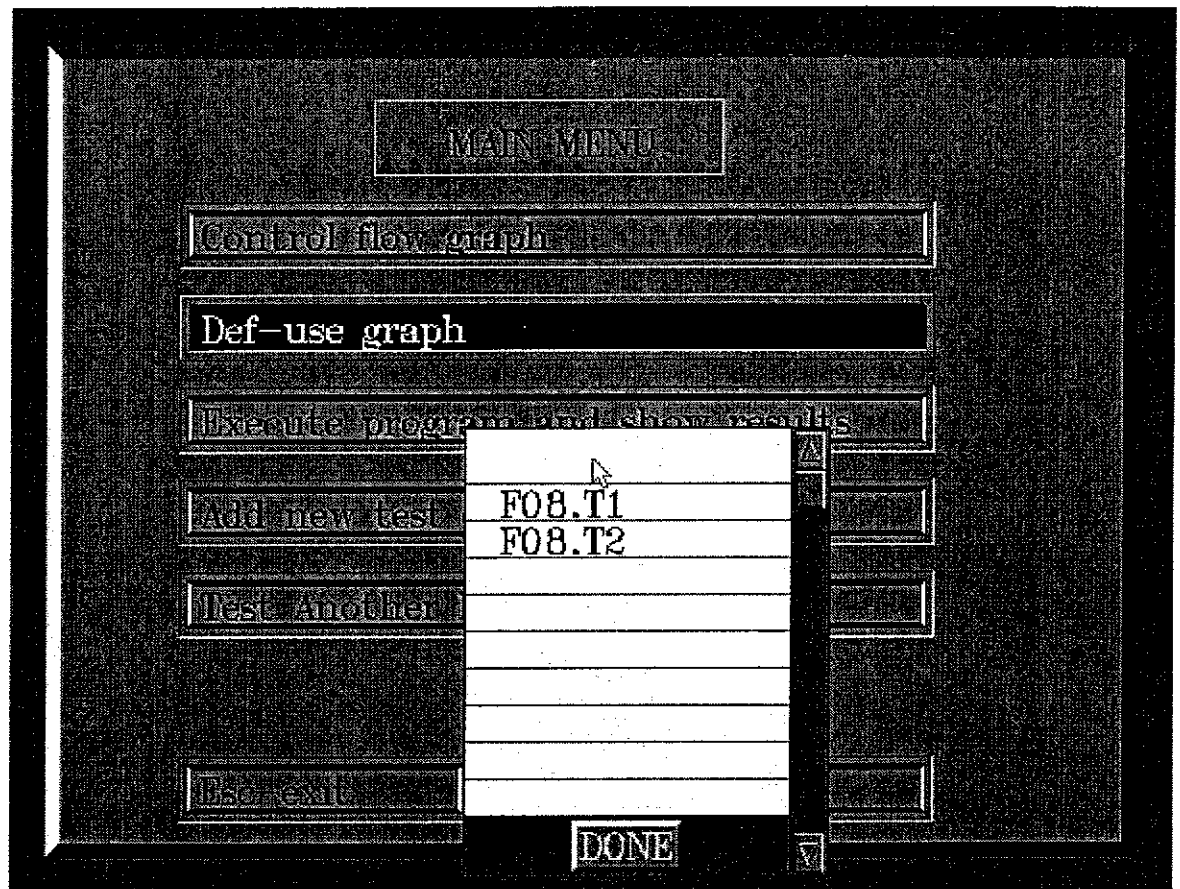


Figure 3.18. The Test Set Input Menu.

FROM	TO
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
10	11
11	12
12	13
13	14
14	15
15	16

PRESS ANY KEY TO SEE NEXT PAGE || PRESS THE LEFT BOTTOM
TO PROCEED || RIGHT BOTTOM / ESC to EXIT

Figure 3.19. The Test Traversal Path Tabular form Screen.

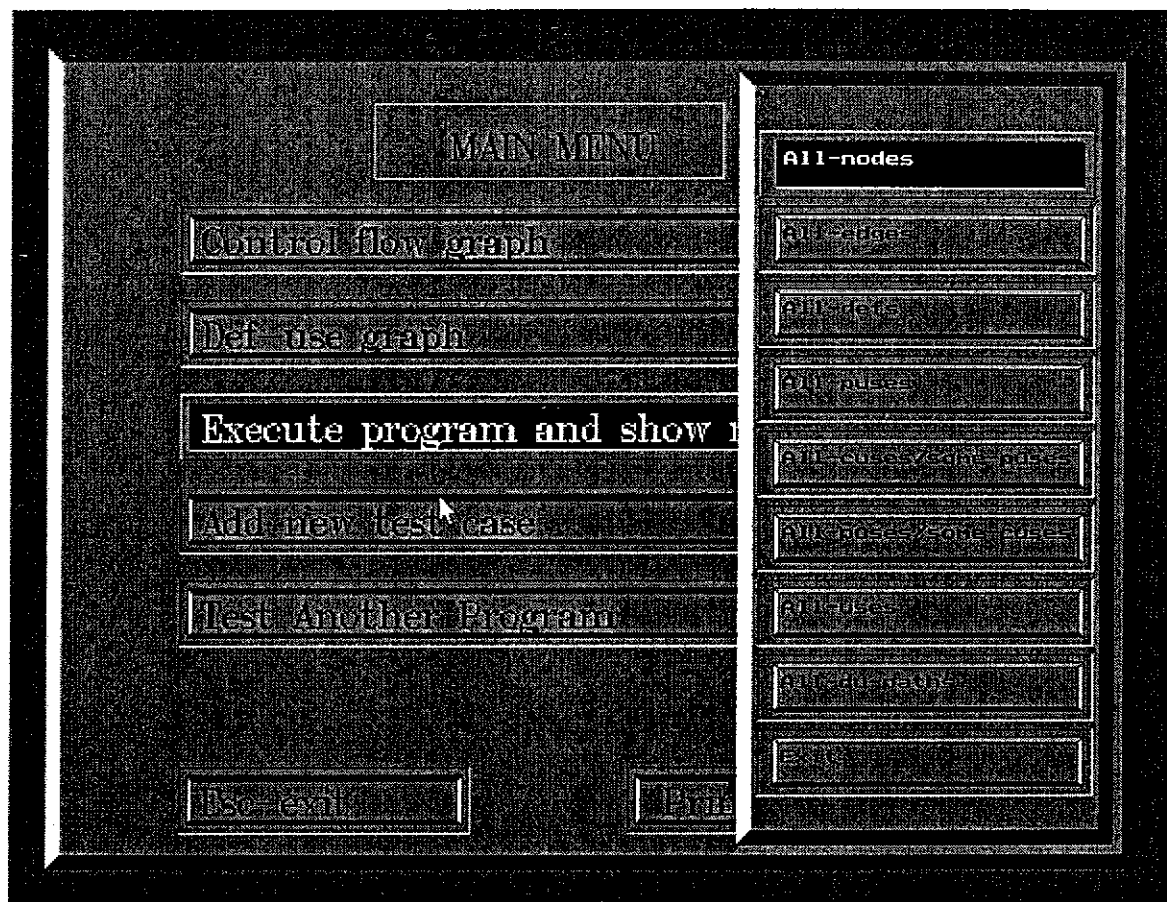


Figure 3.20. The Data Flow Criteria Main Menu.

```
still must exercise the du-path
with respect to  A
0
1
2
and
still must exercise the du-path
with respect to  X
4
5
7
and
still must exercise the du-path
with respect to  inpt
0
1
2
4
5
6
7
```

Figure 3.21. The Output Screen of the Selection of a Data Flow Criterion.

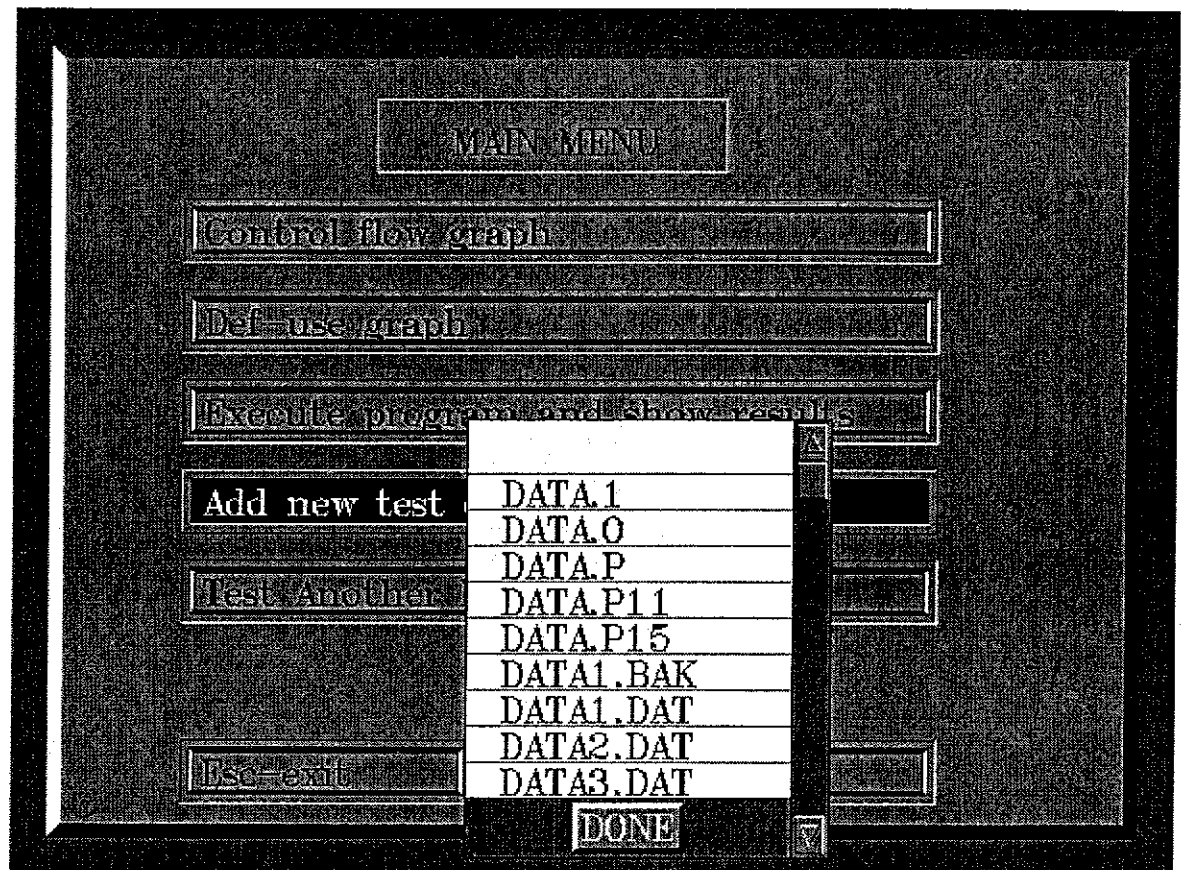


Figure 3.22. The Test Another Program Selection Menu.

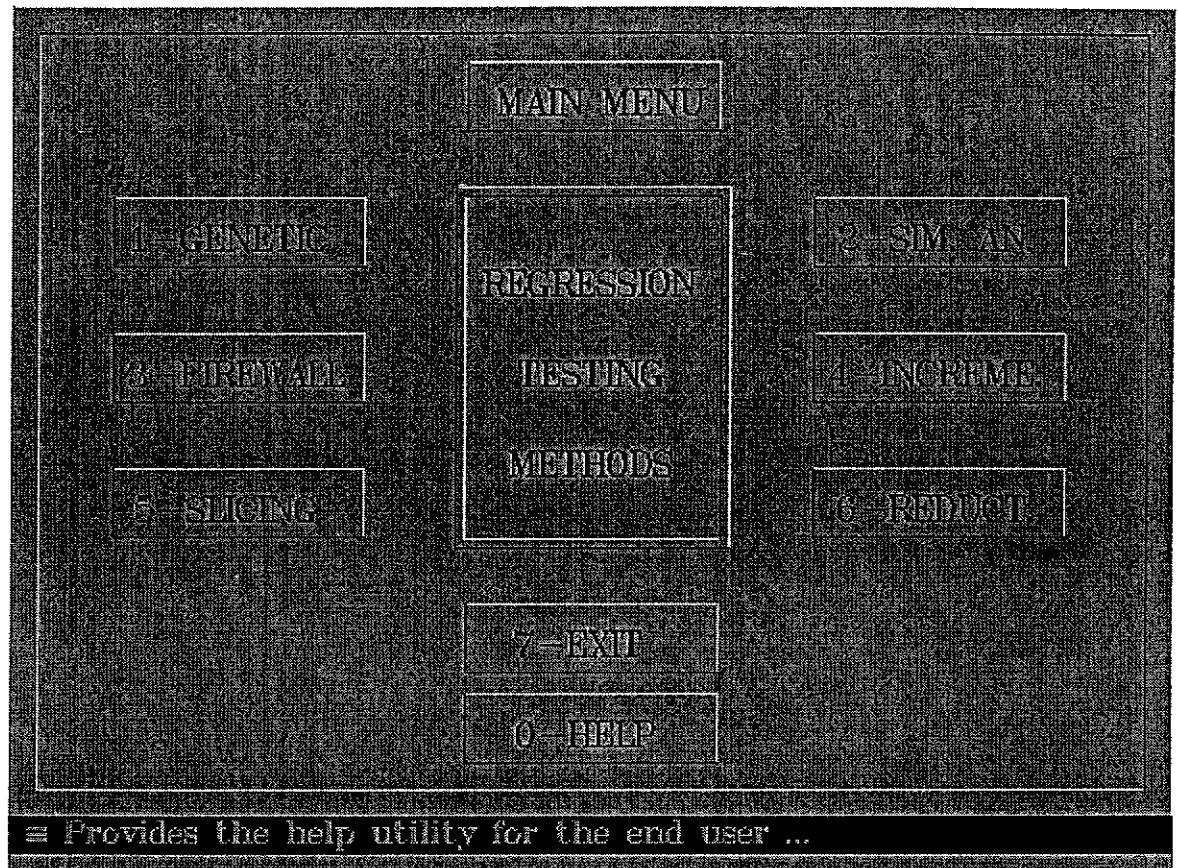


Figure 3.23. The Regression Testing Module Main Menu.

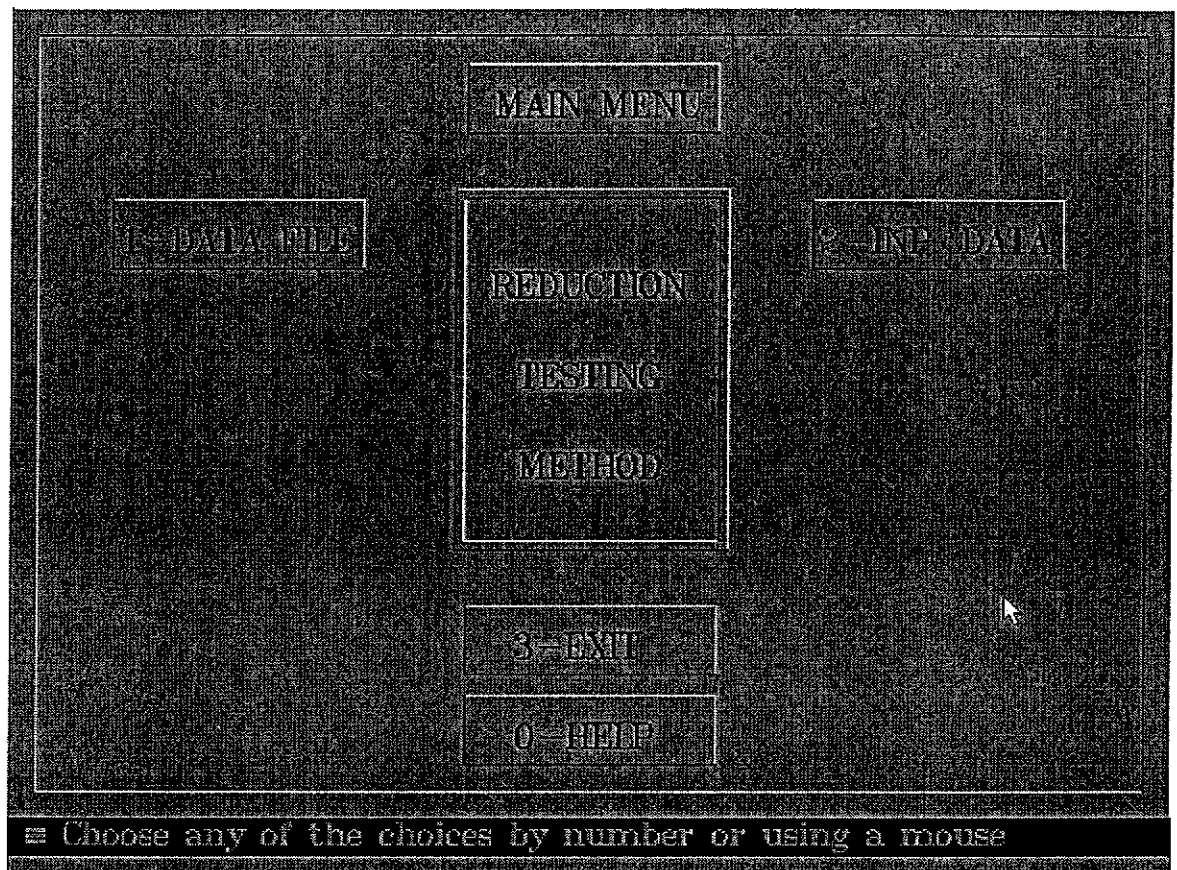


Figure 3.24. The Reduction Testing Method Main Menu.

CHAPTER 4

PROGRAM FLOW GRAPHS

In this chapter we will define control flow and def-use graphs, we will give an example for each one of them and we will talk about the implementation of each one of them. In what follows we will refer to the program found in Figure 4.1.

4.1. Control flow graph

The control flow graph is a graphical representation of a program's control structure. It consists of the following elements: process blocks, decisions, and junctions [Beizer 1990].

1. A process block is a sequence of program statements uninterrupted by either decisions or junctions. It is a sequence of statements such that if any statement of the block is executed, then all statements thereof are executed. A process block can be one source statement or hundreds and it has one entry and one exit
2. A Decisions is a program point at which the control flow can diverge. The if statement and the while statement are examples of decision
3. A junction is a point in the program where the control flow can merge. an example of junctions are the end and until.

The control flow graph of the program found in Figure 4.1 is shown in Figure 4.2.


```
#include <io.h>#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define null 0

FILE *inpt,*onpt;
main( )
{
    int e,t,d,x,c;

    inpt = fopen("data.c","r");
    onpt = fopen("out.c","w");
    while(!feof(inpt))
    {
        fscanf(inpt,"%d %d",&d,&e);
        x=0;
        c=2*d;
        while(d > e)
        {
            d= d/2;
            t= c - (2*x+d);
            if(t > 0)
                c= 2*((c - (2*x+d)));
            else
                x = x+d;
        }
        x = x +8;
        d = 8;
    }
    if( x > 100)
        x = x/2;
    else
        x = 9;
    fprintf(onpt,"%d %d",d,e);
    fclose(inpt);
    fclose(onpt);
    return;
}
```

Figure 4.1. An example of a C program.

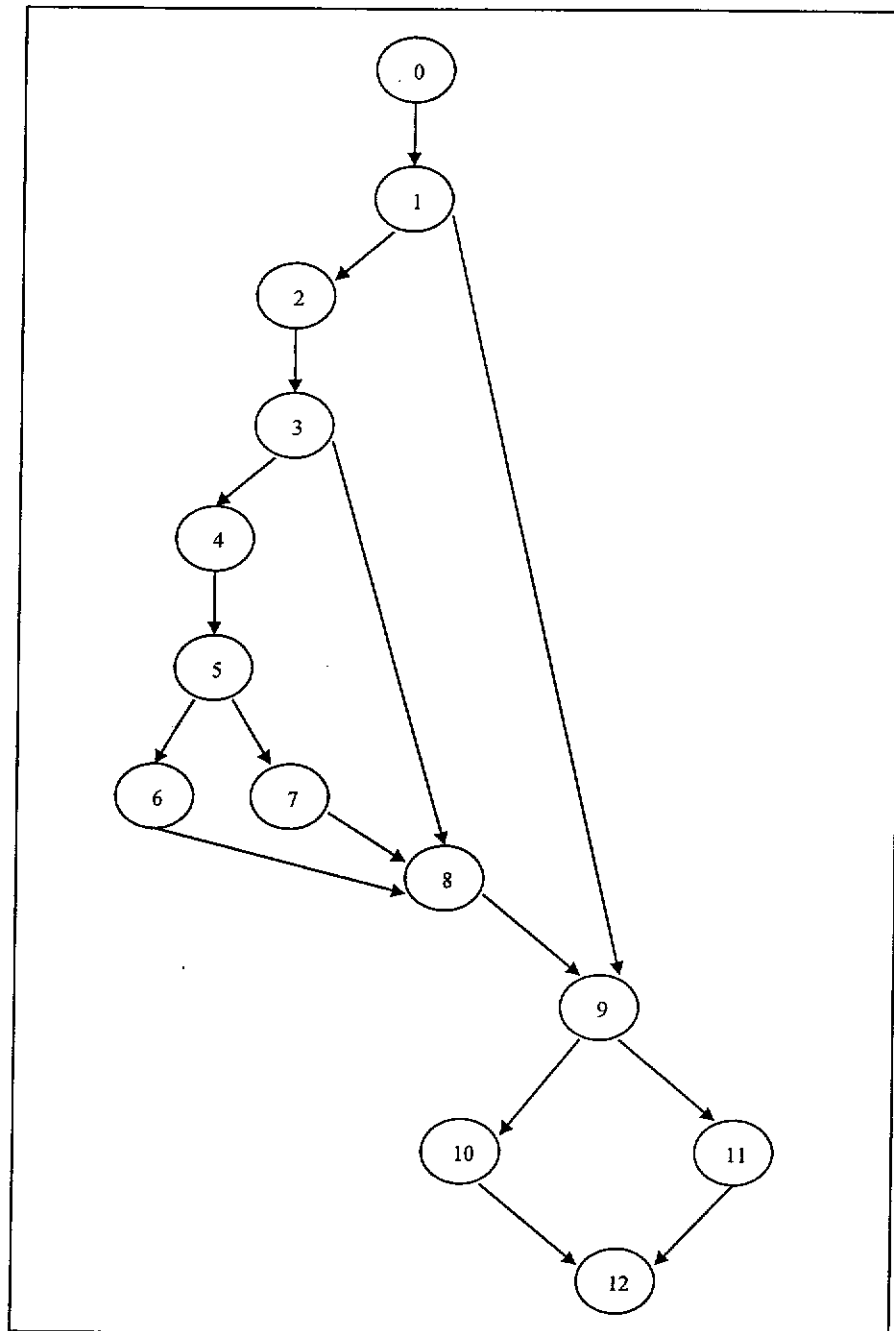


Figure 4.2. The control flow graph of the program found in Figure 4.1.

4.2. Def-use graph

The def-use graph is constructed from the program control flow graph by associating a set with each edge and two sets with each node[Rapps, weyuker 85]. With each edge (i, j) we will associate the set p-use(i, j) which is the set of variables for which edge(i, j) contains a predicate use. And with each node (i) we associate the set def(i) which is the set of variables for which node i contains a global definition, and the set c-use(i) which is the set of variables for which node i contains a global use.

In Figure 4.3 we have the def-use graph of the program found in Figure 4.1.

node	c-use	def	edge	p-use
0	ϕ	{inpt, onpt}	(1, 2)	{inpt}
2	{d}	{d, e, x, c }	(1, 9)	{inpt}
4	{d, c, x,d}	{d, t}	(3, 4)	{d, e}
6	{c, x, d}	{c}	(3, 8)	{d, e}
7	{x, d}	{x}	(5, 6)	{t}
8	{x}	{x, d}	(5, 7)	{t}
10	{x}	{x}	(9, 10)	{x}
11	ϕ	{x}	(9, 11)	{x}
12	{d, e, inpt, onpt}	ϕ		

(a)

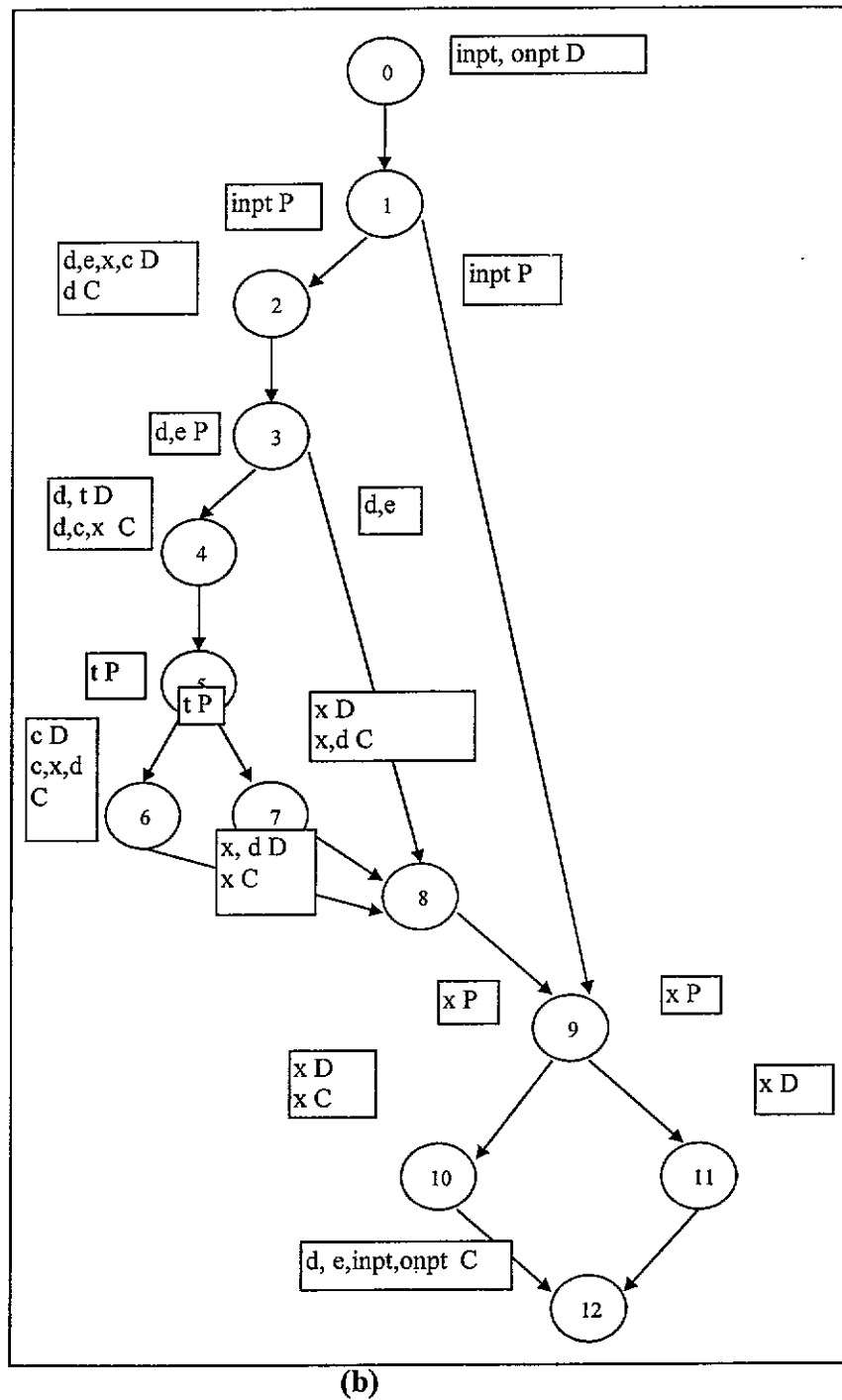


Figure 4.3. (a) The tabular form of the def-use graph of Figure 4.1
 (b) The graphical form of the def-use graph of Figure 4.1.

4.3. Implementation of the control flow and def-use graphs

In order to build the two graphs we have lexically analyzed the input code. Each statement has been identified as being either a control statement such as the if and while... statements, or to be a normal statement such as the assignment, read and write statements. The occurrence of a control statements create a new node in the control flow graph. The statements that have occurred before this control statement constitute a different node, and the statements that occur directly after the branching statement constitute another new node. The occurrence of “}” indicates the end of the current node; therefore the statement that follows a “}” will indicate the beginning of a new node. In this manner the code is divided into nodes. Each node have either one or two successors. If the node is a control statement then it will have two successor, the first one is the node that will be executed if the control statement evaluates to true, the second is the node that will be executed if the control statement evaluates to false. If the node is a normal node then it will have only one successor which is the node that will be executed directly after it.

The def-use graph is built in parallel with the control flow graph. If the node is a normal node, then we will associate to it two sets: the definition and the c-use set. Each variable occurring in this node will be classified as occurring in the definition set or in the c-use set. If the node is a control node then we will associate with it and each of its two successors the p-use set. So each variable occurring in the control statement will be in the p-use set of each of the edges linking the control node to its two successors.

4.3.1. Structure chart

Figure 4.4 contains the structure chart of the ctrlduse module of Figure 2.1.

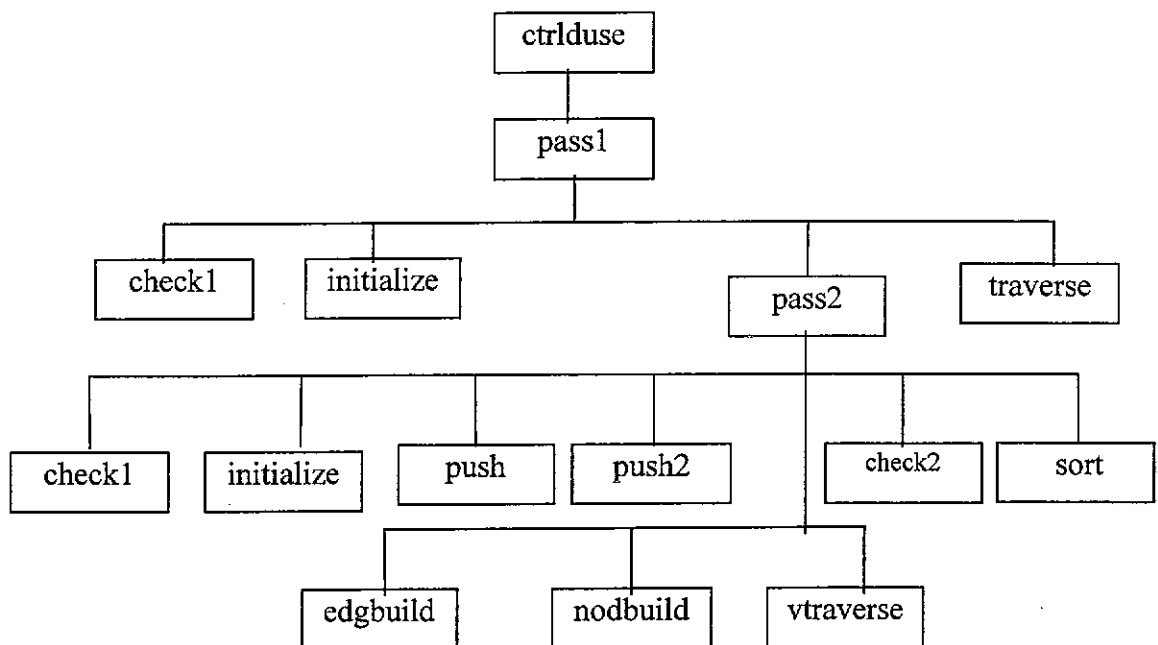


Figure 4.4. Structure chart of the ctrlduse module

4.3.2. Detailed design

A. *Function pass1*

Description

This function reads from the input file until it reaches the first '{'.
when it reads the first '{' it calls pass2 .

Pseudo code

```

open input and output files;
initialize variables ;
brac = 0;
get the first character of the input program into c;
while not end of file of the input program do
    write c to modfile.c;
    if (c = '{' )
        write c to modfile.c;
        brac = brac +1;
        get the next character of the input program into c;
    else
        if (c = '}' )
            brac = brac - 1;
            write c to modfile.c;
            get the next character of the input program into c;
        else
            if (c = '/')
                write c to modfile.c;
                get the next char of the input program into c;
                if c = '*'
                    read the remaining of the comment;
            else
                initialize(arr[]);
                get the next word from the input program into arr[];
                write arr[] to modfile.c;
                get the next character from the input program into c;
                if ((c = '(') and (brac = 0))
                    keep on reading from the input program and writing to modfile.c
until
    you read the { charecter;
    flowlist = null;
    seg_count = -1;
    call pass2(c,brac,flowlist,seg_count);
    call traverse(flowlist);
close input and output files;

```

B. Function pass2**Description**

This function build the control flow graph, the def-use graph and the modfile.c of the program.

Pseudo code

```

initialize variables;
do
  if c is a space or a tab character
    write c to modfile.c;
    get the next character of the input program;
  else
    if c is an operator or a separator
      write c to modfile.c;
      if((c = ';') and (else_ext = 4))
        else_ext = 0 ;
        flag=3;
        write the character '}' to modfile.c;
        if (( c = ';') and (nest_flg = 1))
          assign p to the head list;
          use p to go down the head list until you reach the last node;
          use p to go up the head list until you reach a branching node that is not
          connected to its
            second successor;
            p->con2 = seg-count+1;
            set flags to appropriate values;
            get the next character of the input program into c;
        else
          if (c = '{')
            brac = brac +1;
            write c to modfile.c;
            if( (flag != 2))
              seg_count = seg_count + 1;
              if(seg_count != 0)
                write to modfile.c the statement " fprintf(ofpt, seg-count);
                create a new node and insert it to the head list;
            get the next character from the input program;
          else
            if (c = '}')
              write c to modfile.c;
              brac = brac -1;
              if (brac != 0)
                p = head;
                use p to go down the head list until you reach the last node;
                use p to go up the head list until you reach a branching statement that is not

```



```

    connected to its
    second successor;
    p->con2 = seg_cout + 1;
    set appropriate flags;
else
    seg_count = seg_count + 1;
    create the program exit node an inserted into the head list;
    get the next character of the input program into c;
else
    if (c = '/')
        write c to modfile.c;
        get the next character of the input file into c;
        if (c = '*')
            read the remaining of the comment;
    else
        initialize(arr);
        get the next word from the input program into arr[];
        write arr[] to modfile.c;
        if arr[] is a branching statement
            create a new node for the branching statement and insert it into the
            head list;
            for every variable in the branching statement create a node and insert
            it into the varlist;
            set appropriate flags;
        else
            for every variable in the statement create a node and insert it into
            the varlist;
            set appropriate flags;
    (while brac <> 0)
    call sort(varlist);
    call edgbuild(head, edglist);
    call nodbuild(varlist);
    return(head);

```

C. Function sort

description

This function sorts the varlist in ascending order with respect to the nodenum.

Pseudo code

```

assign a temporary pointer to the head of the varlist;
sort the list in ascending order according to the nodnum;
return(vhead);

```

D. Function edgbuild**Description**

This function consider each node of the varlist.

If the node is a branching node, it insert a new edge in the edglist for each p-used variable.

Pseudocode

```

assign q to the head of the varlist;
use q to traverse the list and for each node do
    if q contains a predicate variable
        create a new node in the edglist corresponding to q and its first successor;
        create a new node in the edglist corresponding to q and its second
successor;
return(edglist);

```

E. Function nodbuild**Description**

This function delete all the node that contains a p-use from the varlist.

Pseudo code

```

assign q to the head of the varlist
use q to traverse the varlist and for every node do
    if (q->vtype = 'P')
        delete q from the varlist;
return(varlist);

```

F. Initialize**Description**

This function initialize the array arr[] to spaces.

Pseudo code

```

loop 250 times
    assign space to the current array cell;

```

G. Check1**Description**

This function check if the read character is a separator or an operator.

Pseudo code

```

If ((c is an operator) or ( c is a separator))
    flag = 1;
else
    flag = 0;

```

H. *Check2***Description**

This function checks if the read array is reserved C word.

Pseudo code

```
If (arr[] = if) or (arr = while) or (arr = for) or (arr = do) or (arr = scanf) or  
  ( arr = printf) or (arr = fscanf) or (arr = fprintf)  
    flag = 1;  
else  
    flag = 0;
```

I. *vtraverse***Description**

This function write the variable information on a file.

Pseudo code

```
assign vn to the head of the varlist;  
traverse the varlist using vn and for every node do  
  write(output file, node num, var name, var type);
```

J. *traverse***Description**

This function write the nodes information on a file.

Pseudo code

```
assign p to the head of the flowlist;  
traverse the flowlist using p and for every node do  
  write(output file, node num, node con1, node con2);
```

K. *Push***Description**

This function insert a node in the flowlist;

Pseudo code

```
If the list is empty  
  assign the head of the list to the new node;  
else  
  traverse the flowlist until you reach the last node;  
  assign the last node link to the new node;
```

L. *Push2*

Description

This function insert a node in the varlist;

Pseudo code

If the list is empty

 assign the head of the list to the new node;

else

 traverse the flowlist until you reach the last node;

 assign the last node link to the new node;

CHAPTER 5

TESTING

5.1. Data FLOW TESTING CRITERIA

Most path selection criteria are based on control flow analysis, which examines the branch and loop structure of a program. Those path selection criteria, have many problems such as dealing with loops structures. In this chapter we present a family of test data selection criteria which are based on data flow analysis and for which the number of paths selected is always finite, and chosen in a systematic and intelligent manner in order to help us uncover errors.

Rather than selecting program paths based solely on the control structure of a program, the data flow criteria track input variables through a program, following them as they are modified, until they are ultimately used to produce output values. In data flow analysis it is believed that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed.

Before discussing how to apply data flow analysis in selecting software test data, we will define some flow graph theoretic concepts [Rapps, Weyuker 1985].

5.1.1. Flow graph theoretic concepts

A path is a finite sequence of nodes (n_1, \dots, n_k) , in the program flow graph, such that there is an edge from n_i to n_{i+1} for all $i = 1, 2, k-1$. A path is simple if all nodes, except possibly the first and the last, are distinct. A path is loop-free if all nodes are distinct. A complete path is a path whose initial node is the program start node and whose final node is a program exit node.

A syntactically endless loop is a path (n_1, \dots, n_k) , $n_1 = n_k$, such that none of the blocks represented by the nodes on the path contain a conditional transfer statement whose target is either in a block which is not on the path or it is a halt statement. We assume that programs contain no syntactically endless loops.

Data flow testing criteria are based on an investigation of the ways in which values are associated with variables, and how these associations can affect the execution of the program. This analysis focuses on the occurrences of variables within the program. Each variable occurrence is classified as being a definitional, computation-use, or predicate-use occurrence. We refer to these as def, c-use, and p-use, respectively for example:

- The input statement 'read x ' contains a def of x .
- The output statement 'print x ' contains a c-use of x .
- The conditional transfer statement 'if $p(x)$ then ...' contains a p-use of x .

The c-use affects the computation being performed or allows one to see the result of an earlier definition. It may indirectly affect the flow control through the program. The p-use directly affects the flow of control through the program, and thereby may indirectly affects the computations performed.

since we are interested in tracing the flow of data between nodes, any definition which is used only within the node in which that definition occurs is of little importance to us. We thus make the following distinction: A c-use of a variable x is a global c-use, provided there is no definition of x preceding the c-use within the block in which it occurs. Otherwise it is a local c-use.

A conditional transfer statement is always the last statement of a block and has two executional successors, which are in two different blocks. Since the value of the variable occurring in the predicate portion of the conditional transfer statement directly determines which of these two blocks is to be executed next, we associate p-uses with edges rather than with the node in which the predicate portion occurs.

A path containing no defs of x , is called a def-clear path with respect to x . A path $(i, n_1, \dots, n_m, j, k)$ containing no defs of x in node n_1, \dots, n_m, j is called a def-clear path with respect to x from node i , to edge (j, k) . A def of a variable x in node i , is a global def if it is the last def of x occurring in the block associated with node i and there is a def-clear path with respect to x from node i to either a node containing a global c-use of x or to an edge containing a p-use of x . A def of a variable x in node i which is not a global def is a local

def if there is a local c-use of x in node i which follows this def, and no other def of x appears between the def and the local use.

We now define several sets needed in data flow testing and we will refer to the program found in Figure 4.1 in what follows:

Let i be any node and x any variable such that x belongs $\text{def}(i)$. Then $\text{dcu}(x, i)$ is the set of all nodes j such that x belongs to $\text{c-use}(j)$ and for which there is a def-clear path with respect to x from i to j , i.e. $\text{dcu}(d, 4) = \{6, 7\}$.

$\text{dpu}(x, i)$ is the set of all edges (j, k) such that x belongs to $\text{p-use}(j, k)$ and for which there is a def-clear path with respect to x from i to j , i.e. $\text{dpu}(e, 2) = \{3 \ 4, 3 \ 8\}$.

A path (n_1, \dots, n_j, n_k) is a du-path with respect to a variable x if n_1 has a global definition of x and either :

- n_k has a c-use of x and (n_1, \dots, n_j, n_k) is a def-clear simple path with respect to x .
 - (n_j, n_k) has a p-use of x and (n_1, \dots, n_j) is a def-clear loop-free path with respect to x .
- i.e. $(2, 3, 4, 5, 6, 8, 9, 10, 12)$ is a du-path with respect to e .

5.1.2. Data flow analysis

We now introduce a family of paths selection criteria based on data flow analysis [Rapps, weyuker 1985]:

Let G be a def-use graph, and P be a set of complete paths of G . Then :

- P satisfies the all-nodes criterion if every node of G is included in p
- P satisfies the all-edges criterion if every edge of G is included in P

- P satisfies the all-def criterion if for every node i of G and every x belongs to $\text{def}(i)$, P includes a def-clear path with respect to x from i to some element of $\text{dcu}(x, i)$ or $\text{dpu}(x, i)$
- P satisfies the all-puses criterion if for every node i and every x belongs to $\text{def}(i)$, P includes a def-clear path with respect to x from i to all elements of $\text{dpu}(x, i)$
- P satisfies the all-c-uses/some-p-uses criterion if for every node i and every x belongs to $\text{def}(i)$, P includes some def-clear path with respect to x from i to every node in $\text{dcu}(x, i)$; if $\text{dcu}(x, i)$ is empty, then P must include a def-clear path with respect to x from i to some edge contained in $\text{dpu}(x, i)$
- P satisfies the all-puses/some-c-uses criterion if for every node i and every x belongs to $\text{def}(i)$, P includes a def-clear path with respect to x from i to all elements of $\text{dpu}(x, i)$.
- P satisfies the all-uses criterion if for every node i and every x belongs to $\text{def}(i)$, P includes a def-clear path with respect to x from i to all element of $\text{dcu}(x, i)$ and to all elements of $\text{dpu}(x, i)$
- P satisfies the all-du-paths criterion if for every node i and every x belongs to $\text{def}(i)$, P includes every du-path with respect to x
- P satisfies the all-paths criterion if P includes every complete path of G .

5.2. ASSET

In 1985 a software tool for testing, ASSET, was developed [Phyllis, Frankl, Weiss and Weyuker 1985] based on the family of data flow test selection and test data adequacy criteria which we have discussed in section 5.1. ASSET, uses data flow information to aid in the selection and evaluation of test data for programs written in a small subset of Pascal

which has the structure and semantics shown in Figure 5.1 and it determines whether a given test set adequately tests a given program with respect to the chosen test data adequacy criteria.

ASSET's first step is the production of the subject program's def-use graph. On the first pass through the subject program, a table of label occurrences in label statements and their uses in transfer statements is constructed. On the second path, statements are classified as labeled or unlabeled, and as conditional transfer, Unconditional transfer, transfer, or other, and this information, along with the table, is used to divide the program into blocks and to produce the flow graph, which is represented as an array of adjacency lists.

```

<program>      ::= <program heading>;
                  <label declaration part>;
                  <variable declaration part>;
                  begin <stmt> {;<stmt>} end.
<stmt>         ::= [<label>:]<simple stmt>
<simple stmt>   ::= <assignment stmt>
                  |<conditional transfer stmt>
                  |<unconditional transfer stmt>
                  |<input stmt>
                  |<output stmt>
<assignment stmt>
                  ::= <identifier> := <expression>
<conditional transfer stmt>
                  ::= if <expression> then goto <label>
<unconditional transfer stmt>
                  ::= goto <label>
<input stmt>    ::= read [(<identlist>)]
                  |readln [(<identlist>)]
<output stmt>   ::= write [(<exprlist>)]
                  |writeln [(<exprlist>)]
<identlist>     ::= <identifier> [, <identlist>]
<exprlist>      ::= <expression> {,<exprlist>}
where
    <program heading>, <label declaration part>,
    <variable declaration part>, <label>,
    <identifier>, and <expression>
are defined as in Pascal.

```

Figure 5.1. The structure and semantics of the Pascal subset, the input to ASSET.

The flow graph provides the information needed to insert probes in the subject program. A statement of the form 'writeln (traversed<block number>)' is inserted into each block. The modified subject program is then compiled by a Pascal compiler and executed on each element of the test set. A record at each test is written, including the input, the output, and

the path traversed. These results can be examined by the person testing the program to determine whether the test was successful, i.e. whether the program met its specification on each test datum. The paths executed by the various test data are recorded in the file "traversed", separated by markers. These will be used later to determine whether or not the test fulfilled the given adequacy criterion.

ASSET uses the def-use graph to determine which pairs or paths are required by the given criterion. For the criteria all-def, all-p-uses, all-c-uses/some-p-uses, all-p-puses/some-c-uses and all-uses, this requires to construct the sets $dcu(x, i)$, and $dpu(x, i)$ which were defined before. These sets are constructed by performing a series of depth-first searches, one for each non-local definition of a variable. For example if the criterion is all-du-paths, then for each definition of variable x in node i , ASSET explores the graph in a depth-first manner, recording in a file every du-path from i to a node containing a c-use or an edge containing a p-use. ASSET next determines which, if any, of the pairs or paths required by the given adequacy criterion were not executed by the given test data.

5.3. Implementation

The first step in the implementation of the data flow testing criteria is to create the Modfile.c of the program. The Modfile.c is a modified version of the input program which will be executed by the tool on a set of test data in order to check if the selected test set satisfies the selected criterion. While building the control flow graph of the program, the Modfile.c is built in the following manner. Statements are copied as they are from the input program to the Modfile.c file, and at the beginning of each node a statement that indicates

the node number is added. The statement will have the form [fprintf(trvpt," node-number")]. the Modfile.c of the program found in Figure 4.1 is in Figure 5.2.

```
main()
{
FILE *trvpt,*tdr,*ovr;char tdata[15],odata[15]; int e,t,d,x,c;
trvpt = fopen("trvpath.dat","w");tdr = fopen("tname1.dat","r");fscanf(tdr,"%s",tdata);
ovr = fopen("tname2.dat","r");fscanf(ovr,"%s",odata);
fprintf(trvpt," 0 "); inpt= fopen( tdata,"r");onpt= fopen( odata,"w");
fprintf(trvpt," 1 ");
while(!feof(inpt)){
fprintf(trvpt," 2 "); fscanf(inpt,"%d %d",&d,&e);
x=0;c=2*d; fprintf(trvpt," 3 ");
while(d > e){
fprintf(trvpt," 4 ");d= d/2;t= c - (2*x+d);fprintf(trvpt," 5 ");
if(t > 0){
fprintf(trvpt," 6 ");c= 2*((c - (2*x+d)));}
else
{fprintf(trvpt," 7 "); x= x+d;
}}
fprintf(trvpt," 8 "); x= x +8; d= 8;
}fprintf(trvpt," 9 ");
if( x > 100)
{fprintf(trvpt," 10 "); x= x/2;}
else { fprintf(trvpt," 11 "); x= 9;}
fprintf(trvpt," 12 ");fprintf(onpt,"%d %d",d,e);
fclose(trvpt);fclose(inpt);fclose(onpt);
return ;
}
```

Figure 5.2. The Modfile.c of the program found in Figure 4.1

The second step is the construction of the $dcu(x,i)$ and the $dpu(x,i)$ sets. In order to build those two sets we perform a depth-first searches on the control flow graph of the program as follows. Starting at the start node of the program we mark the node i as visited, we take information about the node from the def-use table of the program. If the node i contain a definition of a variable x then a $dcu(x,i)$ and a $dpu(x,i)$ sets are constructed for that variable and they are assigned to the empty set. Then we go where the i 's conl point to. The edge

con1 is marked as visited and we take information about the edge from def-use table. if the edge contain a predicate use of the variable then the node is added to the dpu set of the variable. Then if the new visited node contain a c-use of the variable we add it to the c-use set of that variable. We repeat the same thing at each node until we reach an exit statement or a node where the variable is redefined; so we backtrack to the last visited node j and then we go where the j's con2 points to. and we repeat the whole process until we return to the start node.

Moreover when the user select a set of test data, the tool execute the Modfile.c of the program on this set of test data. The execution of the modfile.c will create a list of the nodes that were covered by this set of test data. For example if the program f05 is executed on the test set f05.T11, then the list of the covered node will be written on the file f05.P11.

When a data flow testing criteria is selected. The tool build a list of the nodes that are required by the selected criterion. Then this list of the covered node that was produced by the execution of the Modfile.c on the selected set of test data, is compared to the list of the required nodes. and a list of the nodes that are required by each of the data flow testing criteria but that are not yet covered by the test data is produced.

5.3.1. Structure chart

Figure 5.3 contains the structure chart of the testcov module of Figure 4.1.

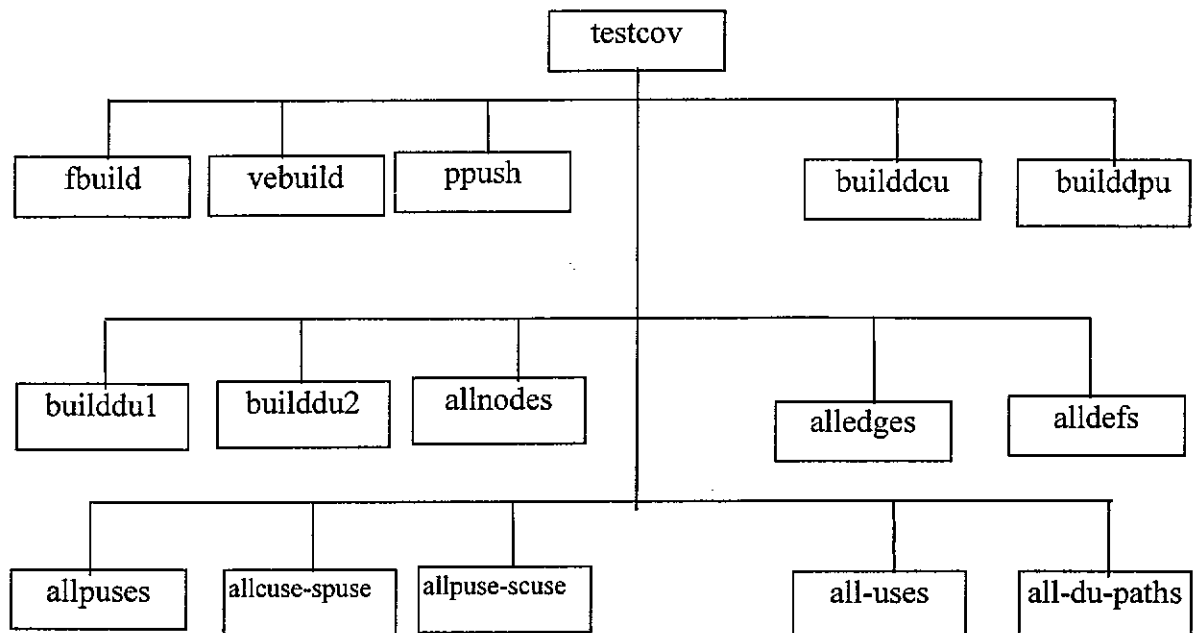


Figure 5.3. Structure chart of the testcov module.

5.3.2. Detailed design

A. Function testcov

Description

This function builds the $dcu(x,i)$ and $dpu(x,i)$ sets.

It also provides test coverage and checks for data flow criteria satisfaction.

Pseudo code

```

Initialize variables;
open input and output files;
call bulddcu(dculist);
call bulddpu(dpulist);
call bulddu1(dulist);
call bulddu2(dulist);
choose a data flow criterion ;
if choice = 1
    call allnode(nhead, phead)
  
```

```

    else
    if choice = 2
        call alledges(ehead, phead)
    else
    if choice = 3
        call alldefs(adhead, phead)
    else
    if choice = 4
        call allpuses(adhead, phead)
    else
    if choice = 5
        call allcuses-spuses(adhead, phead)
    else
    if choice = 6
        call allpuses-scuses(adhead, phead)
    else
    if choice = 7
        call all-uses(adhead, phead)
    else
    if choice = 8
        call all-du-paths(adhead, phead);
    print the list of nodes and paths needed by the selected criterion;
    close input and output files;
    return;

```

B. Function *all-uses*

Description

This function checks if the all-uses criterion is satisfied by the selected set of test data.

Pseudo code

```

assign dcuh to the head of dculist;
assign dpuh to the head of dpulist;
use dcuh to traverse the dculist and for every node do
    assign tp to the head of the node's dcuptr;
    use tp to traverse the node's dcuptr list and for every node do
        if the node is not covered by the test data then
            create a new node and push it in the adhead list;
use dpuh to traverse the dpulist and for every node do
    assign th to the head of the node's dpuptr;
    use th to traverse the node's dpuptr list and for every node do
        if the node is not covered by the test data then
            create a new node and push it in the adhead list;
return(adhead);

```


C. Function *all-du-paths***Description**

This function checks if the all-du-paths criterion is satisfied by the selected set of test data.

Pseudo code

```

found = 0;
assign dcuh to the head of the dulist;
traverse the dulist using dcuh and for each node do
  assign tp to the head of the node's dcuptr;
  traverse the node's dcuptr list and for each node do
    if the node is not traversed by the test data then
      found = 1;
  if found = 1 then
    insert the node's dcuptr list into the adhead list;
return(adhead);

```

D. Function *alldefs***Description**

This function checks if the alldefs criterion is satisfied by the selected set of test data.

Pseudo code

```

assign dcuh to the head of dculist;
found = 0;
use dcuh to traverse the dculist and for every node do
  assign tp to the head of the node's dcuptr;
  use tp to traverse the node's dcuptr list and for every node do
    if the node is covered by the test data then
      found = 1;

if found = 0 then
  assign dpuh to the dpulist node that have a definition of the same variable;
  assign th to the head of the node's dpuptr;
  use th to traverse the node's dpuptr list and for every node do
    if the node is covered by the test data then
      found = 1;
if found = 0 then
  create a new node for each of the tp and th lists and push it into the adhead list;
return(adhead);

```

E. Function *allcuses-spuse***Description**

This function checks if the allcuses-spuse criterion is satisfied by the selected set of test data.

Pseudo code

```

found = 0;
assign dcuh to the head of dculist;
use dcuh to traverse the dculist and for every node do
assign tp to the head of the node's dcuptr;
  if tp  $\neq$  null then
    use tp to traverse the node's dcuptr list and for every node do
      if the node is not covered by the test data then
        create a new node and push it in the adhead list;
      else

        assign dpuh to the node of the dpulist that have a definition of the same
        variable;
        assign th to the head of the node's dpuptr;
        use th to traverse the node's dpuptr list and for every node do
          if the node is covered by the test data then;
            found = 1;
    if found = 0 then
      create a new node for every th node and push it in the adhead list;
return(adhead);

```

F. Function *allpuses-scuse***Description**

This function checks if the allpuses-scuse criterion is satisfied by the selected set of test data.

Pseudo code

```

found = 0;
assign dpuh to the head of dpulist;
use dpuh to traverse the dpulist and for every node do
assign th to the head of the node's dpuptr;
  if th  $\neq$  null then
    use th to traverse the node's dpuptr list and for every node do
      if the node is not covered by the test data then
        create a new node and push it in the adhead list;
    else
      assign dcuh to the node of the dculist that have a definition of the same variable;
      assign tp to the head of the node's dcuptr;
      use tp to traverse the node's dcuptr list and for every node do

```

```

        if the node is covered by the test data then;
            found = 1;
    if found = 0 then
        create a new node for every tp node and push it in the adhead list;
    return(adhead);

```

G. Function allpuses

Description

This function checks if the allpuses criterion is satisfied by the selected set of test data.

Pseudo code

```

    use dpuh to traverse the dpulist and for every node do
    assign th to the head of the node's dpuptr;
    use th to traverse the node's dpuptr list and for every node do
        if the node is not covered by the test data then
            create a new node and push it in the adhead list;
    return(adhead);

```

H. Function allnodes

Description

This function checks if the allnodes criterion is satisfied by the selected set of test data.

Pseudo code

```

    assign p to the head of the flowlist;
    use p to traverse the flowlist and for every node do
        if the node is not traversed by the test data then
            create a new node and push it in the nhead list;
    return(nhead);

```

I. Function alledges

Description

This function checks if the alledges criterion is satisfied by the selected set of test data.

Pseudo code

```

    assign p to the head of the flowlist;
    use p to traverse the flowlist and for every edge do
        if the edge is not traversed by the test data then
            create a new node and push it in the ehead list;
    return(ehead);

```

J. Function *builddcu***Description**

This function checks if the *builddcu* criterion is satisfied by the selected set of test data.

Pseudo code

```

    initialise variables;
    assign q to the head of the flowlist;
    use q to traverse the flowlist and for each node do
        q->tag = 0;
        q->expl1 = 0;
        if(q->con2 = 0)
            q->expl2 = 1;
        else
            q->expl2 = 0;
    assign varp1 to the head of the varlist;
    use varp1 to traverse the varlist and for each node do
        if the node contain a definition of a variable then
            create a new node for the defined variable and push it in the dculist;
            traverse the flowlist until you reach the node where the variable is defined;
            starting at that node explore the flowlist in a depth first search maner

        for each visited node do

            if the node contain a c-use of the defined variable then
                create a new node for this variable and insert it in the c-use list of the
                defined variable
    return(dculist)

```

K. Function *builddpu***Description**

This function checks if the *builddpu* criterion is satisfied by the selected set of test data.

Pseudo code

```

    initialize variables;
    assign q to the head of the flowlist;
    use q to traverse the flowlist and for each node do
        q->tag = 0;
        q->expl1 = 0;
        if(q->con2 == 0)
            q->expl2 = 1;
        else
            q->expl2 = 0;

```

```

    assign varp1 to the head of the varlist;
    use varp1 to traverse the varlist and for each node do
    if the node contain a definition of a variable then
        create a new node for the defined variable and push it in the dpulist;
        traverse the flowlist until you reach the node where the variable is defined;
        starting at that node explore the flowlist in a depth first search maner

        for each visited edge do
            if the edge contain a p-use of the defined variable then
                create a new node for this variable and insert it in the p-use list of the
defined variable
    return(dpulist);

```

L. *Function buildu1*

Description

This function checks if a node contains a definition.

If the node contain a definition it associate with it a set of all definition clear paths from this node to all edges that contain a p-use of the defined variable.

Pseudo code

```

    initialize variables;
    assign varp1 to the head of the varlist;
    assign varp2 to the head of the edgelis;
    assign fhead to the head of the flowlist;
    traverse the flowlist using fhead in a depth first search manner and for each node
do
    use varp1 to see if the node contain a definition;
    if the node contain a definition then
        traverse edglist using varp2 and for each node do
            if the node contain a p-use of the variable and is not traversed by the test
data then
                create a new node and insert it into the dulist;
    return(dulist);

```

M. *Function buildu2*

Description

If the node contain a definition it associate with it a set of all definition clear paths from this node to all nodes that contain a c-use of the defined variable.

Pseudo code

```

    initialize variables;
    assign varp1 to the head of the varlist;
    assign varp2 to the head of the varlist;

```

```

    assign fhead to the head of the flowlist;
    traverse the flowlist using fhead in a depth first search manner and for each node
    do
        use varp1 to see if the node contain a definition;
        if the node contain a definition then
            traverse varlist using varp2 and for each node do
                if the node contain a c-use of the variable and is not traversed by the test
                data then
                    create a new node and insert it into the dulist;
    return(dulist);

```

N. Function fbuild

Description

This function reads from the control flow graph file and build the flowlist.

Pseudo code

```

    loop until the end of file
        create a new node;
        read( node number, con1, con2);
        insert the new node in the flowlist;
    return(flowlist);

```

O. Function Vebuild

Description

This function reads from the def-use graph file and build the varlist.

Pseudo code

```

    loop until the end of file
        create a new node;
        read( node number, var name, var type);
        insert the new node in the varlist;
    return(varlist);

```

P. Function ppush

Description

This function reads from the traversed path file and builds the pathlist.

Pseudo code

```

    loop until end of file
        create a new node;
        read( node number);
        insert the new node into the pathlist;
    return(pathlist);

```

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter we will work on a C program and show how ELISSAR works.

```
#include <io.h>
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#define null 0
FILE *inpt,*onpt;
main()
{
    int e,t,d,x,c;
        inpt = fopen("data.c","r");
        onpt = fopen("out.c","w");
    while(!feof(inpt))
    {
        fscanf(inpt,"%d %d",&d,&e);
        x=0;c=2*d;
        while(d > e)
        {
            d= d/2;
            t= c - (2*x+d);
            if(t > 0)
                c= 2*((c - (2*x+d)));
            else
                x = x+d;
        }
        x = x +8; d = 8;
    }
    if( x > 100)
        x = x/2;
    else
        x = 9;
    fprintf(onpt,"%d %d",d,e);
    fclose(inpt);
    fclose(onpt);
    return;
}
```

Figure 6.1. An example of a C program.

```

main()
{
FILE *trvpt,*tdr,*ovr;
char tdata[15],odata[15]; int e,t,d,x,c;
trvpt = fopen("trvpath.dat","w");
tdr = fopen("tname1.dat","r");fscanf(tdr,"%s",tdata);
ovr = fopen("tname2.dat","r");
fscanf(ovr,"%s",odata);
fprintf(trvpt," 0 ");
inpt= fopen( tdata,"r");onpt= fopen( odata,"w");
fprintf(trvpt," 1 ");
while(!feof(inpt)))
{
fprintf(trvpt," 2 ");
fscanf(inpt,"%d %d",&d,&e);
x=0;c=2*d;
fprintf(trvpt," 3 ");
while(d > e){
fprintf(trvpt," 4 ");
d= d/2;t= c - (2*x+d);
fprintf(trvpt," 5 ");
if(t > 0)
{
fprintf(trvpt," 6 ");
c= 2*((c - (2*x+d)));}
else
{
fprintf(trvpt," 7 ");
x= x+d;
}
}
fprintf(trvpt," 8 ");
x= x +8; d= 8;
}
fprintf(trvpt," 9 ");
if( x > 100)
{
fprintf(trvpt," 10 ");
x= x/2;
}
else
{
fprintf(trvpt," 11 ");
x= 9;
}
fprintf(trvpt," 12 ");
fprintf(onpt,"%d %d",d,e);
fclose(trvpt);fclose(inpt);fclose(onpt);
return ;
}

```

Figure 6.2. The Modfile.c of the program found in Figure 6.1

Figures 6.3 (a) and (b) contain the tabular form of the control flow graph of the program found in Figure 6.1 and 6.3 (c) contains its graphical representation.

NAME	# OF NODES	NODE #	CON 1	CON 2	TYPE
main					
		0	1		NORMAL
		1	3	8	CONTROL
		2	2	2	NORMAL
		3	4	8	CONTROL
		4	5		NORMAL
		5	6	7	CONTROL
		6	8		NORMAL
		7	8		NORMAL
		8	12		NORMAL
		9	10	11	CONTROL
		10	12		NORMAL
PRESS ANY KEY TO SEE NEXT PAGE PRESS THE LEFT BUTTON TO PROCEED RIGHT BUTTON / ESC to EXIT					

Figure 6.3. (a) The First Screen of the Tabular Form of the Control Flow Graph of Figure 6.1.

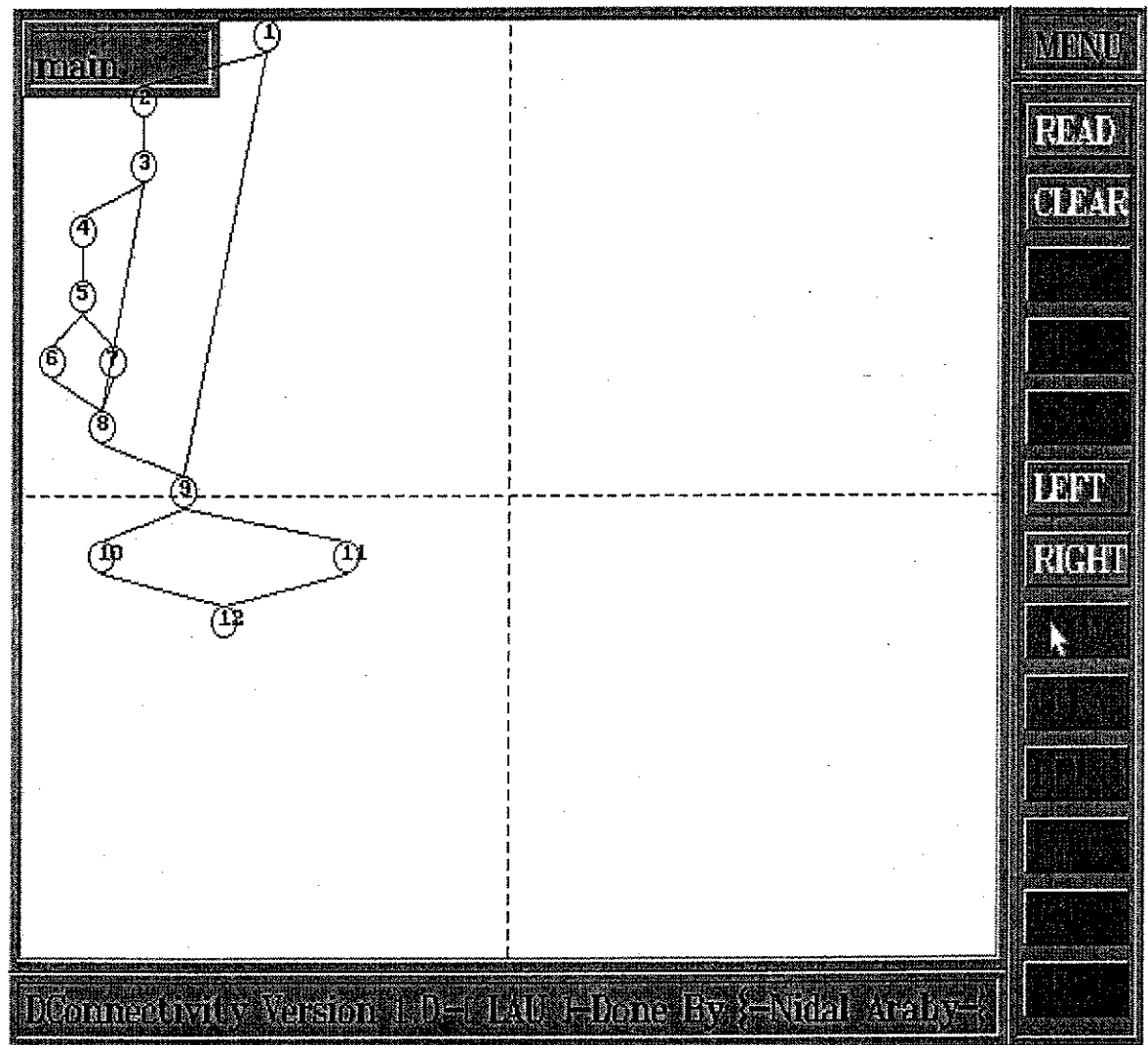


Figure 6.3. (c) The Third Screen of the Tabular Form of the Control Flow Graph of Figure 6.1.

Figures 6.4 (a), (b), (c) and (d) contain the tabular form of the def-use graph of the program found in Figure 6.1.

NAME	NODE#	EDGE#	VAR_NAME	TYPE
main				
	0			
	0			
	2			
	2			
	2			
	2			
	2			
	2			
	4			
	4			
	4			
	4			
PRESS ANY KEY TO SEE NEXT PAGE PRESS THE LEFT BUTTON TO PROCEED RIGHT BUTTON / ESC to EXIT				

Figure 6.4. (a) The First Screen of the Tabular Form of the Def-Use Graph of Figure 6.1.

NAME	NODE#	EDGE#	VAR_NAME	TYPE
main				
	4			
	4			
	6			
	6			
	6			
	6			
	7			
	7			
	7			
	10			
	6			
PRESS ANY KEY TO SEE NEXT PAGE PRESS THE LEFT BUTTON TO PROCEED RIGHT BUTTON / ESC to EXIT				

Figure 6.4. (b) The Second Screen of the Tabular Form of the Def-Use Graph of Figure 6.1.

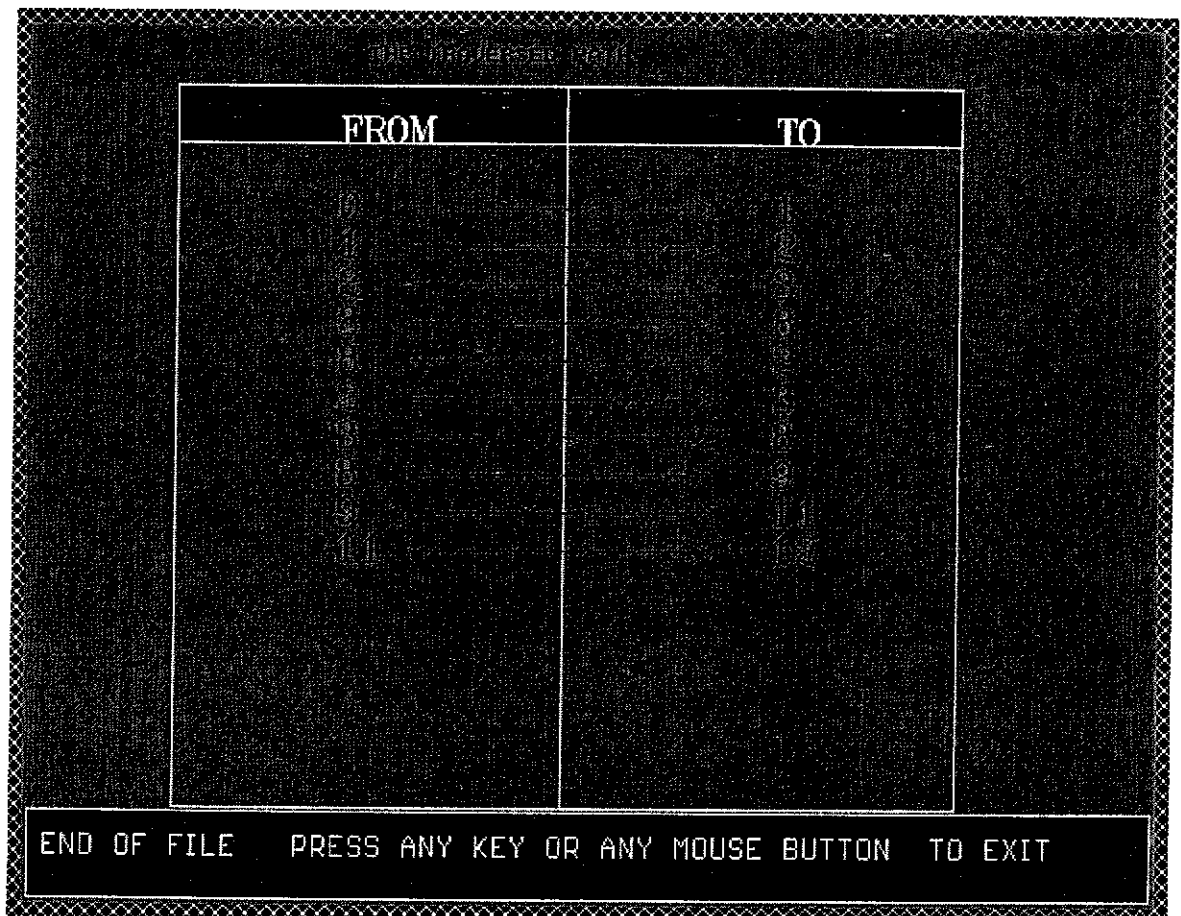
NAME	NODE#	EDGE#	VAR_NAME	TYPE
main				
	10			
	10			
	10			
	11			
	12			
	12			
	12			
	12			
	12			
		1 2		
		1 2		
		3 4		
PRESS ANY KEY TO SEE NEXT PAGE PRESS THE LEFT BUTTON TO PROCEED RIGHT BUTTON / ESC to EXIT				

Figure 6.4. (c) The Third Screen of the Tabular Form
of the Def-Use Graph of Figure 6.1.

NAME	NODE#	EDGE#	VAR_NAME	TYPE
main				
		3 8	2 8	main:main
		3 4	2 4	main:main
		3 8	3 8	main:main
		7 8	4 8	main:main
		5 7	5 7	main:main
		8 10	6 10	main:main
		9 11	7 11	main:main
END OF FILE PRESS ANY KEY OR ANY MOUSE BUTTON TO EXIT				

Figure 6.4. (d) The Fourth Screen of the Tabular Form of the Def-Use Graph of Figure 6.1.

Figure 6.5 contains the tabular form of the path traversed by the execution of the program on a selected test set.

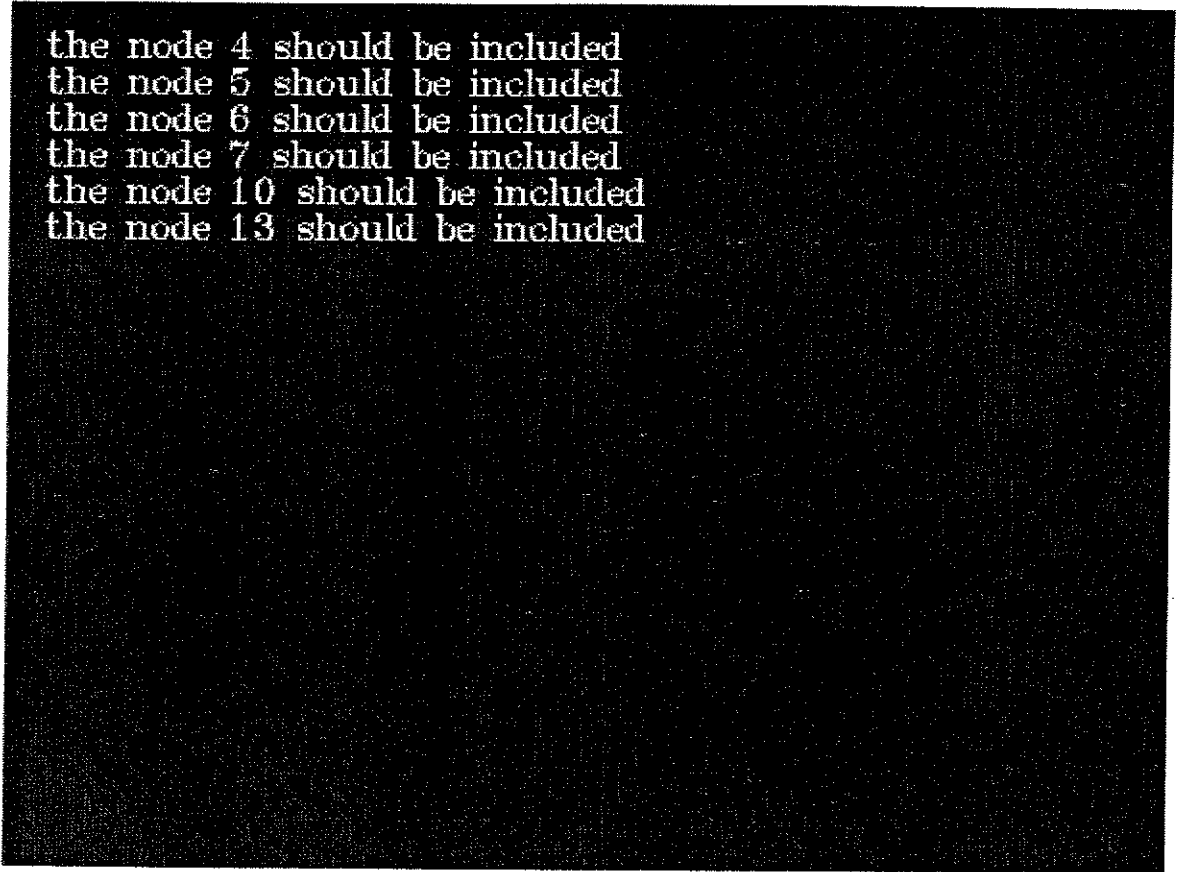


FROM	TO
10	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
10	1

END OF FILE PRESS ANY KEY OR ANY MOUSE BUTTON TO EXIT

Figure 6.5. The Test Traversed path Screen.

Figure 6.6 contains the list of nodes that were not traversed by the execution of the program on the selected test case and needed to satisfy the all-nodes criterion.



```
the node 4 should be included  
the node 5 should be included  
the node 6 should be included  
the node 7 should be included  
the node 10 should be included  
the node 13 should be included
```

Figure 6.6. List of nodes needed to satisfy the all-nodes criterion.

Figure 6.7 contains the list of paths that were not traversed by the execution of the program on the selected test case and needed to satisfy the all-defs criterion.

still must exercise at least one of the def-clear paths		
with respect to	from	to
x	2	4
x	2	6
and		
still must exercise at least one of the def-clear paths		
with respect to	from	to
c	2	4
and		
still must exercise at least one of the def-clear paths		
with respect to	from	to
d	4	6
d	4	7
and		
still must exercise at least one of the def-clear paths		
with respect to	from	to
t	4	5 6
t	4	5 7

Figure 6.7. List of paths needed to satisfy the all-defs criterion.

Figure 6.8 contains the test traversal file of the program found in Figure 6.1 and Figure 6.9 contains its variable information file.

```
11110000110111
11110000110111
11110000110111
11111111110111
11110000110111
11110000110111
11110000110111
11110000110111
```

Figure 6.8. Test traversal file of the program found in Figure 6.1.

```
0 inpt 1 0 1 0 0
0 onpt 1 0 1 0 0
1 inpt 1 1 1 0 1
2 d 1 0 0 1 0
2 e 1 0 0 1 0
2 x 1 0 0 0 0
2 c 1 0 0 0 0
2 d 1 2 0 0 0
3 d 1 1 0 0 1
3 e 1 1 0 0 1
4 d 1 0 0 0 0
4 d 1 2 0 0 0
4 t 1 0 0 0 0
4 c 1 2 0 0 0
4 x 1 2 0 0 0
4 d 1 2 0 0 0
5 t 1 1 0 0 0
6 c 1 0 0 0 0
6 c 1 2 0 0 0
6 x 1 2 0 0 0
6 d 1 2 0 0 0
7 x 1 0 0 0 0
7 x 1 2 0 0 0
7 d 1 2 0 0 0
8 x 1 0 0 0 0
8 x 1 2 0 0 0
8 d 1 0 0 0 0
9 x 1 1 0 0 0
10 x 1 0 0 0 0
10 x 1 2 0 0 0
11 x 1 0 0 0 0
12 d 1 2 0 2 0
12 e 1 2 0 2 0
12 inpt 1 2 1 0 0
12 onpt 1 2 1 0 0
-1
```

Figure 6.9. Variable information file of the program found in Figure 6.1.

CHAPTER 7

USER GUIDELINES

7.1. Syntax limitations

the input program is a compiled C program which have the following syntax limitations:

1. At least a space should separate the operators and operand
2. Switch statements are not allowed
3. The "{", "}" and else should be written on a line
4. Dynamic data structures are not allowed except for the name of the files
5. Goto and label statements are not allowed.
6. The input should be read from one file and the output should be written on one file.

7.2. File naming

The tool produces several useful documents pertaining to the subject program. If the input program is named PRG.c then all the produced output files will have PRG as file name and each file will have an extension that indicates what information are saved on this file. In what follows we will describe all the output files that are produced by the tool.

First, the tool produces the control flow graph and def-use graph of the input program in a tabulated form. The control flow graph is saved on a file which has `cfg` as extension and the def-use graph is saved on a file which has `dfg` as extension.

Second, each set of test data is saved on a file which has `Txx` as extension, where `xx` can be any number between 1 and 99. So the first set will have extension `T1` the second will have extension `T2` and so on. In this way each set of test data is saved on a file indicating the name of the program to test and the number of the test data set.

Third, the tool enable the user to execute the input program on any set of test data, and it saves the program output on a file which will have the extension `Oxx` where `xx` is the same number of the selected test data, and it saves the path traversed by the test data on a file which will have the extension `Pxx` where `xx` is also the same number of the selected test data. For example if the user execute the program `PRG.c` on the `PRG.T22`, the output data is saved on the file `PRG.O22` and the traversed path will be saved on the file `PRG.P22`.

Moreover, all the traversed paths are saved in a file which will have the extension `ttf`. Each line of the file will correspond to a traversed path and it will contains a series of zeroes and ones. For example if the `PRG.P5` contains the following path (1 3 5 7 9) then the 5th line of `PRG.ttf` contains (0 1 0 1 0 1 0 1 0 1) where:

- 0 indicates that the test case does not pass by that segment.
- 1 indicates that the test case passes by that segment.

The last file that the tool produces is the variable file which contains the variables information and the data flow information for the program and it is terminated by a -1 and has the extension `vif` and it has the following general format:

Seg, Variable, C, D-U, Type, IO flag, Loop flag, where:

- Seg = segment number
- Variable is the variable name
- C is a constant that is assumed to be always equal to 1
- D-U =
 - 0 - definition use.
 - 2 - computation use.
 - 1 - predicate use.
- Type =
 - 0 - if ordinary variable.
 - 1 - if pointer variable.
- IO flag=
 - 0 - if the variable has no I/O operations.
 - 1 - if the variable is read.
 - 2 - if the variable is written.
- Loop =
 - 0 - if no loop.
 - 1 - if there is a loop.

For example the variable information file of the program found in Figure 4.1 is in Figure 7.1.

```
0 inpt 1 0 1 0 0
0 onpt 1 0 1 0 0
1 inpt 1 1 1 0 1
2 d 1 0 0 1 0
2 e 1 0 0 1 0
2 x 1 0 0 0 0
2 c 1 0 0 0 0
2 d 1 2 0 0 0
3 d 1 1 0 0 1
3 e 1 1 0 0 1
4 d 1 0 0 0 0
4 d 1 2 0 0 0
4 t 1 0 0 0 0
4 c 1 2 0 0 0
4 x 1 2 0 0 0
4 d 1 2 0 0 0
5 t 1 1 0 0 0
6 c 1 0 0 0 0
6 c 1 2 0 0 0
6 x 1 2 0 0 0
6 d 1 2 0 0 0
7 x 1 0 0 0 0
7 x 1 2 0 0 0
7 d 1 2 0 0 0
8 x 1 0 0 0 0
8 x 1 2 0 0 0
8 d 1 0 0 0 0
9 x 1 1 0 0 0
10 x 1 0 0 0 0
10 x 1 2 0 0 0
11 x 1 0 0 0 0
12 d 1 2 0 2 0
12 e 1 2 0 2 0
12 inpt 1 2 1 0 0
12 onpt 1 2 1 0 0
-1
```

Figure 7.1. The variable information file of the program found in Figure 4.1.

CHAPTER 8

CONCLUSION

ELISSAR is a CASE tool for software testing and regression testing developed at LAU. Using ELISSAR for initial testing assists the user in designing test cases for a procedural language program according to dataflow testing criteria, i.e. it checks what dataflow testing criteria was satisfied by running the tested program on a test set and gives the user a summary of the paths needed to satisfy the rest of the dataflow criteria. Also, ELISSAR provides the user with the following :

- Tabular and graphical display of the control flow graph
- Def-use graphs of the tested program
- Test coverage information
- Ability to add new test cases
- Printing facility

ELISSAR can be also used for regression testing. The tester can use one of the following regression testing algorithms to verify that the modifications made to the program have not caused unintended side effects and that the modified system still meets the requirements:

- Reduction algorithm
- Firewall algorithm
- Genetic algorithm

- Simulated annealing algorithm
- Slicing algorithm
- Incremental algorithm.

REFERENCES

- Araby, N. 1997.** A Software Tool for Regression Testing. Thesis: Lebanese American University. February.
- Baradhi, G. 1996.** A Comparative Study of Regression Testing Methods. Thesis: Lebanese American University. June.
- Beizer, B. 1990.** Software Testing Techniques. 2ed. New York. Van Nostrand Reinhold.
- Dijkstra, E. W. 1972.** The Humble Programmer. Communications of the ACM. October, 859-866.
- Fakih, K., and Mansour, N. 1996.** A Genetic Algorithm for Corrective Retesting. Lebanese Scientific Research Reports, Vol. 1, No. 1, January, 5-19.
- Frankl, P., Weiss, S. and Weyuker, E. 1985.** Asset a system to Select and Evaluate Tests. IEEE Transactions on Software Engineering. 72-79.
- Rapps, S. and Weyuker, E. 1985.** Selecting Software Test Data Using Data flow Information. IEEE Transactions on Software Engineering. Vol. SE-11, No. 4, April, pp. 367-375.